Imperial College London
Department of Computing

Abstraction, Refinement and Concurrent Reasoning

Azalea Raad

September 2016

Supervised by Professor Philippa Gardner

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration of Originality

I herewith certify that all material in this thesis which is not my own work has been properly acknowledged.

AZALEA RAAD

Copyright

Abstract

This thesis explores the challenges in abstract library specification, library refinement and reasoning about fine-grained concurrent programs.

For abstract library specification, this thesis applies structural separation logic (SSL) to formally specify the behaviour of several libraries in an abstract, local and compositional manner. This thesis further generalises the theory of SSL to allow for library specifications that are language-independent. Most notably, we specify a fragment of the Document Object Model (DOM) library. This result is compelling as it significantly improves upon existing DOM formalisms in that the specifications produced are local, compositional and language-independent.

Concerning library refinement, this thesis explores two existing approaches to library refinement for separation logic, identifying their advantages and limitations in different settings. This thesis then introduces a hybrid approach to refinement, combining the strengths of both techniques for simple scalable library refinement. These ideas are then adapted to refinement for SSL by presenting a JavaScript implementation of the DOM fragment studied and establishing its correctness with respect to its specification using the hybrid refinement approach.

As to concurrent reasoning, this thesis introduces concurrent local subjective logic (CoLoSL) for compositional reasoning about fine-grained concurrent programs. CoLoSL introduces subjective views, where each thread is verified with respect to a customised local view of the state, as well as the general composition and framing of interference relations, allowing for better proof reuse.

"The isolated person does not develop any intellectual power. It is necessary for them to be immersed in an environment of other people, whose techniques they absorb during the first twenty years of their life. They may then perhaps do a little research of their own and make a very few discoveries which are passed on to others. From this point of view the search for new techniques must be regarded as carried out by the human community as a whole, rather than by individuals."

–*Alan Turing*

"A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine."

–*Alan Turing*

Acknowledgements

I am eternally grateful to my supervisor Philippa for her untiring support and her passion for excellence; to my friends and mentors at Imperial College for nurturing me emotionally and intellectually; to my collaborators Adam, Aquinas, José, Jules, Mark, Matt and Sophia for inspiring and challenging me; and to the formal verification community for motivating and influencing my work in countless ways.

I am immensely indebted to my mother Azar for her unconditional love, her devotion to my happiness and her sacrifices for bettering me; to my aunt and second mother Sabereh for igniting in me the passion for the pursuit of science and knowledge from an early age; and to my siblings Aydin and Alalea for inspiring me with their strength, intellect and dedication.

I am forever beholden to my best friend, colleague and partner in life Michael for his limitless love and loyalty, his drive for academic excellence, his ardour for moral and intellectual integrity, and his patience to persevere through the hardest times of the last decade with me.

Contents

List of Figures

List of Definitions

List of Theorems

1. Introduction

1.1. Contributions

The main contributions of this thesis are threefold, summarised as follows:

- **Abstraction.** Our first contribution is the application of *structural separation logic* (SSL) [60] to specify the behaviour of several libraries and to reason locally about their client programs. SSL is a program logic for the abstract specification of libraries that manipulate structured data, and local reasoning about their client programs. Most notably, we use SSL to specify a fragment of the Document Object Model (DOM) library and highlight the merits of our specifications over existing formalisms. We demonstrate that unlike the existing DOM specification [52, 24] in context logic (CL) [7, 6], our SSL specification is *compositional*, allowing for scalable client reasoning. In particular, we show that a simple DOM client program (comprising 3 lines of code) can be described with a *single* SSL specification, whilst the same program requires at least *six* separate CL specifications. Moreover, our SSL specification is more local than that of the CL specification, capturing the footprint of DOM operations more accurately. While the non-locality of the CL specification was noted by the authors in [52, 24] at the time, we are the first to discover and remedy its non-compositionality. We further generalise the SSL theory in [60] to allow for a *language-independent* library specification and client reasoning. More concretely, we generalise the SSL theory so that the specification of a library does not rely on the memory model of the programming language used to interact with the library. Rather, the library specification in SSL may be incorporated into an existing program logic as an add-on. This way, our library specification may be used to reason about different client programs of the library written in different programming languages.

- **Refinement.** Our second contribution is the application of refinement techniques in order to establish the correctness of library implementations. We explore two existing approaches to library refinement for separation logic, namely the locality-breaking and locality-preserving techniques. We discuss the merits and shortcomings of each approach. We demonstrate that the locality-breaking approach is neither scalable nor suitable in concurrent settings, whereas the locality-preserving approach overcomes both the scalability and concurrency limitations, albeit at the price of complex translations. We introduce a *hybrid* approach to refinement where we combine the strengths of both techniques for simple scalable refinement of libraries. We then present a JavaScript implementation of the DOM fragment formally specified in this thesis and establish its correctness with respect to its formal specification using our hybrid approach.

- **Concurrency.** Our third contribution is the program logic of CoLoSL (Concurrent Local Subjective Logic) for compositional reasoning about concurrent programs. We introduce the notion of *subjective views* where each thread is verified with respect to its customised local view of the state. Subjective views may arbitrarily overlap with one another, and may expand or contract in accordance with the thread footprint. We introduce the general *composition* and *framing* of interference relations (describing how the shared resources may be manipulated by each thread) in the spirit of resource composition and framing in standard separation logic. We demonstrate that this fluidity allows for better proof reuse. We use CoLoSL to reason about several graph-manipulating algorithms.

1.2. Publications

The following is a list of articles published as part of this PhD:

- **A sip of the Chalice**
  FTfJP 2011 [43]

- **Abstract Local Reasoning for Concurrent Libraries: Mind the Gap**
  MFPS 2014 [23]

- **CoLoSL: Concurrent Local Subjective Logic**
  ESOP 2015 [48]
- **DOM: Specification and Client Reasoning**
  APLAS 2016 [45]
- **Verifying Concurrent Graph Algorithms**
  APLAS 2016 [46]

1.3. Thesis Overview

- Chapter 2 introduces the background theory for the *abstraction* part of this thesis on which the following three chapters are based. We give an overview of program verification, separation logic and the existing abstraction techniques for library specification. We then provide an intuitive account of structural separation logic (SSL) [60] and demonstrate its merits for abstract library specification and local reasoning.
- Chapter 3 presents the SSL theory as given in [60]. We use SSL to specify a simple list library. The program logic of SSL in [60] is based on a simple while language. As such, the SSL library specifications are language dependent and may be used to reason about the client program written in this while language only. We generalise the approach in [60] and demonstrate how to integrate SSL with *any* separation logic (SL) based program logic and use it to reason about client program written in *any* language with an accompanying SL-based program logic (e.g. Java [42] and JavaScript [21]). That is, we describe how SSL may be incorporated into SL-based logics meeting certain conditions as an *add-on* with minimal change to their underlying models. To demonstrate this approach, we integrate SSL with an SL-based logic with a simple while language, a variable stack and a heap.
- Chapter 4 applies the generalised SSL theory introduced in §3 to specify a simple tree library inspired by the Document Object Model (DOM) library [2]. We then use the SL-based logic of the previous chapter to reason about several client programs of the tree library.
- Chapter 5 applies the general SSL theory in §3 to formally specify a fragment of the DOM Core Level 1 standard [1]. We demonstrate that

in comparison to existing formal specification [24, 52] of the DOM library in context logic [7, 6], our specification is both more *local* and *compositional*. Our specification is more local in that several DOM operations have smaller footprints in our specification, closely reflecting the intuitive resources needed by these operations. We demonstrate the compositionality of our specification via a simple DOM client program whose behaviour is specified by a *single* triple using our specification, compared to *six* triples using the existing DOM specification [24, 52]. We then use the general technique described in §3 to integrate our DOM SSL specification with the SL-based JavaScript program logic, JSLogic, introduced in [21]. We use our DOM extension of JSLogic to reason about several JavaScript ad blocker programs that call the DOM.

- Chapter 6 introduces the background theory for the *refinement* part of this thesis on which the following chapter is based. We give an overview of existing refinement techniques for library implementation. We demonstrate how to establish the correctness of a library implementation with respect to its abstract specification. We then provide an intuitive account of the two approaches to library refinement put forth by Dinsdale-Young et al in [16, 13, 59], namely the *locality-breaking* and *locality-preserving* approaches. We discuss the merits and shortcomings of each approach. We demonstrate that the locality-breaking approach is neither scalable nor suitable in concurrent settings, whereas the locality-preserving approach overcomes both the scalability and concurrency limitations, albeit at the price of complex translations. We then introduce a *hybrid* approach to refinement where we combine the strengths of both locality-breaking and locality-preserving techniques for simple scalable refinement of libraries.

- Chapter 7 presents a JavaScript implementation of the DOM fragment specified in §5. We then apply the hybrid refinement approach described in the previous chapter to demonstrate that our implementation correctly refines the DOM specification in §5 and satisfies the same specification.

- Chapter 8 introduces the background theory for the *concurrency* part of this thesis. We give an overview of the existing program logics for fine-grained concurrent reasoning and introduce the program logic of CoLoSL (Concurrent Local Subjective Logic) for compositional reasoning about concurrent programs. CoLoSL introduces the notion of *subjective views* where each thread is verified with respect to its customised local view of the global shared resources, describing only those parts of the state accessed by the thread. Subjective views may arbitrarily overlap with each other, and expand and contract depending on the resource required by the thread. Inspired by local rely-guarantee (LRG) reasoning [19], CoLoSL introduces the general *composition* and *framing* of interference relations (describing how the shared resources may be manipulated by each thread) in the spirit of resource composition and framing in standard separation logic.

- Chapter 9 presents the general theory of CoLoSL including its model, assertion language, programming language, reasoning framework, semantics and soundness.

- Chapter 10 applies CoLoSL to specify and reason about several fine-grained concurrent graph manipulating algorithms. In particular, we specify and verify several algorithms for computing the spanning tree of a graph, copying a heap-represented graph and a speculative parallel variant of Dijkstra's shortest path algorithm.

- Chapter 11 concludes this thesis with a summary of achievements and a discussion of future work.

1.4. Notational Conventions

We outline the basic notational conventions for standard mathematical concepts that are used in this thesis.

Notation (Types as sets)**.** We identify specific types (sets) by names in the Small-Caps format, and write for example Expr to identify a set of expressions. We use the terms "set" and "type" interchangeably depending on the context. Certain mathematical sets have their designated notation:

- $\emptyset$, for the empty set;

- $\mathbb{N} = \{0, 1, \dots\}$, for the set of natural numbers;
- $\mathbb{N}^+ = \{1, 2, \dots\}$, for the set of positive natural numbers; and
- $\mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$, for the set of integers.

We use the following standard set notations and write:

- $s \in \mathrm{S}$ and $s' \notin \mathrm{S}$ for set membership and its negation, respectively;
- $\mathrm{S}_1 \subseteq \mathrm{S}_2$ and $\mathrm{S}_1 \subset S_2$ for the subset and strict subset relations, respectively;
- $\mathrm{S}_1 \supseteq \mathrm{S}_2$ and $\mathrm{S}_1 \supset S_2$ for the superset and strict superset relations, respectively;
- $\mathrm{S}_1 \cap \mathrm{S}_2$ and $\mathrm{S}_1 \setminus \mathrm{S}_2$ for set intersection and set difference, respectively;
- $\mathrm{S}_1 \cup \mathrm{S}_2$ and $\mathrm{S}_1 \uplus \mathrm{S}_2$ for set union and disjoint union, respectively;
- $\mathrm{S}_1 \times \mathrm{S}_2$ for the Cartesian product of the sets S_1 and S_2;
- $|\mathrm{S}|$ for the cardinality of S;
- $\mathcal{P}(\mathrm{S})$ for the powerset of S;
- $\{f(x) \mid P(x)\}$ for set comprehension, denoting the set of values $f(x)$ for each x for which the proposition $P(x)$ holds; and
- $s : \mathrm{S}$ to denote that s is an instance of the type S, i.e. $s \in \mathrm{S}$;

When defining a type S for the first time, typically in the **Definition** or **Parameter** environments, we write $s \in \mathrm{S}$ to denote that the elements of S are ranged over by s and its variants such as s_1, s' and so forth. We write $\mathrm{S}\langle \mathrm{A} \rangle$ to denote a generic type with type parameter A. For instance, the powerset construction $\mathcal{P}(\mathbb{N})$ can be defined as $\textsc{PSet}\langle \mathbb{N} \rangle$, where given a type parameter A, then $\textsc{PSet}\langle \mathrm{A} \rangle \triangleq \{S \mid S \subseteq \mathrm{A}\}$.

Notation (Relations)**.** *Relations* are subsets of the Cartesian product of two (or more) sets. As relations are themselves sets, the set notations also apply to relations. We further appeal to the following standard relation notations. Let $R \in \mathcal{P}(\mathrm{A} \times \mathrm{B})$. We then write:

- $a\,R\,b$ for $(a, b) \in R$;
- $a \not R\, b$ for $(a, b) \notin R$;
- $R(a)$ for $\{b \mid a\,R\,b\}$, namely the R-image of a;
- $\mathit{dom}(R)$ for $\{a \mid \exists b.\ a\,R\,b\}$, namely the domain of R; and
- $\mathit{rng}(R)$ for $\{b \mid \exists a.\ a\,R\,b\}$, namely the range of R.

Notation (Functions)**.** The set of functions from A to B is denoted $\mathrm{A} \to \mathrm{B}$. The set of partial functions from A to B is denoted $\mathrm{A} \rightharpoonup \mathrm{B}$. The set of partial finite functions from A to B is denoted $\mathrm{A} \overset{\text{fin}}{\rightharpoonup} \mathrm{B}$. We use the following standard function notations. When $f \in \mathrm{A} \to \mathrm{B}$ or $f \in \mathrm{A} \rightharpoonup \mathrm{B}$, or $f \in \mathrm{A} \overset{\text{fin}}{\rightharpoonup} \mathrm{B}$, we write:

- $f(a)$ for the application of f to argument $a \in \mathrm{A}$;
- $\mathit{dom}(f) \triangleq \{a \mid f(a) \text{ is defined}\}$, for the domain of f; and
- $\mathit{rng}(f) \triangleq \{b \mid \exists a.\ f(a) \text{ is defined}\}$, for the range of f.

Notation (Lists)**.** Given a type parameter A, the generic type $\textsc{List}\langle\mathrm{A}\rangle$ delimits the set of all lists of A. The $\textsc{List}\langle\mathrm{A}\rangle$ is defined by the following inductive grammar, where $a \in \mathrm{A}$:

$$\textsc{List}\langle\mathrm{A}\rangle \ni L ::= [\,] \mid a : L$$

We use the following list notation and write:

- $[a, b, \ldots, z]$ for $a : (b : (\ldots z : [\,]))$;
- $[a_1, \ldots, a_n] \mathbin{+\!\!+} [a'_1, \ldots, a'_m]$ for $[a_1, \ldots, a_n, a'_1, \ldots, a'_m]$;
- $|L|$ for the length of list L, where $|[\,]| = 0$ and $|a : L| = 1 + |L|$; and
- $|L|^i$ for the i^{th} item of L (indexed from zero with $i \in \mathbb{N}$), defined as follows when $L \neq [\,]$ and is otherwise undefined:

$$|a : L|^0 \triangleq a \qquad\qquad |a : L|^{n+1} \triangleq |L|^n$$

Part I.

Abstraction and Refinement

2. Technical Background: Abstraction

Modern program verification as we know it began with the seminal work of Hoare in [29]. Although others had attempted to specify the behaviour of programs by writing pre- and postconditions describing the program state before and after the program execution, the pioneering work of Hoare in [29] was indeed the first attempt at a *formal* specification. Hoare further revolutionised the field by proposing the use of formal specifications to derive the specifications of larger programs without having to run their code.

The main idea behind Hoare's methodology was the identification of a core set of operations and the axiomatisation of their behaviour. The axioms of core operations are given as *Hoare triples* of the form $\{P\}\ \mathtt{C}\ \{Q\}$ where $\mathtt{C}$ denotes a core command and P and Q are first-order logical assertions describing the program state before and after the execution of $\mathtt{C}$. Hoare triples may have a partial or total interpretation. The partial interpretation of the $\{P\}\ \mathtt{C}\ \{Q\}$ states that if prior to the execution of $\mathtt{C}$ the program state satisfies P, then if and when $\mathtt{C}$ terminates the resulting program state satisfies Q. The total correctness interpretation additionally states that the execution of $\mathtt{C}$ does indeed terminate. It is more common to work with partial correctness interpretation of Hoare triples as termination proofs are challenging, especially in the presence of loops.

Combined with a set of rules for manipulating the Hoare triples, one can use the axiomatisation of the core operations to derive the specification of larger programs. For instance, one such is rule is that of sequential composition:

$$\frac{\{P\}\ \mathtt{C}_1\ \{R\} \qquad \{R\}\ \mathtt{C}_2\ \{Q\}}{\{P\}\ \mathtt{C}_1;\mathtt{C}_2\ \{Q\}}$$

The sequential composition rule states that if running $\mathtt{C}_1$ from a state satisfying P terminates in a state satisfying R, and if running $\mathtt{C}_2$ from a state satisfying R terminates in a state satisfying Q, then the resulting state from running $\mathtt{C}_1;\mathtt{C}_2$ (i.e. running $\mathtt{C}_1$ and $\mathtt{C}_2$ in succession) in a state satisfying

P satisfies Q (if $\mathtt{C}_1;\mathtt{C}_2$ terminates). Hoare reasoning has been studied extensively since its advent. However, the reasoning principles of Hoare logic do not scale to realistic programs. As mentioned earlier, the assertions in the pre- and postconditions of Hoare triples are written in first-order logic, describing the entire (global) program state. When dealing with large complex data structures, describing the entire program state is nontrivial. Moreover, in most realistic programming languages it is common to deal with memory pointers in order to dynamically modify the data structures. These constructs in turn may introduce complex aliasing relationships between memory pointers, further compounding the description of the global program states.

In 2001, O'Hearn, Reynolds and Yang reformed the field of program verification by introducing separation logic [32, 40, 49]. The formalisms hitherto studied advocated *global reasoning* where the program behaviour is specified with respect to the entire state. By contrast, separation logic was a champion of *local reasoning* where the behaviour of a program C is specified with respect to those parts of the state affected by C, namely the *footprint* of C. One can thus infer the behaviour of C on larger states as those parts beyond the footprint of C remain unchanged. This in turn allows one to combine the effects of multiple programs and specify the behaviour of more complex programs *compositionally.*

The novel idea behind separation logic is the introduction of the spatial connective $*$ known as the *separating conjunction.* The separating conjunction splits the program state (usually a heap) into to *separate* components. That is, $P*Q$ describes a state that can be split into two parts such that one part satisfies P and the other satisfies Q. Using the $*$ connective, one can express the disjointness and aliasing properties of program states concisely. For instance, we can write $x \mapsto y * y \mapsto x$ to describe two *distinct* heap cells ($x{\neq}y$) that reference one another. Note that by contrast the first-order assertion $x \mapsto y \wedge y \mapsto x$ merely describes the presence of two heap cells that may or may not be aliased. That is, it is not known whether $x{=}y$. Compared to traditional Hoare reasoning, expressing the disjointness of program states allows for far more tractable reasoning about pointer manipulating programs as one no longer needs to describe the aliasing relation between various memory locations.

In order to use the local specifications of separation logic in larger states, separation logic introduces the *frame* rule of inference:

$$\frac{\{P\}\ \mathtt{C}\ \{Q\}}{\{P * R\}\ \mathtt{C}\ \{Q * R\}}\ \mathsf{mod}(\mathtt{C}) \cap \mathsf{free}(R) = \emptyset$$

The frame rule states that, if executing $\mathtt{C}$ in a local state P transforms it to a state satisfying Q, then executing $\mathtt{C}$ in a larger state (satisfying $P * R$) will behave in the same way in that it will transform the local substate satisfying P to one satisfying Q while the extension substate (satisfying R) remains unchanged by the execution. The side condition ensures that the execution of $\mathtt{C}$ does not modify the state satisfying R by violating its assumptions about the values of program variables.

Later in [39], O'Hearn extended separation logic to allow for reasoning abut *concurrent* programs such as $\mathtt{C}_1 \,||\, \mathtt{C}_2$. The parallel composition construct $\mathtt{C}_1 \,||\, \mathtt{C}_2$ denotes the execution of the $\mathtt{C}_1$ and $\mathtt{C}_2$ programs in parallel by two distinct threads.

In order to reason about concurrent programs, O'Hearn introduced the parallel composition rule of inference:

$$\frac{\{P_1\}\ \mathtt{C}_1\ \{Q_1\} \quad \{P_2\}\ \mathtt{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathtt{C}_1 \,||\, \mathtt{C}_2\ \{Q_1 * Q_2\}}\ (\textsc{Par})$$

provided that $\mathtt{C}_1$ does not modify any variables free in $P_2, \mathtt{C}_2, Q_2$; similarly for $\mathtt{C}_2$ and $P_1, \mathtt{C}_1, Q_1$. The (Par) rule states that, if executing $\mathtt{C}_1$ in a local state P_1 transforms it to a state satisfying Q_1, and similarly executing $\mathtt{C}_2$ in a local state P_2 transforms it to a state satisfying Q_2, then executing $\mathtt{C}_1 \,||\, \mathtt{C}_2$ in a larger state satisfying $P_1 * P_2$ will transform it to a state satisfying $Q_1 * Q_2$. We can use the (Par) rule to prove that two programs, $\mathtt{C}_1$ and $\mathtt{C}_2$, which act on separate (compatible) parts of the state described by P_1 and P_2 can be executed in parallel. For instance, consider the following proof sketch for the program $\mathtt{C} \triangleq \mathtt{[x] = 1 \;||\; [y] = 2}$:

$$\begin{array}{c}
\{\mathtt{x} \mapsto - * \mathtt{y} \mapsto -\} \\
\begin{array}{l||l}
\{\mathtt{x} \mapsto -\} & \{\mathtt{y} \mapsto -\} \\
\mathtt{[x] = 1} & \mathtt{[y] = 2} \\
\{\mathtt{x} \mapsto 1\} & \{\mathtt{y} \mapsto 2\}
\end{array} \\
\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 2\}
\end{array}$$

What we have seen so far provides a rather low-level view of the program state. When reasoning about large software libraries, *abstraction* is as important as locality. While locality allows us to study each part of the system in isolation, abstraction affords us a simpler view of the system, hiding the irrelevant details. That is, abstraction allows us to take a concrete implementation of a module and produce its abstract specification. A client of the module can then appeal to this abstract specification and reason about it without having to understand the specifics of its implementation. We proceed with an overview of the existing work for combining abstraction techniques with separation logic.

2.1. Abstraction for Separation Logic

Separation logic is extensively studied and the benefits of the locality it affords in program verification is well-established. However, separation logic provides a low-level abstraction of the program state akin to that of physical machine states. To allow for more abstract specifications, Parkinson and Bierman extended separation logic with *abstract predicates* [41].

An abstract predicate is a *black box* description that provides a high-level representation of the underlying data structure without exposing its implementation details. For instance, consider variable stores that map program variables onto their associated values. The implementation of variable stores and their representation in memory varies from one programming language to another. For example, while in programming languages such as C program variables are allocated and stored on the stack, the variable store in JavaScript is emulated in the heap. However, when reasoning about client programs it is important to abstract away from the implementation details of the variable store and work instead with a more high-level representation of them. We can thus write an abstract predicate $\mathsf{vars}(\mathtt{x}_1 : \mathrm{v}_1, \mathtt{x}_2 : \mathrm{v}_2, \ldots, \mathtt{x}_n : \mathrm{v}_n)$ stating that the program variables $\mathtt{x}_1, \mathtt{x}_2, \ldots,$ $\mathtt{x}_n$ hold values denoted by the logical variables $\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_n$, respectively. The vars predicate thus provides a high-level representation of program variables that can be instantiated to describe a particular implementation.

As another example, consider specifying the operations of a set module. There are many different ways to represent a set in memory. For instance, amongst many other possible representations, we may choose to represent the set in memory as a singly-linked list or as a skip list. However, the choice of in-memory representation of the set does not affect the high-level behaviour of the set operations. As such, when reasoning about client programs of the set module, it is highly desirable to shield the user from the intricate implementation-specific details, and work instead with a more abstract representation of the sets. We can thus write an abstract predicate $\mathsf{set}(\mathcal{R}_s, \mathrm{S})$ stating that the contents of the set at heap address $\mathcal{R}_s$ is described by the mathematical set S. The abstract predicate $\mathsf{set}(\mathcal{R}_s, \mathrm{S})$ does not reveal the heap representation of the set and merely provides a high-level description of the set via the mathematical set S. The set predicate thus provides a high-level opaque representation of a set and may be instantiated to describe a particular implementation.

Although abstract predicates successfully introduce the benefits of abstraction to separation logic, additional axiomatisation is required in order to achieve the same degree of locality as that of separation logic at each abstraction level. For instance, consider the variable assignment operation $\mathtt{x} = \mathtt{y}$ updating the value of variable $\mathtt{x}$ to that of $\mathtt{y}$. Using the abstract predict vars described above, we can specify this operation locally as follows:

$$\left\{\mathsf{vars}(\mathtt{x} : \mathrm{X}, \mathtt{y} : \mathrm{Y})\right\}\ \mathtt{x} = \mathtt{y}\ \left\{\mathsf{vars}(\mathtt{x} : \mathrm{Y}, \mathtt{y} : \mathrm{Y})\right\}$$

Now let us suppose that we hold the $\mathsf{vars}(\mathtt{x} : \mathrm{X}, \mathtt{y} : \mathrm{Y}, \mathtt{z} : \mathrm{Z})$ resource and we wish to perform the $\mathtt{x} = \mathtt{y}$ operation. In order to verify $\mathtt{x} = \mathtt{y}$, the resources we hold must match those described by the precondition of $\mathtt{x} = \mathtt{y}$. In other words, we must *split* the $\mathsf{vars}(\mathtt{x} : \mathrm{X}, \mathtt{y} : \mathrm{Y}, \mathtt{z} : \mathrm{Z})$ predicate, frame off variable $\mathtt{z}$ and its value and thus arrive at $\mathsf{vars}(\mathtt{x} : \mathrm{X}, \mathtt{y} : \mathrm{Y})$ as required by the precondition of $\mathtt{x} = \mathtt{y}$ above. However, as the vars predicate is opaque and its low-level representation is hidden from us, we do not know how such splitting can be achieved. To remedy this, abstract predicates allow for further axiomatisation of their properties, including that of their splitting. For instance, we can introduce the following axiom in order to describe the splitting of variable stores where S_1 and S_2 describe sets of program variable and value pairs and $\mathrm{S}_1 \uplus \mathrm{S}_2$ ensures that S_1 and S_2 contain disjoint program variables:

$$\forall \mathrm{S}_1, \mathrm{S}_2.\ \mathsf{vars}(\mathrm{S}_1 \uplus \mathrm{S}_2) \Leftrightarrow \mathsf{vars}(\mathrm{S}_1) * \mathsf{vars}(\mathrm{S}_2)$$

Using the above axioms, we may now split the $\mathsf{vars}(\mathtt{x}:\mathrm{X}, \mathtt{y}:\mathrm{Y}, \mathtt{z}:\mathrm{Z})$ predicate as $\mathsf{vars}(\mathtt{x}:\mathrm{X}, \mathtt{y}:\mathrm{Y}) * \mathsf{vars}(\mathtt{z}:\mathrm{Z})$, frame off $\mathsf{vars}(\mathtt{z}:\mathrm{Z})$ and thus match the precondition of $\mathtt{x}=\mathtt{y}$ above. As such, in order to achieve the same degree of locality as that readily offered by separation logic, we must appeal to further axiomatisation. To better illustrate this and to understand the advantages and shortcomings of abstract predicates, we study two simple library modules, sets and lists, and specify their operations using abstract predicates.

2.1.1. Set Module

Consider a set module with three operations `add`, `remove` and `size`. For a given set at address `x`, the `x.add(n)` operation adds the element identified by `n` to the set if it does not contain `n`; otherwise it leaves the set unchanged. Similarly, the `x.remove(n)` operation removes the element identified by `n` from the set when it is contained in the set; otherwise it leaves the set unchanged. Lastly, the `r := x.size()` operation returns the size of the set in `r`.[1]

Using the vars and set abstract predicates described earlier, we can specify the behaviour of the set operations as follows:

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}:\mathcal{R}_s, \mathtt{n}:\mathrm{N})\\ *\ \mathsf{set}(\mathcal{R}_s, \mathrm{S})\end{array}\right\}\ \mathtt{x.add(n)}\ \left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}:\mathcal{R}_s, \mathtt{n}:\mathrm{N})\\ *\ \mathsf{set}(\mathcal{R}_s, \mathrm{S}\cup\{\mathrm{N}\})\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}:\mathcal{R}_s, \mathtt{n}:\mathrm{N})\\ *\ \mathsf{set}(\mathcal{R}_s, \mathrm{S})\end{array}\right\}\ \mathtt{x.remove(n)}\ \left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}:\mathcal{R}_s, \mathtt{n}:\mathrm{N})\\ *\ \mathsf{set}(\mathcal{R}_s, \mathrm{S}\setminus\{\mathrm{N}\})\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}:\mathcal{R}_s, \mathtt{n}:\mathrm{N}, \mathtt{r}:\mathrm{R})\\ *\ \mathsf{set}(\mathcal{R}_s, \mathrm{S})\end{array}\right\}\ \mathtt{r:=x.size()}\ \left\{\begin{array}{l}\exists \mathrm{R}.\ \mathsf{vars}(\mathtt{x}:\mathcal{R}_s, \mathtt{n}:\mathrm{N}, \mathtt{r}:\mathrm{R})\\ *\ \mathsf{set}(\mathcal{R}_s, \mathrm{S}) \land \mathrm{R}=|\mathrm{S}|\end{array}\right\}$$

In the postcondition of `x.add(n)`, the contents of the mathematical set S at heap address $\mathcal{R}_s$ is extended with value N. Analogously, in the postcondition of `x.remove(n)`, value N is removed from the mathematical set S at $\mathcal{R}_s$. Finally, in the postcondition of `x.size()`, the set S at $\mathcal{R}_s$ remains unchanged and the result `r` reflects the number of elements in S, i.e. $|\mathrm{S}|$.

Observe that the footprint of the `size` operation spans the *entire* mathematical set S in that without having each and every element in S, its size

[1]We write, for example, `r := x.size()` for `x.size(r)`. That is, `r` denotes a program variable in which the operation result is returned, and ‘`:=`’ does not necessarily correspond to assignment in the client programming language.

cannot be determined. The specification of the `size` operation above thus accurately captures its intuitive footprint. By contrast, it is possible to specify the behaviour of the `x.add(n)` and `x.remove(n)` operations more locally by focussing solely on the element denoted by `n`, rather than the entire set S. More concretely, we can specify the `x.add(n)` operation such that the pre- and postcondition respectively assert "the set may or may not contain `n`", and "the set does indeed contain `n`"; *mutatis mutandis* for `x.remove(n)`. As such, the specifications of the `add` and `remove` operations above greatly overestimate the footprint of their associated operations. This in turn hinders local and compositional reasoning, especially in concurrent settings.

For instance, consider the concurrent set program `x.add(17)||x.add(23)`. Intuitively, the footprints of `x.add(17)` and `x.add(23)` are limited to the set elements `17` and `23`, respectively. Since `17` and `23` are distinct elements, the footprints of the two operations are disjoint from one another. As such, using the (Par) rule above (see p. 37), we expect to be able to execute `x.add(17)` and `x.add(23)` in parallel. However, in the specification above, the footprint of the `add` operation spans the entire set $\mathsf{set}(\mathcal{R}_s, \mathrm{S})$. As the set predicate $\mathsf{set}(\mathcal{R}_s, \mathrm{S})$ cannot be duplicated as $\mathsf{set}(\mathcal{R}_s, \mathrm{S}) * \mathsf{set}(\mathcal{R}_s, \mathrm{S})$, we cannot execute the `x.add(17)` and `x.add(23)` programs in parallel, as expected.

In order to specify the set operations locally, we thus appeal to *more local* abstract predicates $\mathsf{in}(\mathcal{R}_s, \mathrm{N})$ and $\mathsf{out}(\mathcal{R}_s, \mathrm{N})$. The $\mathsf{in}(\mathcal{R}_s, \mathrm{N})$ predicate states that the set contains value N. Similarly, the $\mathsf{out}(\mathcal{R}_s, \mathrm{N})$ predicate states that the set does not contain value N. Using these predicates, we can specify the behaviour of `x.add(n)` and `x.remove(n)` as follows:

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{x} : \mathcal{R}_s, \mathtt{n} : \mathrm{N}) \\ * \ (\mathsf{in}(\mathcal{R}_s, \mathrm{N}) \vee \mathsf{out}(\mathcal{R}_s, \mathrm{N}))\end{array}\right\} \quad \mathtt{x.add(n)} \quad \left\{\begin{array}{l}\mathsf{vars}(\mathtt{x} : \mathcal{R}_s, \mathtt{n} : \mathrm{N}) \\ * \ \mathsf{in}(\mathcal{R}_s, \mathrm{N})\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{x} : \mathcal{R}_s, \mathtt{n} : \mathrm{N}) \\ * \ (\mathsf{in}(\mathcal{R}_s, \mathrm{N}) \vee \mathsf{out}(\mathcal{R}_s, \mathrm{N}))\end{array}\right\} \quad \mathtt{x.remove(n)} \quad \left\{\begin{array}{l}\mathsf{vars}(\mathtt{x} : \mathcal{R}_s, \mathtt{n} : \mathrm{N}) \\ * \ \mathsf{out}(\mathcal{R}_s, \mathrm{N})\end{array}\right\}$$

Ideally, since the `size` operation involves the *global* predicate $\mathsf{set}(\mathcal{R}_s, \mathrm{S})$, in order to reason about all set operations simultaneously, we need a mechanism that allows us to move back and forth between the global predicate $\mathsf{set}(\mathcal{R}_s, \mathrm{S})$ and the local predicates $\mathsf{in}(\mathcal{R}_s, \mathrm{N})$ and $\mathsf{out}(\mathcal{R}_s, \mathrm{N})$. To do this, we introduce the following axioms:

$$\begin{aligned}\forall \textsc{s}.\ \mathsf{set}(\mathcal{R}_s, \textsc{s}) &\iff \mathsf{ins}(\mathcal{R}_s, \textsc{s}) * \mathsf{outs}(\mathcal{R}_s, \mathbb{N} \setminus \textsc{s}) \\ \forall \textsc{s}, \textsc{n}.\ \mathsf{ins}(\mathcal{R}_s, \textsc{s} \uplus \{\textsc{n}\}) &\iff \mathsf{ins}(\mathcal{R}_s, \textsc{s}) * \mathsf{in}(\mathcal{R}_s, \textsc{n}) \\ \forall \textsc{s}, \textsc{n}.\ \mathsf{outs}(\mathcal{R}_s, \textsc{s} \uplus \{\textsc{n}\}) &\iff \mathsf{outs}(\mathcal{R}_s, \textsc{s}) * \mathsf{out}(\mathcal{R}_s, \textsc{n})\end{aligned} \tag{2.1}$$

That is, when $\textsc{s}$ is a set of natural numbers, the above axiom describes how the global resources of $\mathsf{set}(\mathcal{R}_s, \textsc{s})$ may be *split* into smaller local resources to reflect the elements in $\textsc{s}$ (via $\mathsf{in}(\mathcal{R}_s, \textsc{n})$) and those not in $\textsc{s}$ (via $\mathsf{out}(\mathcal{R}_s, \textsc{n})$).

By combining abstract predicates with a set of axioms for enforcing the structural *splitting* of resources (e.g. those in 2.1), we can recover the benefits of locality afforded by separation logic at different abstraction levels. However, while the *splitting* of resources comes for free in separation logic (via the $*$ connective), it must be explicitly axiomatised on a per example basis for each abstract data structure. Moreover, as the data structures grow more complex, the number of the required axioms as well as their complexity must also grow to achieve the same degree of expressivity and separation as that of separation logic. To better demonstrate this, we next study a list module and specify its operations using abstract predicates.

2.1.2. List Module

We will shortly study a generic list module and specify its operations using abstract predicates. Lists are widely used data structures and have long been the subject of study in separation logic. In particular, consider the following inductive definition of lists first introduced as a motivating example for separation logic:

$$\begin{aligned}\mathit{list}(\mathcal{R}_l, \textsc{l}) &\triangleq \mathit{lseg}(\mathcal{R}_l, \textsc{l}, \mathtt{null}) \\ \mathit{lseg}(\textsc{x}, \textsc{l}, \textsc{n}) &\triangleq (\textsc{x}{=}\textsc{n} \land \textsc{l}{=}[\,] \land \mathsf{emp}) \\ &\quad \lor (\exists \textsc{y}, \textsc{v}, \textsc{l}'.\ \textsc{l}{=}\textsc{v}{:}\textsc{l}' \land \textsc{x} \mapsto \textsc{v}, \textsc{y} * \mathit{lseg}(\textsc{y}, \textsc{l}', \textsc{n}))\end{aligned}$$

The *list* and *lseg* predicates above are well-known in the separation logic community and are a staple in separation logic teachings. The $\mathit{list}(\mathcal{R}_l, \textsc{l})$ predicate describes a list at heap address $\mathcal{R}_l$ with its contents captured by the mathematical list $\textsc{l}$. The $\mathit{lseg}(\textsc{x}, \textsc{l}, \textsc{n})$ predicate describes a list *segment* $\textsc{l}$ between heap addresses $\textsc{x}$ and $\textsc{n}$, and provides a mechanism for splitting lists and list segments into smaller segments, thus enabling local reasoning. More concretely, for any list segments L_1 and L_2 we have:

$$\mathit{lseg}(\text{X}, \text{L}_1{+\!+}\text{L}_2, \text{N}) \iff \exists \text{Y}.\ \mathit{lseg}(\text{X}, \text{L}_1, \text{Y}) * \mathit{lseg}(\text{Y}, \text{L}_2, \text{N}) \tag{2.2}$$

The use of the mathematical list L in conjunction with the splitting property above is reminiscent of abstract predicates studied so far and at first glance may suggest that the *list* and *lseg* predicates are abstract (i.e. implementation independent). However, on closer inspection we note that the *lseg* predicate (and consequently the *list* predicate) exposes the in-heap representation of the underlying list as a singly-linked list, and thus may only be used when reasoning about singly-linked lists. As such, these predicates are not abstract enough and cannot be used to specify the behaviour of a generic list module. More concretely, the N parameter in $\mathit{lseg}(\text{X}, \text{L}, \text{N})$ describes the value of the "next" pointer for the last list node in the segment. Were we to reason about a doubly-linked list instead, we would then need to redefine both predicates as follows by extending them with a new parameter P tracking the value of the "previous" pointer for the first node in the segment:

$$\begin{aligned}
\mathit{dlist}(\mathcal{R}_l, \text{L}) &\triangleq \mathit{dlseg}(\mathcal{R}_l, \text{L}, \texttt{null}, \texttt{null}) \\
\mathit{dlseg}(\text{X}, \text{L}, \text{N}, \text{P}) &\triangleq (\text{X}{=}\text{N} \land \text{L}{=}[\,] \land \mathsf{emp}) \\
&\quad \lor (\exists \text{Y}, \text{V}, \text{L}'.\ \text{L}{=}\text{V}{:}\text{L}' \land \text{X} \mapsto \text{V}, \text{Y}, \text{P} * \mathit{dlseg}(\text{Y}, \text{L}', \text{N}, \text{X}))
\end{aligned}$$

In other words, by exposing the *connectivity* information of a list segment via the N and P parameters (i.e. how a list segment connects with its surrounding data), we have to redefine our predicates each time we consider a different list representation. Ideally, we would like to abstract this connectivity and work instead with a general list predicate that may be implemented by various list representations.

To this end, rather than *defining* the list predicates by describing their heap representation (as in *list* and *lseg* above), we assume two abstract predicates list and lfrag (list fragment), analogous to *list* and *lseg*, and hide their heap representation. To enable local reasoning, we must then *axiomatise* how a list may be split into smaller fragments. In other words, while the splitting of *list* follows immediately from its in-heap definition, we must axiomatise this property for the abstract list predicate list as follows:

$$\forall \text{L}_1, \text{L}_2.\ \mathsf{list}(\mathcal{R}_l, \text{L}_1{+\!+}\text{L}_2) \iff \exists\alpha.\ \mathsf{list}(\mathcal{R}_l, \text{L}_1{+\!+}\alpha) * \mathsf{lfrag}(\alpha, \text{L}_2)$$

That is, we can split off an arbitrary L_2 fragment of the list, provided that we track how this fragment fits in its associated list via α. The α in $\mathsf{list}(\mathcal{R}_l, \text{L}_1{+\!+}\alpha)$ identifies a *context hole* denoting the position to which the list fragment at α is to return. Conversely, the counterpart α in $\mathsf{lfrag}(\alpha, \text{L}_2)$ identifies an *abstract address* denoting the location at which the data associated with the context hole α is stored. The address α is abstract in that unlike $\mathcal{R}_l$ (or X and N in $\mathit{lseg}(\text{X}, \text{L}, \text{N})$), it does not correspond to an actual address in the heap and is a mere product of the abstraction. Observe that by describing the splitting of list data via an abstract address α, we abstract the *connectivity* of lists and list fragments to their surrounding data without exposing their heap representations. As we describe shortly, when we consider a particular list implementation (e.g. a singly-linked list), we also concretise this abstract connectivity captured by α.

In order to move between list fragments of various lengths and break list fragments into smaller ones, we use the following axiom:

$$\forall \text{L}_1, \text{L}_2, \beta.\ \big(\mathsf{lfrag}(\beta, \text{L}_1{+\!+}\text{L}_2) \iff \exists\alpha.\ \mathsf{lfrag}(\beta, \text{L}_1{+\!+}\alpha) * \mathsf{lfrag}(\alpha, \text{L}_2)\big)$$

We can now use the list and lfrag abstract predicates to specify the behaviour of a generic list module. We can then use the list specification in conjunction with the splitting properties above in order to enable local client reasoning. To reason about a particular implementation of the list module, we can *instantiate* the list and lfrag predicates by describing their heap representation e.g. as a singly-linked list or a doubly-linked list, defined respectively as *list* and *dlist* above. In doing so, we also concretise the abstract connectivity captured by the abstract address α. In case of the singly-linked list representation the concrete connectivity is realised by a single heap address N denoting the value of the “next” pointer for the last node in the fragment. In case of the doubly-linked list representation the concrete connectivity is realised by two heap addresses N and P, where N denotes the value of the “next” pointer for the last node in the fragment, and P captures the value of the “previous” pointer for the first node in the fragment.

Note that in the same spirit as the *lseg* splitting property in (2.2), in the abstract list splitting axioms above the splitting of the list data is allowed from the right end only. As such, given a list fragment of the form $\mathsf{lfrag}(\alpha, \text{L}_1{+\!\!+}\text{L}_2{+\!\!+}\text{L}_3)$, in order to get at the L_2 fragment we must use the splitting axiom twice as follows:

$$\begin{aligned}\mathsf{lfrag}(\alpha, \text{L}_1{+\!\!+}\text{L}_2{+\!\!+}\text{L}_3) &\Leftrightarrow \exists\beta.\ \mathsf{lfrag}(\alpha, \text{L}_1{+\!\!+}\beta) * \mathsf{lfrag}(\beta, \text{L}_2{+\!\!+}\text{L}_3)\\ &\Leftrightarrow \exists\beta, \gamma.\ \mathsf{lfrag}(\alpha, \text{L}_1{+\!\!+}\beta) * \mathsf{lfrag}(\beta, \text{L}_2{+\!\!+}\gamma) * \mathsf{lfrag}(\gamma, \text{L}_3)\end{aligned}$$

Although this is not a significant restriction, we generalise the splitting axioms as follows and use them instead in the remainder of this section:

$$\begin{aligned}&\forall \text{L}_1, \text{L}_2, \text{L}_3.\\ &\quad \mathsf{list}(\mathcal{R}_l, \text{L}_1{+\!\!+}\text{L}_2{+\!\!+}\text{L}_3) \iff \exists\alpha.\ \mathsf{list}(\mathcal{R}_l, \text{L}_1{+\!\!+}\alpha{+\!\!+}\text{L}_3) * \mathsf{lfrag}(\alpha, \text{L}_2) \qquad (2.3)\\ &\forall \text{L}_1, \text{L}_2, \text{L}_3, \beta.\\ &\quad \mathsf{lfrag}(\beta, \text{L}_1{+\!\!+}\text{L}_2{+\!\!+}\text{L}_3) \iff \exists\alpha.\ \mathsf{lfrag}(\beta, \text{L}_1{+\!\!+}\alpha{+\!\!+}\text{L}_3) * \mathsf{lfrag}(\alpha, \text{L}_2) \qquad (2.4)\end{aligned}$$

Let us now turn our focus to a generic list module and specify its operations using the list and lfrag abstract predicates. Consider a list module with three operations `add`, `remove` and `item`. For a given list at address `x`, the `x.add(n)` operation appends `n` to the end of the list. The `x.remove(n)` operation removes the first occurrence of `n` from the list when the list contains `n`; otherwise the operation faults. The `r := x.item(i)` operation returns the $\texttt{i}^{\text{th}}$ element of the list (indexed from 0) in `r` when `i` denotes a non-negative integer. When `i` is out of bounds (i.e. greater than or equal to the length of the list), then `null` is returned in `r`. This operation faults if `i` denotes a negative value.

As with the set module, it is possible to specify all list operations by appealing to a global abstract predicate $\mathsf{list}(\mathcal{R}_l, \text{L})$, stating that the contents of the list at `x` are described by the mathematical list L. However, to allow local reasoning we must accurately capture the operation footprints and choose our abstract predicates accordingly. Let us then turn our focus to the operation footprints.

The locality of footprints varies from one operation to another. Since lists allow for duplicate elements, the `x.add(n)` operation does not need to know of other `n` elements in the list. That is, the `x.add(n)` operation adds `n` to the end of the list regardless of other list elements. As such, the precondition of `x.add(n)` must ideally accommodate access to the end of the list without exposing its elements. The footprint of `x.add(n)` thus contains no tangible resources associated with the underlying list and merely requires access to the last position in the list. When the list contains the value denoted by `n`, the footprint of `x.remove(n)` is limited to the list fragment between the beginning of the list and the first occurrence of `n` (inclusive). Lastly, the footprint of the `x.item(i)` operation depends on the value of the index `i`. When `i` denotes a valid index (i.e. a non-negative integer less than the length of the list), then the footprint of `x.item(i)` comprises all elements from the beginning of the list up to and including the $\texttt{i}^{\text{th}}$ element. When `i` is greater than or equal to the length of the list, in order to calculate the length of the list and compare it against the index `i`, the footprint of `x.item(i)` spans the entire list.

Using the list and lfrag abstract predicates, we can specify the list operations as follows, capturing the footprints locally. For a binary operator $\oplus \in \{\in, =, \leq, \dots\}$, we write $\mathrm{X} \dot{\oplus} \mathrm{Y}$ for $\mathrm{X} \oplus \mathrm{Y} \wedge \mathsf{emp}$.

$$\left\{\begin{array}{l}\mathsf{vars}(\texttt{x}\!:\!\mathcal{R}_l, \texttt{n}\!:\!\mathrm{N})\\ * \,\mathsf{list}(\mathcal{R}_l, \alpha)\end{array}\right\} \quad \texttt{x.add(n)} \quad \left\{\begin{array}{l}\mathsf{vars}(\texttt{x}\!:\!\mathcal{R}_l, \texttt{n}\!:\!\mathrm{N})\\ * \,\mathsf{list}(\mathcal{R}_l, \alpha +\!\!+ [\mathrm{N}])\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\texttt{x} : \mathcal{R}_l, \texttt{n} : \mathrm{N})\\ * \,\mathsf{list}(\mathcal{R}_l, \mathrm{L} +\!\!+ [\mathrm{N}] +\!\!+ \alpha)\\ * \,\mathrm{N} \dot{\notin} \mathrm{L} * \mathsf{complete}(\mathrm{L})\end{array}\right\} \quad \texttt{x.remove(n)} \quad \left\{\begin{array}{l}\mathsf{vars}(\texttt{x} : \mathcal{R}_l, \texttt{n} : \mathrm{N})\\ * \,\mathsf{list}(\mathcal{R}_l, \mathrm{L} +\!\!+ [\mathrm{N}])\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\texttt{x}\!:\!\mathcal{R}_l, \texttt{i}\!:\!\mathrm{I}, \texttt{r}\!:\!\mathrm{R})\\ * \,\mathsf{list}(\mathcal{R}_l, \mathrm{L} +\!\!+ [\mathrm{N}] +\!\!+ \alpha)\\ * \,\mathrm{I} \dot{=} |\mathrm{L}| * \mathsf{complete}(\mathrm{L})\end{array}\right\} \quad \texttt{r := x.item(i)} \quad \left\{\begin{array}{l}\mathsf{vars}(\texttt{x}\!:\!\mathcal{R}_l, \texttt{i}\!:\!\mathrm{I}, \texttt{r}\!:\!\mathrm{N})\\ * \,\mathsf{list}(\mathcal{R}_l, \mathrm{L} +\!\!+ [\mathrm{N}] +\!\!+ \alpha)\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\texttt{x}\!:\!\mathcal{R}_l, \texttt{i}\!:\!\mathrm{I}, \texttt{r}\!:\!\mathrm{R})\\ * \,\mathsf{list}(\mathcal{R}_l, \mathrm{L}) * \mathrm{I} \dot{\geq} |\mathrm{L}|\\ * \,\mathsf{complete}(\mathrm{L})\end{array}\right\} \quad \texttt{r := x.item(i)} \quad \left\{\begin{array}{l}\mathsf{vars}(\texttt{x}\!:\!\mathcal{R}_l, \texttt{i}\!:\!\mathrm{I}, \texttt{r}\!:\!\texttt{null})\\ * \,\mathsf{list}(\mathcal{R}_l, \mathrm{L})\end{array}\right\}$$

Recall that for $\texttt{n}=\mathrm{N}$, the footprint of `x.add(n)` is independent of the list elements and solely requires access to the last position in the list. To specify this, we use the $\mathsf{lfrag}(\mathcal{R}_l, \alpha)$ predicate in the precondition of `x.add(n)` to assert that the contents of the list have been split off, leaving behind the context hole α. Observe that for any list L, we can start with the list $\mathsf{list}(\mathcal{R}_l, \mathrm{L})$, split off the contents of the list, and obtain $\mathsf{list}(\mathcal{R}_l, \alpha)$ using the axiom in (2.3). That is, $\mathsf{list}(\mathcal{R}_l, \text{L}) \iff \exists\alpha.\ \mathsf{list}(\mathcal{R}_l, \alpha) * \mathsf{lfrag}(\alpha, \text{L})$. In the postcondition, the list is simply extended with the new element N.

When `n`=N and the list contains N, the footprint of `x.remove(n)` comprises the fragment from the beginning of the list up to and including the first occurrence of N. This is captured by $\mathsf{list}(\mathcal{R}_l, \text{L}{+\!\!+}[\text{N}]{+\!\!+}\alpha)$ in the precondition. The $\text{N} \notin \text{L}$ in the precondition ensures that the N given is indeed the first occurrence of N in the list and does not occur in L. To ensure that the N does not occur in L, the list fragment described by L must be *complete* (i.e. contain no context holes). Were this not the case, we could not ascertain that N is not contained in L (e.g. when $\text{L}=\alpha{+\!\!+}\text{L}'$, the list fragment at α may also contain N). To capture this, we appeal to the following inductive predicate $\mathsf{complete}(\text{L})$, asserting that L is a complete list fragment with no context holes:

$$\mathsf{complete}(\text{L}) \triangleq (\text{L}\dot{=}[\,]) \vee (\exists \text{H}, \text{L}'.\ \text{L}\dot{=}[\text{H}]{+\!\!+}\text{L}' * \mathsf{complete}(\text{L}')\ )$$

The $\mathsf{complete}(\text{L})$ predicate states that either the list L is empty and thus the contents of the list are captured by $[\,]$; or the list has at least one element H, and the tail of the list, L', is also complete. This is captured by the $\text{L}\dot{=}[\,]$ and $\text{L}\dot{=}[\text{H}]{+\!\!+}\text{L}'$ assertions respectively, where $[\,]$ and $[\text{H}]{+\!\!+}\text{L}'$ denote logical expressions. In the postcondition the N is removed from the list as required.

Recall that when `i`=I denotes a valid index (non-negative and less than the length of the list), the footprint of `x.item(i)` comprises the fragment from the beginning of the list up to and including the I^{th} element. This is specified in the precondition of the first axiom by $\mathsf{list}(\mathcal{R}_l, \text{L}{+\!\!+}[\text{N}]{+\!\!+}\alpha) * |\text{L}| \dot{=} \text{I}$. As before, to ensure that the L is equal to I, the list fragment given by L must be complete, as stipulated by $\mathsf{complete}(\text{L})$ in the precondition. In the postcondition the I^{th} element (i.e. N) is returned in `r`.

When `i`=I is out of bounds (i.e. greater than or equal to the length of the list), the footprint of `x.item(i)` spans the entire list as specified by $\mathsf{list}(\mathcal{R}_l, \text{L}) * \text{I} \dot{\geq} |\text{L}|$ in the precondition of the second axiom. In the postcondition, the list remains unchanged and `null` is returned in `r` as required.

The above specification of the list operations is both *abstract* and *local.* That is, the list and lfrag abstract predicates allowed us to specify the list operations in a way that is independent of the underlying list implementation. However, the locality of our specifications comes at the price of explicitly axiomatising the list splitting properties in (2.3)-(2.4). In general, splitting abstract data structures is crucial to local reasoning. As such, it is highly desirable to write our specifications in a logic where this splitting is derivable within the logic itself, rather than having to be axiomatised separately for each abstract data structure. This is precisely the motivation behind *structural separation logic* [60]. Structural separation logic integrates the local reasoning of separation logic (where resources may always be split using the $*$ connective), with the abstract treatment of data structures advocated by abstract predicates. We proceed with an overview of structural separation logic and use it to specify the operations of the list module studied here.

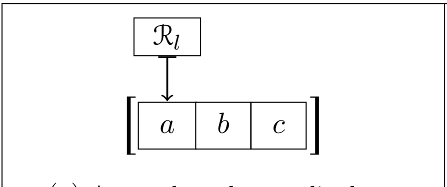
2.1.3. Structural Separation Logic: Informal Development

Structural separation logic [60], henceforth SSL, is a general program logic for the abstract specification of libraries that manipulate structured data, and local reasoning about their client programs. To achieve this, SSL combines the local reasoning of separation logic [40, 49] with the abstract reasoning of context logic [7, 6]. Here, we give an intuitive account of SSL and use it to specify the operations of the list module studied in §2.1.2. We give the technical details of SSL and its general theory in §3. We then use SSL to specify the operations of a simple tree library, as well as the Document Object Model (DOM) library [1] and to reason about their client programs.

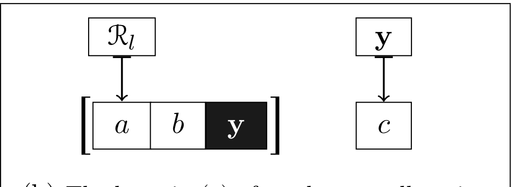
Consider the list operation `x.remove(n)`. Informally, this operation removes the first occurrence of the element identified by the value of variable `n`, leaving the rest of the list unchanged. To formalise this English description, we write SSL assertions that describe *abstract heaps*. Abstract heaps allow us to avoid considering the implementation details of the list structure. They are similar to the flat heaps of separation logic in that they consist of cells that map addresses to values. However, rather than storing simple values such as integers, they may store rich and complex structures.

For instance, Fig. 2.1a illustrates an *abstract list heap*, comprising a single cell at $\mathcal{R}_l$ with the complete abstract list $[a, b, c]$ as its value. This heap is abstract in that it hides the details of how the list at $\mathcal{R}_l$ is concretely represented in a machine.

Intuitively, when `n`$=b$ then the footprint of the `x.remove(n)` operation comprises the list fragment $[a, b]$ and not the list tail given by the $[c]$ frag-



(a) A complete abstract list heap



(b) The heap in (a) after abstract allocation

Figure 2.1.: Abstract list heaps

ment. Abstract heaps allow for their data to be split by imposing additional instrumentation using *abstract addresses*. This is reflected in the transition from Fig. 2.1a to Fig. 2.1b. The heap in Fig. 2.1a contains the *complete* list $[a, b, c]$ at $\mathcal{R}_l$. We then split this complete list using *abstract allocation* to obtain the abstract heap in Fig. 2.1b with the list fragment $[c]$ at a fresh, fictional abstract address $\mathbf{y}$, and an incomplete list at $\mathcal{R}_l$ with a *context hole* $\mathbf{y}$, indicating the position to which the list fragment will return. The list fragment $[a, b]{+\!\!+}\mathbf{y}$ now matches the intuitive footprint of `x.remove(n)` with the tail $[c]$ split off and excluded from the footprint. Once the `x.remove(n)` operation is completed and the necessary updates have been carried out, the heap can be joined back together using *abstract deallocation*, as in the transition from Fig. 2.1b to 2.1a.

To specify the behaviour of list operations, we write *assertions* that describe abstract heaps. For instance, let $\textsc{n}$ denote a logical variable with value b, $\textsc{l}$ denote a logical variable with value $[a]$, α denote a logical variable with value $\mathbf{y}$, and $\textsc{l}'$ denote a logical variable with value $[c]$. To describe the list cell at $\mathcal{R}_l$ in Fig. 2.1b, we write $\mathcal{R}_l \mapsto \textsc{l}{+\!\!+}[\textsc{n}]{+\!\!+}\alpha$. This assertion describes an abstract heap with a single list cell at address $\mathcal{R}_l$, containing the list fragment $\textsc{l}{+\!\!+}[\textsc{n}]{+\!\!+}\alpha$. Similarly, to describe the list cell at $\mathbf{y}$ in Fig. 2.1b, we write $\alpha \mapsto \textsc{l}'$. This assertion describes an abstract heap with a single list cell at abstract address α, containing the list fragment $\textsc{l}'$.

We now have all the ingredients to describe the behaviour of list operations (specified using the list and lfrag abstract predicates in §2.1.2) using SSL assertions instead. Observe that there is close correspondence between the abstract predicates list and lfrag, and the SSL assertions describing abstract list heaps. More concretely, given any list data $\textsc{l}$ and abstract address α, the $\mathsf{list}(\mathcal{R}_l, \textsc{l})$ predicate corresponds to the SSL assertion $\mathcal{R}_l \mapsto \textsc{l}$, and the $\mathsf{lfrag}(\alpha, \textsc{l})$ predicate corresponds to the SSL assertion $\alpha \mapsto \textsc{l}$. The list splitting axioms of (2.3)-(2.4) correspond to abstract allocation and deallocation on abstract list heaps (e.g. the transition from Fig. 2.1a to 2.1b and back). That is,

$$\forall \mathrm{L}_1, \mathrm{L}_2, \mathrm{L}_3, \mathrm{A}. \\ \mathrm{A} \mapsto \mathrm{L}_1 +\!\!+ \mathrm{L}_2 +\!\!+ \mathrm{L}_3 \iff \exists \alpha.\ \mathrm{A} \mapsto \mathrm{L}_1 +\!\!+ \alpha +\!\!+ \mathrm{L}_3 \ * \ \alpha \mapsto \mathrm{L}_2 \tag{2.5}$$

where A ranges over the heap address $\mathcal{R}_l$ and abstract addresses such as α. However, while we must explicitly axiomatise (2.3)-(2.4) for abstract list predicates, the property in (2.5) is derivable within the logic of SSL and need not be axiomatised.

We can now specify the behaviour of `x.remove(n)` in a similar way to the lfrag specification (p. 46), using SSL assertions instead:

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{x}\!:\!\mathcal{R}_l, \mathtt{n}\!:\!\mathrm{N}) \\ * \, \mathcal{R}_l \mapsto \mathrm{L} +\!\!+ [\mathrm{N}] +\!\!+ \alpha \\ * \, \mathrm{N} \dot{\notin} \mathrm{L} * \mathsf{complete}(\mathrm{L}) \end{array}\right\} \ \mathtt{x.remove(n)} \ \left\{\begin{array}{l} \mathsf{vars}(\mathtt{x}\!:\!\mathcal{R}_l, \mathtt{n}\!:\!\mathrm{N}) \\ * \, \mathcal{R}_l \mapsto \mathrm{L} +\!\!+ \alpha \end{array}\right\}$$

As before, the precondition comprises four assertions, with the $\mathsf{vars}(\mathtt{x}\!:\!\mathcal{R}_l, \mathtt{n}\!:\!\mathrm{N})$ assertion describing the values associated with program variables. The second assertion, $\mathcal{R}_l \mapsto \mathrm{L} +\!\!+ [\mathrm{N}] +\!\!+ \alpha$, describes an abstract list heap comprising a single cell at address $\mathcal{R}_l$, holding the list fragment $\mathrm{L} +\!\!+ [\mathrm{N}] +\!\!+ \alpha$. The last two assertions, $\mathrm{N} \dot{\notin} \mathrm{L} * \mathsf{complete}(\mathrm{L})$, state that the list fragment given by L is complete and does not contain N. Analogously, the postcondition states that the result of removing N is the list fragment $\mathrm{L} +\!\!+ \alpha$ at address $\mathcal{R}_l$. Note that the context hole α must be preserved in order to join this list fragment with the list that was split from it, using abstract deallocation. Were this not the case, the resulting heap would be *malformed* in that one could not join up the data at abstract address α with its associated context hole α, as it would have been deleted by `remove`. We can use SSL to specify the remaining list operations as given in Fig. 2.2.

We have given an intuitive account of SSL, a program logic for abstract specification and local reasoning about libraries that manipulate structured data. SSL consolidates the ideas behind context logic [7, 6] and segment logic [59, 25], refining them into a logic with a simple heap model based on that of separation logic. The heap model of SSL allows it to be integrated

$$\left\{\mathsf{vars}(\mathtt{x}{:}\mathcal{R}_l, \mathtt{n}{:}\mathrm{N}) * \mathcal{R}_l \mapsto \alpha\right\} \quad \mathtt{x.add(n)} \quad \left\{\mathsf{vars}(\mathtt{x}{:}\mathcal{R}_l, \mathtt{n}{:}\mathrm{N}) * \mathcal{R}_l \mapsto \alpha +\!\!+ [\mathrm{N}]\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}{:}\mathcal{R}_l, \mathtt{n}{:}\mathrm{N}) \\ * \mathcal{R}_l \mapsto \mathrm{L} +\!\!+ [\mathrm{N}] +\!\!+ \alpha \\ * \mathrm{N} \dot{\notin} \mathrm{L} * \mathsf{complete}(\mathrm{L})\end{array}\right\} \quad \mathtt{x.remove(n)} \quad \left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}{:}\mathcal{R}_l, \mathtt{n}{:}\mathrm{N}) \\ * \mathcal{R}_l \mapsto \mathrm{L} +\!\!+ \alpha\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}{:}\mathcal{R}_l, \mathtt{i}{:}\mathrm{I}, \mathtt{r}{:}\mathrm{R}) \\ * \mathcal{R}_l \mapsto \mathrm{L} +\!\!+ [\mathrm{N}] +\!\!+ \alpha \\ * \mathrm{I} \doteq |\mathrm{L}| * \mathsf{complete}(\mathrm{L})\end{array}\right\} \quad \mathtt{r := x.item(i)} \quad \left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}{:}\mathcal{R}_l, \mathtt{i}{:}\mathrm{I}, \mathtt{r}{:}\mathrm{N}) \\ * \mathcal{R}_l \mapsto \mathrm{L} +\!\!+ [\mathrm{N}] +\!\!+ \alpha\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}{:}\mathcal{R}_l, \mathtt{i}{:}\mathrm{I}, \mathtt{r}{:}\mathrm{R}) \\ * \mathcal{R}_l \mapsto \mathrm{L} \\ * \mathrm{I} \dot{\geq} |\mathrm{L}| * \mathsf{complete}(\mathrm{L})\end{array}\right\} \quad \mathtt{r := x.item(i)} \quad \left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}{:}\mathcal{R}_l, \mathtt{i}{:}\mathrm{I}, \mathtt{r}{:}\mathtt{null}) \\ * \mathcal{R}_l \mapsto \mathrm{L}\end{array}\right\}$$

Figure 2.2.: The axiomatic specification of the list module using SSL

with other program logics based on separation logic, such as the original work of O'Hearn [49], Parkinson's separation logic for Java [42], and the work on JavaScript by Smith and others [21]. SSL allows us to write specifications that are both abstract and local, without additional axiomatisation or complex machinery. As such, in the remainder of this thesis we choose SSL as the abstraction mechanism, and use it to specify the behaviour of the library operations we study, as well as to reason about their client programs.

2.2. Client Reasoning

To reason about our client programs, we use the WLogic program logic presented by Wright in his thesis [60]. WLogic is a program logic for a simple while language comprising the standard constructs of sequential composition (`;`), `skip`, conditionals, loops and parallel composition (`||`). The reasoning principles and proof rules of WLogic are standard and are omitted here. We describe the technical details of WLogic in §3. We combine the list module specification given in Fig. 2.2 with WLogic to reason about the `x.size()` client program given in Fig. 2.3.

2.2.1. The `x.size()` Client Program

The `r := x.size()` client program in Fig. 2.3 computes the length of the list at address `x` and returns it in `r`. To do this, the program proceeds by traversing the list from the head (the first element) to the end, keeping a

```
r := x.size() ≜ var curr, size in {
                  size = 0;
                  curr := x.item(size);
                  while(curr != null){
                    size++;
                    curr := x.item(size);
                  }
                  r = size;
                }
```

Figure 2.3.: The `x.size()` client program of the list module

count along the way. As such, the footprint of the `x.size()` program spans the entire list, and the precondition must thus include the complete list at `x`. We can specify the behaviour of the `x.size()` program as follows.

$$\left\{\mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \mathrm{R}) * \mathcal{R}_l \mapsto \mathrm{L} * \mathsf{complete}(\mathrm{L})\right\}$$
$$\mathtt{r := x.size()}$$
$$\left\{\exists \mathrm{S}.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \mathrm{S}) * \mathcal{R}_l \mapsto \mathrm{L} * \mathrm{S} \doteq |\mathrm{L}|\right\}$$

A proof sketch of `x.size()` is given in Figs. 2.4 and 2.5. At each proof point, we have highlighted the effect of the preceding command, where applicable. For instance, after the assignment of line 2, the value of variable `size` is updated to 0, whereas the `while` statement of line 4 has no effect.

Concluding Remarks We have given an informal account of SSL, a program logic for abstract specification, and have demonstrated how it may be used for local reasoning about libraries that manipulate structured data. We present the general theory of SSL in §3 and demonstrate how it may be integrated into an SL-based program logic for client reasoning. In §4, we use SSL to specify a simple tree library $\mathbb{T}$, and integrate it into WLogic, an SL-based program logic for a simple while language, to reason about the client programs of $\mathbb{T}$ in WLogic. Later in §5, we specify the behaviour of a fragment of the Document Object Library (DOM) [1]. We then integrate our DOM specification into JSLogic, an SL-based program logic for JavaScript [21], to reason about the DOM client programs written in JavaScript. As mentioned earlier, SSL consolidates the ideas behind context logic [7, 6], refining them

```
$\{\mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \mathrm{R}) * \mathcal{R}_l \mapsto \mathrm{L} * \mathsf{complete}(\mathrm{L})\}$
  1. var curr, size in {
     $\{\mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \mathrm{R}, \mathtt{curr}{:}-, \mathtt{size}{:}-) * \mathcal{R}_l \mapsto \mathrm{L} * \mathsf{complete}(\mathrm{L})\}$
  2.   size = 0;
     $\{\mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \mathrm{R}, \mathtt{curr} : -, \mathtt{size} : 0) * \mathcal{R}_l \mapsto \mathrm{L} * \mathsf{complete}(\mathrm{L})\}$
     // Unwrap the definition of complete
     $\left\{\begin{array}{l} \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \mathrm{R}, \mathtt{curr} : -, \mathtt{size} : 0) \\ * \Big((\mathrm{L} \doteq [\,] * \mathcal{R}_l \mapsto [\,]) \vee (\exists \mathrm{H}, \mathrm{L}'.\ \mathrm{L} \doteq [\mathrm{H}] \mathbin{+\!\!+} \mathrm{L}' * \mathsf{complete}(\mathrm{L}') * \mathcal{R}_l \mapsto [\mathrm{H}] \mathbin{+\!\!+} \mathrm{L}')\Big) \end{array}\right\}$
     // Abstract allocation at $\alpha$ by (2.5)
     $\left\{\begin{array}{l} \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \mathrm{R}, \mathtt{curr} :, \mathtt{size} : 0) * \Big((\mathrm{L} \doteq [\,] * \mathcal{R}_l \mapsto [\,]) \\ \vee (\exists \mathrm{H}, \mathrm{L}', \alpha.\ \mathrm{L} \doteq [\mathrm{H}] \mathbin{+\!\!+} \mathrm{L}' * \mathsf{complete}(\mathrm{L}') * \mathcal{R}_l \mapsto [\mathrm{H}] \mathbin{+\!\!+} \alpha * \alpha \mapsto \mathrm{L}')\Big) \end{array}\right\}$
  3.   curr := x.item(size);
     $\left\{\begin{array}{l} \exists \mathrm{C}.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \mathrm{R}, \mathtt{curr} : \mathrm{C}, \mathtt{size} : 0) * \Big((\mathrm{L} \doteq [\,] * \mathrm{C} \doteq \mathtt{null} * \mathcal{R}_l \mapsto [\,]) \\ \vee (\exists \mathrm{H}, \mathrm{L}'.\ \mathrm{L} \doteq [\mathrm{H}] \mathbin{+\!\!+} \mathrm{L}' * \mathsf{complete}(\mathrm{L}') * \mathrm{C} \doteq \mathrm{H} * \mathcal{R}_l \mapsto [\mathrm{H}] \mathbin{+\!\!+} \alpha * \alpha \mapsto \mathrm{L}')\Big) \end{array}\right\}$
     // Abstract deallocation at $\alpha$ by (2.5)
     $\left\{\begin{array}{l} \exists \mathrm{C}.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \mathrm{R}, \mathtt{curr} : \mathrm{C}, \mathtt{size} : 0) * \Big((\mathrm{L} \doteq [\,] * \mathrm{C} \doteq \mathtt{null} * \mathcal{R}_l \mapsto [\,]) \\ \vee (\exists \mathrm{H}, \mathrm{L}'.\ \mathrm{L} \doteq [\mathrm{H}] \mathbin{+\!\!+} \mathrm{L}' * \mathsf{complete}(\mathrm{L}') * \mathrm{C} \doteq \mathrm{H} * \mathcal{R}_l \mapsto [\mathrm{H}] \mathbin{+\!\!+} \mathrm{L}')\Big) \end{array}\right\}$
     // Weaken each disjunct to arrive at the loop invariant
     $\left\{\begin{array}{l} \exists \mathrm{C}, \mathrm{S}, \mathrm{L}_1, \mathrm{L}_2.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \mathrm{R}, \mathtt{curr} : \mathrm{C}, \mathtt{size} : \mathrm{S}) \\ * \Big((\mathrm{C} \doteq \mathtt{null} * \mathcal{R}_l \mapsto \mathrm{L} * \mathrm{S} \doteq |\mathrm{L}| * \mathsf{complete}(\mathrm{L})) \vee (\mathrm{L} \doteq \mathrm{L}_1 \mathbin{+\!\!+} [\mathrm{C}] \mathbin{+\!\!+} \mathrm{L}_2 * \mathrm{S} \doteq |\mathrm{L}_1| \\ \qquad * \mathsf{complete}(\mathrm{L}_1 \mathbin{+\!\!+} [\mathrm{C}]) * \mathsf{complete}(\mathrm{L}_2) * \mathcal{R}_l \mapsto \mathrm{L}_1 \mathbin{+\!\!+} [\mathrm{C}] \mathbin{+\!\!+} \mathrm{L}_2)\Big) \end{array}\right\}$
```

Figure 2.4.: A proof sketch of `x.size()` (continued in Fig. 2.5)

into a logic with a simple heap model based on that of separation logic. As we demonstrate later in §5, SSL improves on context logic in several ways. Most notably, SSL allows for compositional client reasoning, significantly reducing the number of specifications needed for describing the behaviour of programs. To illustrate this, we show that specifying a simple DOM client program (comprising 3 lines of code) in context logic requires at least *six* separate Hoare triples, whereas the same program can be specified in SSL with a single triple (see p. 139).

```
    // Repeat the last line from Fig. 2.4
    $\left\{\begin{array}{l}\exists \text{C}, \text{S}, \text{L}_1, \text{L}_2.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{R}, \mathtt{curr} : \text{C}, \mathtt{size} : \text{S}) \\ * \Big((\text{C} \doteq \mathtt{null} * \mathcal{R}_l \mapsto \text{L} * \text{S} \doteq |\text{L}| * \mathsf{complete}(\text{L})) \vee (\text{L} \doteq \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \text{L}_2 * \text{S} \doteq |\text{L}_1| \\ \qquad * \mathsf{complete}(\text{L}_1 +\!\!+ [\text{C}]) * \mathsf{complete}(\text{L}_2) * \mathcal{R}_l \mapsto \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \text{L}_2)\Big)\end{array}\right\}$
4.    while(curr != null) {
    $\left\{\begin{array}{l}\exists \text{C}, \text{S}, \text{L}_1, \text{L}_2.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{R}, \mathtt{curr} : \text{C}, \mathtt{size} : \text{S}) * \text{L} \doteq \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \text{L}_2 * \text{S} \doteq |\text{L}_1| \\ * \mathsf{complete}(\text{L}_1 +\!\!+ [\text{C}]) * \mathsf{complete}(\text{L}_2) * \mathcal{R}_l \mapsto \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \text{L}_2\end{array}\right\}$
5.      size++;
    $\left\{\begin{array}{l}\exists \text{S}, \text{L}_1, \text{L}_2.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{R}, \mathtt{curr} : -, \mathtt{size} : \text{S}) * \text{L} \doteq \text{L}_1 +\!\!+ \text{L}_2 * \text{S} \doteq |\text{L}_1| \\ * \mathsf{complete}(\text{L}_1) * \mathsf{complete}(\text{L}_2) * \mathcal{R}_l \mapsto \text{L}_1 +\!\!+ \text{L}_2\end{array}\right\}$
    // Unwrap the definition of $\mathsf{complete}(\text{L}_2)$
    $\left\{\begin{array}{l}\exists \text{S}, \text{L}_1, \text{L}_2.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{R}, \mathtt{curr} : \text{-}, \mathtt{size} : \text{S}) \\ * \Big((\mathcal{R}_l \mapsto \text{L} * \text{S} \doteq |\text{L}| * \mathsf{complete}(\text{L})) \vee (\exists \text{C}.\ \text{L} \doteq \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \text{L}_2 * \text{S} \doteq |\text{L}_1| \\ \qquad * \mathsf{complete}(\text{L}_1 +\!\!+ [\text{C}]) * \mathsf{complete}(\text{L}_2) * \mathcal{R}_l \mapsto \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \text{L}_2)\Big)\end{array}\right\}$
    // Abstract allocation at $\alpha$ by (2.5)
    $\left\{\begin{array}{l}\exists \text{S}, \text{L}_1, \text{L}_2, \alpha.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{R}, \mathtt{curr} : \text{-}, \mathtt{size} : \text{S}) \\ * \Big((\mathcal{R}_l \mapsto \text{L} * \text{S} \doteq |\text{L}| * \mathsf{complete}(\text{L})) \vee (\exists \text{C}.\ \text{L} \doteq \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \text{L}_2 * \text{S} \doteq |\text{L}_1| \\ \qquad * \mathsf{complete}(\text{L}_1 +\!\!+ [\text{C}]) * \mathsf{complete}(\text{L}_2) * \mathcal{R}_l \mapsto \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \alpha * \alpha \mapsto \text{L}_2)\Big)\end{array}\right\}$
6.      curr := x.item(size)
    $\left\{\begin{array}{l}\exists \text{S}, \text{L}_1, \text{L}_2, \alpha, \text{C}.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{R}, \mathtt{curr} : \text{C}, \mathtt{size} : \text{S}) \\ * \Big((\text{C} \doteq \mathtt{null} * \mathcal{R}_l \mapsto \text{L} * \text{S} \doteq |\text{L}| * \mathsf{complete}(\text{L})) \vee (\text{L} \doteq \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \text{L}_2 * \text{S} \doteq |\text{L}_1| \\ \qquad * \mathsf{complete}(\text{L}_1 +\!\!+ [\text{C}]) * \mathsf{complete}(\text{L}_2) * \mathcal{R}_l \mapsto \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \alpha * \alpha \mapsto \text{L}_2)\Big)\end{array}\right\}$
    // Abstract deallocation at $\alpha$ by (2.5)
    $\left\{\begin{array}{l}\exists \text{S}, \text{L}_1, \text{L}_2, \text{C}.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{R}, \mathtt{curr} : \text{C}, \mathtt{size} : \text{S}) \\ * \Big((\text{C} \doteq \mathtt{null} * \mathcal{R}_l \mapsto \text{L} * \text{S} \doteq |\text{L}| * \mathsf{complete}(\text{L})) \vee (\text{L} \doteq \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \text{L}_2 * \text{S} \doteq |\text{L}_1| \\ \qquad * \mathsf{complete}(\text{L}_1 +\!\!+ [\text{C}]) * \mathsf{complete}(\text{L}_2) * \mathcal{R}_l \mapsto \text{L}_1 +\!\!+ [\text{C}] +\!\!+ \text{L}_2)\Big)\end{array}\right\}$
7.    }
    // Exit from the loop; only the first disjunct of the invariant applies.
    $\{\exists \text{S}.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{R}, \mathtt{curr} : \mathtt{null}, \mathtt{size} : \text{S}) * \mathcal{R}_l \mapsto \text{L} * \text{S} \doteq |\text{L}| * \mathsf{complete}(\text{L})\}$
    $\{\exists \text{S}.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{R}, \mathtt{curr} : \mathtt{null}, \mathtt{size} : \text{S}) * \mathcal{R}_l \mapsto \text{L} * \text{S} \doteq |\text{L}|\}$
8.    r = size;
    $\{\exists \text{S}.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{S}, \mathtt{curr} : \mathtt{null}, \mathtt{size} : \text{S}) * \mathcal{R}_l \mapsto \text{L} * \text{S} \doteq |\text{L}|\}$
9.  }
$\{\exists \text{S}.\ \mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{r} : \text{S}) * \mathcal{R}_l \mapsto \text{L} * \text{S} \doteq |\text{L}|\}$
```

Figure 2.5.: A proof sketch of `x.size()` (continued from Fig. 2.4)

3. Structural Separation Logic (SSL)

Structural separation logic (SSL) introduced by Wright in [60] is a general program logic for abstractly specifying the behaviour of libraries of structured data, and for reasoning locally about their client programs. We present the general theory of SSL and its various ingredients necessary to specify a library abstractly, as presented in [60]. The general theory of SSL is parametric and may be instantiated accordingly to specify the behaviour of any library for structured data. We present the general model of SSL and its assertion language in §3.1.

Given a library $\mathbb{A}$ for abstract data (e.g. a list library such as that in §2.1.2), in §3.2 we demonstrate how to use the SSL specification of $\mathbb{A}$ to reason about its client programs written in an arbitrary programming language with an SL-based program logic. More concretely, given a programming language PL, with an SL-based program logic PLogic, we demonstrate how to extend PLogic to $\text{PLogic}_{\mathbb{A}}$ to specify the behaviour of $\mathbb{A}$ operations, and to reason about the client programs of $\mathbb{A}$ written in PL, provided that PLogic meets certain assumptions.

In order to extend PLogic to $\text{PLogic}_{\mathbb{A}}$, we integrate the various components of SSL (e.g. states, assertions, etc.) with those of PLogic. In particular, we must integrate the assertions of the SSL and PLogic in a way that allows for combining the information about their respective underlying sates. For instance, in the specification of the `x.remove(n)` list operation in Fig. 2.2, the $\mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{n} : \text{N})$ assertion describes a variable store in the client programming language where *program variables* `x` and `n` respectively hold the value $\mathcal{R}_l$, and the value denoted by the *logical variable* N. In other words, program variables provide a means of interaction between the client programming language PL and the library $\mathbb{A}$ operations. Similarly, logical variables provide a means of information exchange between the client program logic PLogic and the library $\mathbb{A}$ specification in SSL. As such, to integrate PLogic and SSL we must adopt a unified notion of program and logical variables. More concretely, since we aim to reason about library $\mathbb{A}$ client programs written in PL, the set of program variables are those of PL. Analogously, since we aim to incorporate the SSL specification of library $\mathbb{A}$ into the program logic of PLogic, the set of logical variables are those of PLogic. To this end, we define a unified set of program variables, PVar, as well as a unified set of logical variables, LVar.

Unlike program variables, the choice of *program values* may differ between the PL language and library $\mathbb{A}$. Program values describe the set of values that can be represented in the machine and may be observed via program variables. For instance, in the above variable store assertion $\mathsf{vars}(\mathtt{x} : \mathcal{R}_l, \mathtt{n} : \textsc{n})$, the value associated with variable $\mathtt{x}$, namely $\mathcal{R}_l$, denotes a program value of the list library describing the location of the list in the underlying abstract list heap. Moreover, the $\mathcal{R}_l$ is a list library value and may not necessarily correspond to a program value in PL. As such, the programming language PL and library $\mathbb{A}$ may each define their own sets of program values.

Similarly, the choice of *logical values* in the program logic PLogic may differ from those of SSL. Logical values describe the values that may be associated with logical variables and often include program values. The set of logical values may additionally include instrumented values used purely for reasoning (e.g. an abstract address $\mathbf{x}$ denoting a fictional address in the abstract heap). Since PL program values may not necessarily correspond to those of library $\mathbb{A}$, the SSL specification for $\mathbb{A}$ and PLogic may each define their own sets of logical value.

To track the values of logical variables in assertions, we then define a set of *parametric* logical environments, $\textsc{LEnv}\langle \mathrm{V} \rangle$, that map logical variables onto logical values in V. As we show later, when the set of logical values for PLogic is given by PLLVal, and the set of SSL logical values for library $\mathbb{A}$ is given by $\textsc{LVal}_{\mathbb{A}}$, we interpret $\textsc{PLogic}_{\mathbb{A}}$ assertions with respect to a logical environment in $\textsc{LEnv}\langle \textsc{PLLVal} \cup \textsc{LVal}_{\mathbb{A}} \rangle$.

The W-*coercion* of a logical environment Γ, written $\Gamma\!\downarrow_{\mathrm{W}}$, limits the domain of Γ to those variables whose values inhabit W. We use this to enforce the correct interpretation of assertions when the range of Γ is bigger than permitted. For instance, when interpreting $\textsc{PLogic}_{\mathbb{A}}$ assertions (which include both SSL and PLogic assertions) with respect to $\Gamma \in \textsc{LEnv}\langle \textsc{PLLVal} \cup \textsc{LVal}_{\mathbb{A}} \rangle$, SSL assertions are interpreted over $\Gamma\!\downarrow_{\textsc{LVal}_{\mathbb{A}}}$.

Definition 1 (Program variables)**.** The countably infinite set of *program variables* is $\mathtt{x} \in \textsc{PVar}$.

Definition 2 (Logical variables)**.** The countably infinite set of *logical variables* is $\mathrm{x} \in \textsc{LVar}$.

Definition 3 (Logical environments)**.** Given the set of logical variables $\textsc{LVar}$ (Def. 2) and a set of logical values V, the set of *parametric logical environments on* V is $\Gamma \in \textsc{LEnv}\langle \mathrm{V} \rangle \triangleq \textsc{LVar} \overset{\text{fin}}{\rightharpoonup} \mathrm{V}$.

Given two sets of logical values V and W and a logical environment $\Gamma \in \textsc{LEnv}\langle \mathrm{V} \rangle$, the W-*coercion of* Γ, written $\Gamma{\downarrow_{\mathrm{W}}}$, is defined as follows:

$$\Gamma{\downarrow_{\mathrm{W}}}(\mathrm{x}) \triangleq \begin{cases} \Gamma(\mathrm{x}) & \text{if } \Gamma(\mathrm{x}) \in \mathrm{W} \\ \text{undefined} & \text{otherwise} \end{cases}$$

We proceed with the general theory of SSL and its assertion language.

3.1. SSL Model and Assertions

The general theory of SSL [60] is parametric and may be instantiated to specify the behaviour of a particular library of structured data. We present the SSL theory for a general library $\mathbb{A}$, and delineate the parameters of SSL enclosed in solid boxes labelled "SSL Parameter". To provide a clearer account of the SSL theory, we appeal to the list library $\mathbb{L}$ presented informally in §2.1 and follow each SSL parameter with the instantiation of the corresponding parameter for $\mathbb{L}$ enclosed in dashed boxes labelled "SSL $\mathbb{L}$ Instance (Parameter X)", where X is the reference to the corresponding parameter. We proceed with the general SSL model and its assertion language.

SSL Model

We begin by modelling *abstract program heaps* such as the list heap in Fig. 2.1a. Abstract program heaps are mappings from *root addresses* (e.g. $\mathcal{R}_l$) to *complete* program data with no context holes (e.g. $[a, b, c]$). Root addresses are library specific and may vary from one library to another. We thus parameterise the library-specific root addresses $\mathcal{R} \in \textsc{RAdd}_{\mathbb{A}}$.

SSL Parameter

Parameter 1 (Root addresses)**.** Assume a countable set of *root addresses*, $\mathcal{R} \in \textsc{RAdd}_{\mathbb{A}}$.

For list heaps, we define a designated *root address* $\mathcal{R}_l$, denoting the location in the list heap where the list data is stored.

SSL $\mathbb{L}$ Instance (Parameter 1)

Definition 4 (List root addresses)**.** The set of *root addresses for lists* is $\textsc{RAdd}_{\mathbb{L}} \triangleq \{\mathcal{R}_l\}$.

Recall that abstract program heaps are mappings from root addresses to program data. Program data is library specific and describes complete data with no context holes. It provides a high-level representation of the underlying data structure that is agnostic to the implementation details and how the data structure may be represented in the heap. For instance, program data for the list module in §2.1 comprises complete abstract lists such as $[a, b, c]$ in Fig. 2.1a.

SSL Parameter

Parameter 2 (Program data)**.** Assume a set of *program data*, $\mathrm{d} \in \textsc{PData}_{\mathbb{A}}$.

For the list library $\mathbb{L}$, program data describes an ordered collection of values defined inductively and includes empty lists, singleton lists, and composite lists. To model lists, we assume a countably infinite set of *list elements*, $n \in \textsc{LElem}$, delimiting the space from which list elements are drawn.

SSL $\mathbb{L}$ Instance (Parameter 2)

Definition 5 (List program data)**.** Let $\textsc{LElem}$ denote a countably infinite set of *list elements*. The set of *program data for lists*, $\mathrm{l} \in \textsc{PData}_{\mathbb{L}}$, is defined by the following grammar where $n \in \textsc{LElem}$:

$$\mathrm{l} ::= [\,] \mid [n] \mid \mathrm{l}_1 {+\!\!+} \mathrm{l}_2$$

The $+\!\!+$ operation is associative with identity $[\,]$ and all list data are equal up to the associativity of $+\!\!+$.[1]

We write $[n_1, \ldots, n_k]$ as a shorthand for $[n_1]+\!\!+\ldots+\!\!+[n_k]$.

As previously discussed, the set of program values may vary from one library to another. We thus assume a set of *library-specific program values* that include root addresses. Library-specific program values denote the set of program values that may be observed by the clients of the library (via program variables). For instance, for the list library studied in §2.1.3, the library-specific program values comprise the list root address $\mathcal{R}_l$, as well as the list elements in $\textsc{LElem}$ (Def. 5). This is evident in the list axioms of Fig. 2.2 where the program variables may contain either the root address $\mathcal{R}_l$ or logical values corresponding to elements in the underlying list.

SSL Parameter

Parameter 3 (Library program values)**.** Given the set of root addresses $\textsc{RAdd}_{\mathbb{A}}$ (Par. 1), assume a set of *library program values*, $v \in \textsc{PVal}_{\mathbb{A}}$, such that $\textsc{RAdd}_{\mathbb{A}} \subseteq \textsc{PVal}_{\mathbb{A}}$.

For the list library $\mathbb{L}$, the program values include the list root address $\mathcal{R}_l$ and the elements in $\textsc{LElem}$.

SSL $\mathbb{L}$ Instance (Parameter 3)

Definition 6 (List program values)**.** Given the set of list root addresses $\textsc{RAdd}_{\mathbb{L}}$ (Def. 4) and the set of list elements $\textsc{LElem}$ (Def. 5), the set of *program values for lists* is $v \in \textsc{PVal}_{\mathbb{L}} \triangleq \textsc{RAdd}_{\mathbb{L}} \uplus \textsc{LElem}$.

We can now model abstract program heaps as partial functions mapping root addresses onto abstract program data.

Definition 7 (Abstract program heaps)**.** Given the set of root address $\textsc{RAdd}_{\mathbb{A}}$ (Par. 1) and the set of program data $\textsc{PData}_{\mathbb{A}}$ (Par. 2), the set of

[1]It is straightforward to formalise this associativity property.

abstract program heaps for library $\mathbb{A}$ is:

$$\mathrm{h} \in \textsc{PHeap}_{\mathbb{A}} \triangleq \textsc{RAdd}_{\mathbb{A}} \xrightarrow{\text{fin}} \textsc{PData}_{\mathbb{A}}$$

We now proceed to model abstract *logical heaps* such as the list heap in Fig. 2.1b. Abstract logical heaps are mappings from *addresses* to *logical data*. Addresses comprise abstract addresses (e.g. $\mathbf{y}$ in Fig. 2.1b), as well as library-specific root addresses (e.g. $\mathcal{R}_l$ for lists). To model this, we assume a countably infinite set of *abstract addresses*, $\mathbf{x} \in \textsc{AAdd}$, such that the sets of abstract addresses and root addresses are disjoint.

Definition 8 (Abstract addresses)**.** Given the set of library root addresses $\textsc{RAdd}_{\mathbb{A}}$ (Par. 1), the countably infinite set of *abstract addresses* is $\mathbf{x} \in \textsc{AAdd}$, such that $\textsc{AAdd} \cap \textsc{RAdd}_{\mathbb{A}} = \emptyset$.

Definition 9 (Addresses)**.** Given the set of library root addresses $\textsc{RAdd}_{\mathbb{A}}$ (Par. 1) and the set of abstract addresses $\textsc{AAdd}$ (Def. 8), the set of *addresses for library* $\mathbb{A}$ is $a \in \textsc{Add}_{\mathbb{A}} \triangleq \textsc{AAdd} \uplus \textsc{RAdd}_{\mathbb{A}}$.

As mentioned earlier, abstract logical heaps are mappings from addresses to *logical data*. As with program data, logical data is library specific and provides a high-level representation of the underlying data structure that is agnostic to how the data structure may be represented in the heap. Logical data contains the (complete) program data (Par. 2) and *may* additionally contain incomplete data with context holes (e.g. $\mathbf{x}$). In other words, since the definition of logical data is library specific, we may choose to limit logical data to include program data only. However, as we demonstrated in §2.1, admitting incomplete data allows us to specify library operations more locally. Notationally, as with context holes, we use the **boldface** font and write $\mathbf{d}$, $\mathbf{d}_1$ and so forth to range over logical data. This is to remind the reader that logical data may contain context holes. Given the set of logical data, there is an associated *address function* which returns the set of context holes present in the logical data.

SSL Parameter

Parameter 4 (Logical data)**.** Given the set of program data $\textsc{PData}_{\mathbb{A}}$ (Par. 2), assume a set of *logical data*, $\mathbf{d} \in \textsc{LData}_{\mathbb{A}}$, such that $\textsc{PData}_{\mathbb{A}} \subseteq \textsc{LData}_{\mathbb{A}}$.
Given the set of abstract addresses $\textsc{AAdd}$ (Def. 8), assume an *address function*, $\mathrm{addr}_{\mathbb{A}}(.) : \textsc{LData}_{\mathbb{A}} \to \mathcal{P}(\textsc{AAdd})$, returning the set of context holes present in the logical data.

When the type of logical data is clear from the context, we drop the subscript and write $\mathrm{addr}(.)$ for $\mathrm{addr}_{\mathbb{A}}(.)$.

For the list library $\mathbb{L}$, logical data includes the complete lists defined by list program data (Def. 5), as well as incomplete lists with context holes.

SSL $\mathbb{L}$ Instance (Parameter 4)

Definition 10 (List logical data)**.** The set of *logical list data*, $\mathbf{l} \in \textsc{LData}_{\mathbb{L}}$, is defined by the following grammar where $\mathbf{x} \in \textsc{AAdd}$ (Def. 8) and $n \in \textsc{LElem}$ (Def. 5):

$$\mathbf{l} ::= [\,] \mid \mathbf{x} \mid [n] \mid \mathbf{l}_1 {+\!\!+} \mathbf{l}_2$$

Logical list data does not contain repeated context holes; the ${+\!\!+}$ operation is associative with identity $[\,]$ and all logical list data are equal up to the associativity of ${+\!\!+}$.[2]
The *list address function*, $\mathrm{addr}_{\mathbb{L}}(.) : \textsc{LData}_{\mathbb{L}} \to \mathcal{P}(\textsc{AAdd})$, is defined inductively over the structure of logical list data as follows:

$$\mathrm{addr}_{\mathbb{L}}([\,]) \triangleq \emptyset \qquad \mathrm{addr}_{\mathbb{L}}(\mathbf{x}) \triangleq \{\mathbf{x}\} \qquad \mathrm{addr}_{\mathbb{L}}([n]) \triangleq \emptyset$$
$$\mathrm{addr}_{\mathbb{L}}(\mathbf{l}_1 {+\!\!+} \mathbf{l}_2) \triangleq \mathrm{addr}_{\mathbb{L}}(\mathbf{l}_1) \uplus \mathrm{addr}_{\mathbb{L}}(\mathbf{l}_2)$$

Recall that data splitting in SSL is achieved through abstract allocation (e.g. the transition from Fig. 2.1a to 2.1b). Conversely, data can be combined via abstract deallocation (e.g. the transition from Fig. 2.1b to 2.1a) where subdata at an abstract address is *compressed* into its counterpart context hole. To model this data compression, we appeal to *context application*

[2]It is straightforward to formalise these restrictions.

on logical data, written $\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2$, describing the collapsing of logical data $\mathbf{d}_2$ into the context hole $\mathbf{x}$ in $\mathbf{d}_1$. This is typically defined as the standard substitution of $\mathbf{x}$ in $\mathbf{d}_1$ with $\mathbf{d}_2$.

SSL Parameter

Parameter 5 (Application)**.** Given the set of abstract addresses $\text{AA}\textsc{dd}$ (Def. 8) and the set of logical data $\text{LD}\textsc{ata}_{\mathbb{A}}$ (Par. 4), assume a *context application* function, $\diamond : \text{LD}\textsc{ata}_{\mathbb{A}} \times \text{AA}\textsc{dd} \times \text{LD}\textsc{ata}_{\mathbb{A}} \rightharpoonup \text{LD}\textsc{ata}_{\mathbb{A}}$, such that the following properties hold for all $\mathbf{d}, \mathbf{d}_1, \mathbf{d}_2 \in \text{LD}\textsc{ata}_{\mathbb{A}}$ and $\mathbf{x}, \mathbf{y} \in \text{AA}\textsc{dd}$, where for clarity $\diamond(\mathbf{d}_1, \mathbf{x}, \mathbf{d}_2)$ is written as $\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2$:

1. if $\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2$ is defined then:
   a) *Containment*: $\mathbf{x} \in \text{addr}(\mathbf{d}_1)$
   b) *Non-overlap*: $\text{addr}(\mathbf{d}_1) \cap \text{addr}(\mathbf{d}_2) \subseteq \{\mathbf{x}\}$
   c) *Preservation*: $(\text{addr}(\mathbf{d}_1) \setminus \{\mathbf{x}\}) \cup \text{addr}(\mathbf{d}_2) = \text{addr}(\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2)$
2. *Identity*: $\mathbf{x} \diamond_{\mathbf{x}} \mathbf{d} = \mathbf{d}$.
3. *Arbitrary addresses*: if $\mathbf{x} \in \text{addr}(\mathbf{d})$ and either $\mathbf{y} \notin \text{addr}(\mathbf{d})$ or $\mathbf{y} = \mathbf{x}$, then $\mathbf{d} \diamond_{\mathbf{x}} \mathbf{y}$ is defined. Also, $\mathbf{d} \diamond_{\mathbf{x}} \mathbf{x} = \mathbf{d}$.
4. *Left-cancellativity*: if $\mathbf{d} \diamond_{\mathbf{x}} \mathbf{d}_1 = \mathbf{d} \diamond_{\mathbf{x}} \mathbf{d}_2$, then $\mathbf{d}_1 = \mathbf{d}_2$.
5. *Quasi-associativity*: if $\mathbf{y} \in \text{addr}(\mathbf{d}_2)$ and either $\mathbf{y} \notin \text{addr}(\mathbf{d}_1)$ or $\mathbf{x} = \mathbf{y}$, then $(\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2) \diamond_{\mathbf{y}} \mathbf{d} = \mathbf{d}_1 \diamond_{\mathbf{x}} (\mathbf{d}_2 \diamond_{\mathbf{y}} \mathbf{d})$.
6. *Quasi-commutativity*: if $\mathbf{x} \notin \text{addr}(\mathbf{d}_2)$ and $\mathbf{y} \notin \text{addr}(\mathbf{d}_1)$, then $(\mathbf{d} \diamond_{\mathbf{x}} \mathbf{d}_1) \diamond_{\mathbf{y}} \mathbf{d}_2 = (\mathbf{d} \diamond_{\mathbf{y}} \mathbf{d}_2) \diamond_{\mathbf{x}} \mathbf{d}_1$.

where undefined terms are considered equal.

For list data, we define context application $\mathbf{l}_1 \diamond_{\mathbf{x}} \mathbf{l}_2$ in the standard way: it is undefined when $\mathbf{x} \notin \text{addr}(\mathbf{l}_1)$; otherwise, it is defined as $\mathbf{l}_1[\mathbf{l}_2/\mathbf{x}]$, denoting the standard substitution of $\mathbf{l}_2$ for $\mathbf{x}$ in $\mathbf{l}_1$, provided that the substitution result is in $\text{LD}\textsc{ata}_{\mathbb{L}}$.

SSL $\mathbb{L}$ Instance (Parameter 5)

Definition 11 (List application)**.** Given the set of abstract addresses $\textsc{AAdd}$ (Def. 8) and the set of logical list data $\textsc{LData}_{\mathbb{L}}$ (Def. 10), the *list context application* function, $\diamond : \textsc{LData}_{\mathbb{L}} \times \textsc{AAdd} \times \textsc{LData}_{\mathbb{L}} \rightharpoonup \textsc{LData}_{\mathbb{L}}$, is defined inductively over the structure of logical list data as follows:

$$[\,] \diamond_{\mathbf{x}} \mathbf{l} \quad \text{undefined} \qquad\qquad [n] \diamond_{\mathbf{x}} \mathbf{l} \quad \text{undefined}$$

$$\mathbf{y} \diamond_{\mathbf{x}} \mathbf{l} \triangleq \begin{cases} \mathbf{l} & \text{if } \ \mathbf{x} = \mathbf{y} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\mathbf{l}_1 +\!\!+ \mathbf{l}_2) \diamond_{\mathbf{x}} \mathbf{l} \triangleq \begin{cases} \mathbf{l}' +\!\!+ \mathbf{l}_2 & \text{if } \ \mathbf{l}_1 \diamond_{\mathbf{x}} \mathbf{l} = \mathbf{l}' \ \text{ and } \ \mathbf{l}' +\!\!+ \mathbf{l}_2 \in \textsc{LData}_{\mathbb{L}} \\ \mathbf{l}_1 +\!\!+ \mathbf{l}' & \text{if } \ \mathbf{l}_2 \diamond_{\mathbf{x}} \mathbf{l} = \mathbf{l}' \ \text{ and } \ \mathbf{l}_1 +\!\!+ \mathbf{l}' \in \textsc{LData}_{\mathbb{L}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

We are almost in a position to formally define abstract logical heaps. An abstract logical heap is a mapping from addresses (Def. 9) to logical data (Par. 4). A logical heap may be: i) complete, with no use of abstract addresses in its domain or range, i.e. a program heap (e.g. the list heap in Fig. 2.1a); or ii) complete, but with its data split across several abstract heap cells (e.g. the list heap in Fig. 2.1b); or iii) incomplete, missing some heap cells needed to join some abstract addresses (e.g. the list subheap $\mathbf{y} \mapsto [n]$ in Fig. 2.1b). As we demonstrated in §2.1, incomplete heaps are necessary for local reasoning using the frame rule. In this case, there will be some choice for the missing cells in the frame that would render the heap complete. Logical heaps are subject to structural invariants to ensure that they are well-formed. In particular, a context hole $\mathbf{x}$ must not be reachable from the abstract address $\mathbf{x}$ in the domain of the heap. For instance, the list heap $\{\mathbf{x} \mapsto [n] +\!\!+ \mathbf{y}, \mathbf{y} \mapsto [m] +\!\!+ \mathbf{x}\}$ is not well-formed due to the cycle. To this end, we first define *logical pre-heaps* as mappings from addresses (Def. 9) to logical data (Par. 4). We then define *abstract logical heaps* as logical pre-heaps that are well-formed. More concretely, a logical pre-heap $\mathbf{ph}$ is well-formed if:

i) $\mathbf{ph}$ is complete with no abstract addresses in its domain or range, i.e. $\mathbf{ph}$ is a program heap in $\textsc{PHeap}_{\mathbb{A}}$ (Def. 7); or

ii) **ph** is complete with its data split across several abstract heap cells, and it is possible to *collapse* the data at abstract addresses to their counterpart context holes to produce a complete pre-heap as in i); or

iii) **ph** is incomplete and it is possible to *complete* it by extending it to a well-formed complete pre-heap as in ii).

Note that the *collapsibility* property in ii) ensures that logical heaps are acyclic with respect to abstract addresses, and dismisses malformed heaps such as $\{\mathbf{x} \mapsto [n]{+\!\!+}\mathbf{y}, \mathbf{y} \mapsto [m]{+\!\!+}\mathbf{x}\}$ since it cannot be collapsed to a complete heap. We proceed with the formal definition of logical pre-heaps, followed by the definitions of heap *collapse* and *completion* functions. We then formulate the definition of abstract logical heaps.

Definition 12 (Logical pre-heaps)**.** Given the set of library $\mathbb{A}$ addresses $\textsc{Add}_{\mathbb{A}}$ (Def. 9) and the set of logical data $\textsc{LData}_{\mathbb{A}}$ (Par. 4), the set of *logical pre-heaps for library* $\mathbb{A}$ is:

$$\mathbf{ph} \in \textsc{PreHeap}_{\mathbb{A}} \triangleq \textsc{Add}_{\mathbb{A}} \overset{\text{fin}}{\rightharpoonup} \textsc{LData}_{\mathbb{A}}$$

The *empty pre-heap*, $\mathbf{0}_{\mathbb{A}}$, is a function with an empty domain.

As before, when the type of an abstract heap is clear from the context, we drop the subscript and write $\mathbf{0}$ for $\mathbf{0}_{\mathbb{A}}$.

Definition 13 (Completion/collapse)**.** A pre-heap $\mathbf{ph} \in \textsc{PreHeap}_{\mathbb{A}}$ is *complete*, written $\mathsf{isComp}(\mathbf{ph})$, if and only if it satisfies the following condition, where $dom(\mathbf{ph})$ denotes the domain of $\mathbf{ph}$, and addr(.) is the address function (Par. 4):

$$\mathsf{isComp}(\mathbf{ph}) \overset{\text{def}}{\iff} \forall \mathbf{x}.\ \Big(\mathbf{x} \in dom(\mathbf{ph}) \Leftrightarrow \exists a \in dom(\mathbf{ph}).\ \mathbf{x} \in \mathrm{addr}(\mathbf{ph}(a))\Big)$$

The pre-heap *completion function*, $\mathrm{comp}(.) : \textsc{PreHeap}_{\mathbb{A}} \to \mathcal{P}\left(\textsc{PreHeap}_{\mathbb{A}}\right)$, is defined as follows, where $\uplus$ denotes the standard disjoint function union:

$$\mathrm{comp}(\mathbf{ph}) \triangleq \Big\{\mathbf{ph}' \;\Big|\; \exists \mathbf{ph}''.\ \mathbf{ph}'{=}\mathbf{ph} \uplus \mathbf{ph}'' \wedge \mathsf{isComp}(\mathbf{ph}')\Big\}$$

The pre-heap *collapse function*, $\mathrm{collapse}(.) : \textsc{PreHeap}_{\mathbb{A}} \rightharpoonup \textsc{PreHeap}_{\mathbb{A}}$, is defined as follows:

$$\text{collapse}(\mathbf{ph}) \triangleq \begin{cases} \mathbf{ph} & \text{if} \quad \mathsf{isComp}(\mathbf{ph}) \\ & \text{and} \quad \mathit{dom}(\mathbf{ph}) \cap \text{AA}\textsc{dd} = \emptyset \\[1em] \text{collapse}(\mathbf{ph}') & \text{if} \quad \mathsf{isComp}(\mathbf{ph}) \\ & \text{and} \quad \exists \mathbf{ph}'', a, \mathbf{x}, \mathbf{d}_1, \mathbf{d}_2. \\ & \qquad \mathbf{ph}{=}\mathbf{ph}'' \uplus [a \mapsto \mathbf{d}_1] \uplus [\mathbf{x} \mapsto \mathbf{d}_2] \\ & \qquad \wedge \mathbf{ph}'{=}\mathbf{ph}'' \uplus [a \mapsto (\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2)] \\[1em] \text{undefined} & \text{otherwise} \end{cases}$$

Observe that only *complete* pre-heaps with *unique* context holes in their range can be collapsed. That is, for any two distinct addresses a_1 and a_2 in the domain of $\mathbf{ph}$, the context holes of $\mathbf{ph}(a_1)$ and $\mathbf{ph}(a_2)$ should not overlap. As such, the order of data collapse does not matter. As the collapsibility of a pre-heap ensures its acyclicity with respect to abstract addresses, we define the set of abstract logical heaps to comprise logical pre-heaps that can be *completed* into *collapsible* pre-heaps.

Definition 14 (Abstract logical heaps)**.** Given the set of logical pre-heaps $\text{P}\textsc{re}\text{H}\textsc{eap}_{\mathbb{A}}$ (Def. 12), the set of *abstract logical heaps* for library $\mathbb{A}$, $\mathbf{h} \in \text{LH}\textsc{eap}_{\mathbb{A}}$, is defined as follows:

$$\mathbf{h} \in \text{LH}\textsc{eap}_{\mathbb{A}} \triangleq \left\{ \mathbf{ph} \;\middle|\; \begin{array}{l} \mathbf{ph} \in \text{P}\textsc{re}\text{H}\textsc{eap}_{\mathbb{A}} \wedge \exists \mathbf{ph}_1, \mathbf{ph}_2. \\ \quad \mathbf{ph}_1 \in \text{comp}(\mathbf{ph}) \wedge \mathbf{ph}_2{=}\text{collapse}(\mathbf{ph}_1) \end{array} \right\}$$

Logical heap composition, $\bullet_{\mathbb{A}} : \text{LH}\textsc{eap}_{\mathbb{A}} \times \text{LH}\textsc{eap}_{\mathbb{A}} \rightharpoonup \text{LH}\textsc{eap}_{\mathbb{A}}$, is defined as the standard disjoint function union $\uplus$. The *empty logical heap*, $\mathbf{0}_{\mathbb{A}}$, is the logical heap with empty domain. The separation algebra of logical heaps for library $\mathbb{A}$ is $\text{SA}_{\mathbb{A}} \triangleq (\text{LH}\textsc{eap}_{\mathbb{A}}, \bullet_{\mathbb{A}}, \mathbf{0}_{\mathbb{A}})$.

Observe that abstract heaps are also abstract pre-heaps: $\text{LH}\textsc{eap}_{\mathbb{A}} \subset \text{P}\textsc{re}\text{H}\textsc{eap}_{\mathbb{A}}$. When the type of abstract heaps is clear from the context, we drop the subscripts and write $\bullet$ for $\bullet_{\mathbb{A}}$ and $\mathbf{0}$ for $\mathbf{0}_{\mathbb{A}}$.

We can now formalise the abstract allocation/deallocation relation on abstract logical heaps.

Definition 15 (Abstract (de)allocation)**.** The *abstract (de)allocation relation*, $\approx$: $\text{LH}\textsc{eap}_{\mathbb{A}} \times \text{LH}\textsc{eap}_{\mathbb{A}}$, is defined as follows, where $*$ denotes the reflexive transitive closure of the relation and $\mathbf{h}_1[a \mapsto \mathbf{d}_1]$ denotes a function that behaves like $\mathbf{h}_1$ except that a is mapped onto $\mathbf{d}_1$:

$$\approx \triangleq \left\{ (\mathbf{h}_1, \mathbf{h}_2), (\mathbf{h}_2, \mathbf{h}_1) \;\middle|\; \exists a, \mathbf{d}_1, \mathbf{d}_2, \mathbf{x}. \begin{array}{l} \mathbf{h}_1(a) = (\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2) \wedge \\ \mathbf{h}_2 = \mathbf{h}_1[a \mapsto \mathbf{d}_1] \bullet [\mathbf{x} \mapsto \mathbf{d}_2] \end{array} \right\}^*$$

During abstract allocation (in the transition from $\mathbf{h}_1$ to $\mathbf{h}_2$), the subdata $\mathbf{d}_2$ at address a is split and promoted to a fresh abstract address $\mathbf{x}$ in the heap, leaving the context hole $\mathbf{x}$ behind in its place. Dually, during abstract deallocation (in the transition from $\mathbf{h}_2$ to $\mathbf{h}_1$), the context hole $\mathbf{x}$ in logical data $\mathbf{d}_1$ is replaced by its associated data $\mathbf{d}_2$ at abstract address $\mathbf{x}$, removing $\mathbf{x}$ from the domain of the heap in doing so.

An abstract library provides an interface for manipulating the underlying data structure. In SSL this is achieved through a set of atomic operations associated with the library without exposing their implementation details. As such, the operations of library $\mathbb{A}$ are parameterised and must be provided alongside the other SSL parameters studied so far. For instance, for the list library $\mathbb{L}$ in §2.1.3, the library operations comprise `x.add(n)`, `x.remove(n)` and `r := x.item(i)`.

SSL Parameter

Parameter 6 (Library operations)**.** Assume a set of *atomic library operations*, $\mathtt{C}_{\mathbb{A}} \in \textsc{Op}_{\mathbb{A}}$.

SSL $\mathbb{L}$ Instance (Parameter 6)

Definition 16 (List operations)**.** The set of *atomic list operations*, $\mathtt{C}_{\mathbb{L}} \in \textsc{Op}_{\mathbb{L}}$, is defined by the following grammar, for all program variables $\mathtt{x}, \mathtt{n}, \mathtt{r} \in \textsc{PVar}$ (Def. 1):

$$\mathtt{C}_{\mathbb{L}} ::= \mathtt{x.add(n)} \mid \mathtt{x.remove(n)} \mid \mathtt{r := x.item(i)}$$

We have now introduced all the ingredients of the SSL model. We proceed with the SSL assertions and their semantics.

SSL Assertions

As we discussed earlier at the beginning of this chapter, the set of *logical values* are library specific and may vary from one library to another. Logical values denote the values associated with logical variables. We thus assume a set of *library-specific logical values* that include library program values (Par. 3) as well as abstract addresses. For instance, in the list axioms of Fig. 2.2, the (logical) value of the logical variable α is an abstract address $\mathbf{x}$.

SSL Parameter

Parameter 7 (Library logical values)**.** Given the set of library program values $\textsc{PVal}_{\mathbb{A}}$ (Par. 3) and the set of abstract addresses $\textsc{AAdd}$ (Def. 8), assume a set of *library logical values*, $v \in \textsc{LVal}_{\mathbb{A}}$, such that $\textsc{PVal}_{\mathbb{A}} \cup \textsc{AAdd} \subseteq \textsc{LVal}_{\mathbb{A}}$.

For the list library $\mathbb{L}$, the logical values are the extension of list program values (Def. 6) with abstract addresses and logical list data (Def. 10). We include the logical list data in the set of logical values to allow for writing logical expressions that inspect the structure of list data.

SSL $\mathbb{L}$ Instance (Parameter 7)

Definition 17 (List logical values)**.** Given the set of program values for lists $\textsc{PVal}_{\mathbb{L}}$ (Def. 6), the set of abstract addresses $\textsc{AAdd}$ (Def. 8) and the set of list logical data $\textsc{LData}_{\mathbb{L}}$ (Def. 10), the set of *logical values for lists* is $v \in \textsc{LVal}_{\mathbb{L}} \triangleq \textsc{PVal}_{\mathbb{L}} \cup \textsc{AAdd} \cup \textsc{LData}_{\mathbb{L}}$.

Library $\mathbb{A}$ may specify a set of *logical expressions* in order to assert certain properties about the underlying data. For instance, the logical expressions of the list library studied in §2.1.2 include the $|\textsc{l}|$ expression describing the length of list $\textsc{l}$ (see the specification of `x.item(i)` operation in Fig. 2.2). As such, SSL is parametric in the set of library logical expressions and their evaluation function.

SSL Parameter

Parameter 8 (Library logical expressions)**.** Assume a set of *library logical expressions*, $e \in \textsc{LExp}_{\mathbb{A}}$.
Given the set of library logical values $\textsc{LVal}_{\mathbb{A}}$ (Par. 7) and the set of logical environments in $\textsc{LEnv}\langle\textsc{LVal}_{\mathbb{A}}\rangle$ (Def. 3), assume an *evaluation function*, $(\!|.|\!)^{(.)}_{\mathbb{A}} : (\textsc{LExp}_{\mathbb{A}} \times \textsc{LEnv}\langle\textsc{LVal}_{\mathbb{A}}\rangle) \rightharpoonup \textsc{LVal}_{\mathbb{A}}$, that given a logical environment in $\textsc{LEnv}\langle\textsc{LVal}_{\mathbb{A}}\rangle$, evaluates a logical expression in $\textsc{LExp}_{\mathbb{A}}$ to a logical value in $\textsc{LVal}_{\mathbb{A}}$.

For the list library $\mathbb{L}$, the logical expressions include logical variables, list expressions defined by the same grammar as that of logical list data (Def. 10), and the list length expression.

SSL $\mathbb{L}$ Instance (Parameter 8)

Definition 18 (List logical expressions)**.** The set of *logical expressions for lists*, $e \in \textsc{LExp}_{\mathbb{L}}$, is defined by the following grammar where $\textsc{l}, \textsc{n}, \alpha \in \textsc{LVar}$ (Def. 2):

$$e ::= [\,] \mid \textsc{l} \mid \alpha \mid [\textsc{n}] \mid e_1 {+\!\!+} e_2 \mid |e|$$

Given the set of logical values for lists $\textsc{LVal}_{\mathbb{L}}$ (Def. 17) and the set of logical environments $\textsc{LEnv}\langle\textsc{PVal}_{\mathbb{L}}\rangle$ (Def. 3), the *evaluation function* for list expressions, $(\!|.|\!)^{(.)}_{\mathbb{L}} : (\textsc{LExp}_{\mathbb{L}} \times \textsc{LEnv}\langle\textsc{LVal}_{\mathbb{L}}\rangle) \rightharpoonup \textsc{LVal}_{\mathbb{L}}$, is defined inductively over the structure of list expressions as follows, where $\Gamma \in \textsc{LEnv}\langle\textsc{LVal}_{\mathbb{L}}\rangle$:

$$(\!|[\,]|\!)^{\Gamma}_{\mathbb{L}} = [\,] \qquad (\!|\textsc{l}|\!)^{\Gamma}_{\mathbb{L}} = \Gamma(\textsc{l}) \qquad (\!|\alpha|\!)^{\Gamma}_{\mathbb{L}} = \begin{cases} \Gamma(\alpha) & \text{if } \Gamma(\alpha) \in \textsc{AAdd} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|[\textsc{n}]|\!)^{\Gamma}_{\mathbb{L}} = \begin{cases} [n] & \text{if } \Gamma(\textsc{n}) = n \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|e_1 {+\!\!+} e_2|\!)^{\Gamma}_{\mathbb{L}} = \begin{cases} \mathbf{l}_1 {+\!\!+} \mathbf{l}_2 & \text{if } (\!|e_1|\!)^{\Gamma}_{\mathbb{L}} = \mathbf{l}_1 \wedge (\!|e_2|\!)^{\Gamma}_{\mathbb{L}} = \mathbf{l}_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|\, |e| \,|\!)^{\Gamma}_{\mathbb{L}} = \begin{cases} |\mathbf{l}| & \text{if } (\!|e|\!)^{\Gamma}_{\mathbb{L}} = \mathbf{l} \wedge \mathrm{addr}_{\mathbb{L}}(\mathbf{l}) = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

Given a library $\mathbb{A}$, the *SSL assertions* for $\mathbb{A}$ comprise heap assertions describing sets of abstract heaps in $\textsc{LHeap}_{\mathbb{A}}$ (Def. 14). Heap assertions in turn are defined via *data assertions*, describing the underlying data in $\textsc{LData}_{\mathbb{A}}$. As well as assertions describing context holes, data assertions are parameterised and may be instantiated with library-specific data assertions. For instance, as we demonstrated in §2.1, the data assertions for the list library include $[\,]$, describing an empty list, as well as assertions of the form $\textsc{l}_1{+\!\!+}\textsc{l}_2$, describing composite lists. Data assertions are interpreted as sets of library-specific data given a logical environment.

SSL Parameter

Parameter 9 (Library data assertions)**.** Assume a set of *library data assertions* $\Lambda \in \textsc{LAst}_{\mathbb{A}}$.
Given the set of library logical values $\textsc{LVal}_{\mathbb{A}}$ (Par. 7), the set of logical environments $\textsc{LEnv}\,\langle \textsc{LVal}_{\mathbb{A}} \rangle$ (Def. 3) and the set of logical data $\textsc{LData}_{\mathbb{A}}$ (Par. 4), assume a *satisfiability relation* for library data assertions, $|||\!\!=_{\mathbb{A}}\colon (\textsc{LEnv}\,\langle \textsc{LVal}_{\mathbb{A}} \rangle \times \textsc{LData}_{\mathbb{A}}) \times \textsc{LAst}_{\mathbb{A}}$.

Given a set of library data assertions, we can now define the SSL heap and data assertions.

Definition 19 (SSL assertions)**.** The set of *library* $\mathbb{A}$ *heap assertions*, $\Theta \in \textsc{HAst}_{\mathbb{A}}$, and the set of *library* $\mathbb{A}$ *data assertions*, $\Delta \in \textsc{DAst}_{\mathbb{A}}$, are defined by the following grammars, where $\mathcal{R} \in \textsc{RAdd}_{\mathbb{A}}$ (Par. 1); $\alpha, \textsc{x} \in \textsc{LVar}$ (Def. 2); and $\Lambda \in \textsc{LAst}_{\mathbb{A}}$ (Par. 9):

$$\begin{array}{ll} \Theta ::= \mathcal{R} \mapsto \Delta & \text{root cell assertions} \\ \quad\;\; \mid \alpha \mapsto \Delta & \text{abstract cell assertions} \\ \\ \Delta ::= \mathsf{false} \mid \Delta_1 \Rightarrow \Delta_2 \mid \exists \textsc{x}.\, \Delta & \text{classical assertions} \end{array}$$

$\mid \mathrm{x}$	logical variable
$\mid \Delta_1 \diamond_\alpha \Delta_2$	context hole application
$\mid \Diamond\alpha$	context hole containment
$\mid \Lambda$	library data assertions

The $\mathcal{R} \mapsto \Delta$ assertion describes a logical heap comprising a single cell at root address $\mathcal{R}$ containing data described by data assertion Δ. Analogously, the $\alpha \mapsto \Delta$ assertion describes a logical heap comprising a single cell at the abstract address denoted by α containing data described by Δ.

For data assertions, classical assertions are standard. Other classical connectives $(\neg, \wedge, \vee, \forall)$ can be derived in the usual way. The x denotes a logical variable interpreted in the usual way by looking up its value in the logical environment. The $\Delta_1 \diamond_\alpha \Delta_2$ assertion (borrowed from context logic [7]) describes data satisfying Δ_1 with its context hole denoted by α replaced by data satisfying Δ_2. The $\Diamond\alpha$ assertion (read "somewhere α") describes data that contains the context hole α.

Recall that in order to specify the behaviour of library operations (e.g. the list library in §2.1.3), we appeal to standard separation logic (SL) assertions such as $P * Q$. On the other hand, observe that the SSL assertion language (Def. 19) comprises assertions for describing single-cell abstract heaps only, and does not include the standard SL assertions. As the underlying model of SSL is a separation algebra (Def. 14), it is straightforward to extend the SSL assertions with the standard SL assertions. However, we view SSL as an *add-on* to SL and expect the SSL assertions to be incorporated into an SL-based logic including the standard connectives such as $P * Q$.

Definition 20 (SSL satisfiability relations)**.** Given the library logical values $\textsc{LVal}_{\mathbb{A}}$ (Par. 7), the logical environments $\textsc{LEnv}\,\langle\textsc{LVal}_{\mathbb{A}}\rangle$ (Def. 3), the logical heaps $\textsc{LHeap}_{\mathbb{A}}$ (Def. 14) and the logical data $\textsc{LData}_{\mathbb{A}}$ (Par. 4), the *heap assertion satisfiability relation*, $\models_{\mathbb{A}}$: $(\textsc{LEnv}\,\langle\textsc{LVal}_{\mathbb{A}}\rangle \times \textsc{LHeap}_{\mathbb{A}}) \times \textsc{HAst}_{\mathbb{A}}$, and the *data assertion satisfiability relation*, $\Vert\!\models_{\mathbb{A}}$: $(\textsc{LEnv}\,\langle\textsc{LVal}_{\mathbb{A}}\rangle \times \textsc{LData}_{\mathbb{A}}) \times \textsc{DAst}_{\mathbb{A}}$, are defined as follows, where $\Vert|\!\models_{\mathbb{A}}$ denotes the satisfiability relation for $\mathbb{A}$-specific data assertions (Par. 9):

$$\Gamma, \mathbf{h} \models_{\mathbb{A}} \alpha \mapsto \Delta \quad \text{iff} \quad \exists \mathbf{x}, \mathbf{d}.\, \Gamma(\alpha){=}\mathbf{x} \wedge \mathit{dom}(\mathbf{h}){=}\{\mathbf{x}\} \\ \wedge\, \mathbf{h}(\mathbf{x}){=}\mathbf{d} \wedge \Gamma, \mathbf{d} \Vert\!\models_{\mathbb{A}} \Delta$$

$\Gamma, \mathbf{h} \models_{\mathbb{A}} \mathcal{R} \mapsto \Delta$	iff	$\exists \mathbf{d}.\ dom(\mathbf{h}){=}\{\mathcal{R}\} \wedge \mathbf{h}(\mathcal{R}){=}\mathbf{d} \wedge \Gamma, \mathbf{d} \mid\models_{\mathbb{A}} \Delta$
$\Gamma, \mathbf{d} \mid\models_{\mathbb{A}} \mathsf{false}$		never
$\Gamma, \mathbf{d} \mid\models_{\mathbb{A}} \Delta_1 \Rightarrow \Delta_2$	iff	$\Gamma, \mathbf{d} \mid\models_{\mathbb{A}} \Delta_1 \;\Rightarrow\; \Gamma, \mathbf{d} \mid\models_{\mathbb{A}} \Delta_2$
$\Gamma, \mathbf{d} \mid\models_{\mathbb{A}} \exists \textsc{x}.\ \Delta$	iff	$\exists v.\ \Gamma[\textsc{x} \mapsto v], \mathbf{d} \mid\models_{\mathbb{A}} \Delta$
$\Gamma, \mathbf{d} \mid\models_{\mathbb{A}} \textsc{x}$	iff	$\mathbf{d} = \Gamma(\textsc{x})$
$\Gamma, \mathbf{d} \mid\models_{\mathbb{A}} \Delta_1 \diamond_\alpha \Delta_2$	iff	$\exists \mathbf{d}_1, \mathbf{d}_2, \mathbf{x}.\ \Gamma(\alpha){=}\mathbf{x} \wedge \mathbf{d}{=}\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2 \wedge \Gamma, \mathbf{d}_1 \mid\models_{\mathbb{A}} \Delta_1 \wedge \Gamma, \mathbf{d}_2 \mid\models_{\mathbb{A}} \Delta_2$
$\Gamma, \mathbf{d} \mid\models_{\mathbb{A}} \Diamond\alpha$	iff	$\Gamma(\alpha) \in \mathrm{addr}(\mathbf{d})$
$\Gamma, \mathbf{d} \mid\models_{\mathbb{A}} \Lambda$	iff	$\Gamma, \mathbf{d} \mid\mid\models_{\mathbb{A}} \Lambda$

The list-specific data assertions comprise assertions to describe list data, including empty lists, context holes (where the associated list data has been split away, leaving behind a hole), singleton lists and composite lists.

SSL $\mathbb{L}$ Instance (Parameter 9)

Definition 21 (List data assertions)**.** The set of *data assertions for lists*, $\Lambda \in \textsc{LAst}_{\mathbb{L}}$, is defined by the following grammar, where $\Delta_1, \Delta_2 \in \textsc{DAst}_{\mathbb{L}}$ (Def. 19) and $\alpha, \textsc{n} \in \textsc{LVar}$ (Def. 2):

$$\Lambda ::= [\,] \mid \alpha \mid [\textsc{n}] \mid \Delta_1 {+\!\!+} \Delta_2$$

Given the logical values for lists $\textsc{LVal}_{\mathbb{L}}$ (Def. 17) and the logical environments $\textsc{LEnv}\langle\textsc{LVal}_{\mathbb{L}}\rangle$ (Def. 3), the *satisfiability relation for list-specific data assertions*, $\mid\mid\models_{\mathbb{L}}$: $(\textsc{LEnv}\langle\textsc{LVal}_{\mathbb{L}}\rangle \times \textsc{LData}_{\mathbb{L}}) \times \textsc{LAst}_{\mathbb{L}}$, is defined as follows, where $\Gamma \in \textsc{LEnv}\langle\textsc{LVal}_{\mathbb{L}}\rangle$, $\mathbf{l} \in \textsc{LData}_{\mathbb{L}}$ (Def. 10), $n \in \textsc{LElem}$ (Def. 5), $\textsc{AAdd}$ denotes the set of abstract addresses (Def. 8) and $\mid\models_{\mathbb{L}}$ denotes the list data satisfiability relation (Def. 20):

$\Gamma, \mathbf{l} \mid\mid\models_{\mathbb{L}} [\,]$	iff	$\mathbf{l} = [\,]$
$\Gamma, \mathbf{l} \mid\mid\models_{\mathbb{L}} \alpha$	iff	$\Gamma(\alpha){=}\mathbf{l} \wedge \mathbf{l} \in \textsc{AAdd}$
$\Gamma, \mathbf{l} \mid\mid\models_{\mathbb{L}} [\textsc{n}]$	iff	$\exists n.\ \Gamma(\textsc{n}){=}n \wedge \mathbf{l}{=}[n]$
$\Gamma, \mathbf{l} \mid\mid\models_{\mathbb{L}} \Delta_1 {+\!\!+} \Delta_2$	iff	$\exists \mathbf{l}_1, \mathbf{l}_2.\ \mathbf{l}{=}\mathbf{l}_1 {+\!\!+} \mathbf{l}_2 \wedge \Gamma, \mathbf{l}_1 \mid\models_{\mathbb{L}} \Delta_1 \wedge \Gamma, \mathbf{l}_2 \mid\models_{\mathbb{L}} \Delta_2$

We write $[\mathrm{N}_1, \ldots, \mathrm{N}_k]$ as a shorthand for $[\mathrm{N}_1] +\!\!+ \ldots +\!\!+ [\mathrm{N}_k]$.

Recall that in order to specify the behaviour of library operations (e.g. the list library in §2.1.3), we appeal to an abstract predicate, $\mathsf{vars}(\ldots)$, describing the values associated with program variables. However, the SSL assertion language (Def. 19) does not include assertions for describing a variable store. As the underlying model of SSL is a separation algebra (Def. 14), it is straightforward to extend the SSL model to incorporate a variable store (based on e.g. the variables-as-resource model [5]), and accordingly include assertions for describing the variable store.

In order to use the axiomatic specification of a library for client-side reasoning, the specification must ideally be impartial to the choice of client programming language. That is, since the library operations may be called by different client programs written in different programming languages, the specification must be agnostic to the variable store model in the client programming language. However, explicit modelling of the variable store as described above makes certain assumptions about the store model in the client programming language, and thus limits the usability of the specification. For instance, unlike most languages where program variables are tracked in a dedicated stack, in JavaScript the variable store is emulated in the heap. As such, modelling the variable store as a stack departs from the JavaScript model, and consequently our library specifications cannot be used to reason about client programs written in JavaScript.

To remedy this, we leave the choice of the client programming language open, so long as the client provides an accompanying SL-based program logic in order reason about its programs. In other words, since we expect SSL to be incorporated into an SL-based logic as an add-on, we further expect this SL-based logic to provide assertions describing the underlying variable store. In what follows, we demonstrate how to extend this SL-based program logic in order to enable client reasoning for library $\mathbb{A}$.

3.2. The PLogic$_{\mathbb{A}}$ Reasoning Framework

Given an abstract library $\mathbb{A}$, we show how to reason about its client programs in an arbitrary programming language with an SL-based program logic. More concretely, given a programming language PL, with an SL-based program logic PLogic, we demonstrate how to extend PLogic to PLogic$_{\mathbb{A}}$ to reason about the client programs of $\mathbb{A}$ written in PL, provided that PLogic meets certain assumptions.

To demonstrate our techniques better, we present two instances of our methodology. The first instance, WLogic, is due to Wright in his thesis [60] where he illustrates how to reason about SSL client programs written in a simple while language by extending the program logic of the Views framework [14]. We use WLogic in §4 in order to reason about the client programs of a tree library. In the second instance (§5), we show how to reason about the JavaScript client programs of the DOM (Document Object Model) library, by extending the SL-based JavaScript program logic of [21].

As described above, in order to extend PLogic to $\text{PLogic}_{\mathbb{A}}$, we assume that PLogic meets certain assumptions. We delineate these assumptions in solid boxes labelled "PLogic Parameter". To provide the reader with a clearer account of our extension methodology, we follow each PLogic parameter with its corresponding instantiation for WLogic, enclosed in dashed boxes labelled "WLogic Instance (Parameter X)" where X is a reference to the corresponding PLogic parameter.

Although we demonstrate how to extend PLogic with a *single* library $\mathbb{A}$, the approach described here may be used to extend PLogic with multiple libraries. This can be done by extending PLogic with one library at a time, e.g. first extending PLogic to $\text{PLogic}_{\mathbb{A}}$, then extending $\text{PLogic}_{\mathbb{A}}$ to $\text{PLogic}_{\mathbb{A}\mathbb{B}}$ and so forth.

In what follows, we visit the various components of PLogic, stipulate the conditions they must meet, and describe how we extend them in $\text{PLogic}_{\mathbb{A}}$ to enable client reasoning for library $\mathbb{A}$.

Program values We assume a set of PL program values describing the set of values that may be observed via program variables. In order to reason about the client programs of $\mathbb{A}$, in $\text{PLogic}_{\mathbb{A}}$ we extend the set of PL program value with those of library $\mathbb{A}$ in $\text{PVal}_{\mathbb{A}}$ (Par. 3).

PLogic Parameter

Parameter 10 (PL program values)**.** Assume a set of PL *program values* $v \in \text{PLPVal}$.

Definition 22 (PL$_{\mathbb{A}}$ program values)**.** Given the set of library program values $\textsc{PVal}_{\mathbb{A}}$ (Par. 3), the set of PL$_{\mathbb{A}}$ *program values* is $v \in \textsc{PLPVal}_{\mathbb{A}} \triangleq \textsc{PLPVal} \cup \textsc{PVal}_{\mathbb{A}}$.

WL program values The set of program values for WL comprises integer values, boolean values and the `null` location.

WLogic Instance (Parameter 10)

Definition 23 (WL program values)**.** The set of WL program values is $\textsc{WPVal} \triangleq \mathbb{Z} \cup \{\mathsf{true}, \mathsf{false}, \mathtt{null}\}$.

Example 1 (WL$_{\mathbb{L}}$ program values)**.** Given the list program values $\textsc{PVal}_{\mathbb{L}}$ (Def. 6) and the WL program values WPVal (Def. 23), the WL$_{\mathbb{L}}$ program values are $\textsc{WPVal}_{\mathbb{L}} \triangleq \textsc{WPVal} \cup \textsc{PVal}_{\mathbb{L}}$.

Program states Recall that the interaction between the programs written in PL and a library $\mathbb{A}$ are carried out via program variables. Therefore, a PL program state must embody a variable store representation in the PL language. Furthermore, the representation of the variable store must be parametric in the choice of the values associated with program variables. This is to ensure that we can extend the set of possible program values with those of library $\mathbb{A}$ so that the PL program variables may record $\mathbb{A}$-specific values. We thus assume a set of PL program states $\textsc{PState}\langle \mathrm{V} \rangle$, parametric in the choice of program values $\mathrm{V} \supseteq \textsc{PLPVal}$ (Par. 10), denoting a generic extension of PL program values. To incorporate the PL program states with those of library $\mathbb{A}$, in PL$_{\mathbb{A}}$ we instantiate the PL program states with PL$_{\mathbb{A}}$ program values as $\textsc{PState}\langle \textsc{PLPVal}_{\mathbb{A}} \rangle$, and further extend the states to incorporate abstract program heaps (Def. 7). That is, we define a program state to be a pair, (s, h), comprising a PL program state $s \in \textsc{PState}\langle \textsc{PLPVal}_{\mathbb{A}} \rangle$ and an abstract program heap $\mathrm{h} \in \textsc{PHeap}_{\mathbb{A}}$.

PLogic Parameter

Parameter 11 (PL program states)**.** Given the set of PL program values PLPVal (Par. 10) and a generic extension of PL program values $\mathrm{V} \supseteq \mathrm{PLPVal}$, assume a set of *parametric program states* $\mathrm{PState}\langle \mathrm{V} \rangle$. Assume that the set of PL program states is $\mathrm{PState}\langle \mathrm{PLPVal} \rangle$.

Definition 24 ($\mathrm{PL}_{\mathbb{A}}$ program states)**.** Given the set of $\mathrm{PL}_{\mathbb{A}}$ program values $\mathrm{PLPVal}_{\mathbb{A}}$ (Def. 22), the set of parametric program states $\mathrm{PState}\langle . \rangle$ (Par. 11) and the set of abstract program heaps $\mathrm{PHeap}_{\mathbb{A}}$ (Def. 7), the set of $\mathrm{PL}_{\mathbb{A}}$ *program states* is: $w \in \mathrm{PLPStates}_{\mathbb{A}} \triangleq \mathrm{PState}\langle \mathrm{PLPVal}_{\mathbb{A}} \rangle \times \mathrm{PHeap}_{\mathbb{A}}$.

WL program states The WL program states comprise a designated *fault* state $\lightning$, as well as pairs of the form (σ, h) where σ denotes a *variable stack* and h denotes a *heap*. A variable stack is based on the *variable-as-resource* model introduced by Bornat, Calcagno and Yang in [5]. The variables-as-resource model treats program variables as (spatial) resource, much like heap cells. This removes the need for the side-condition on the frame and parallel composition rules. A *variable stack* is a function mapping program variables onto program values. Similarly, a *heap* is a function mapping addresses (in $\mathbb{N}^{+}$) onto program values. In order to enable client reasoning, variable stacks are parametric in the choice of program values (the values in their range) and may be extended with library values. That is, since program variables are the means of interaction between the client programming language and the library, we allow their values to extend beyond WL program values to include library values. On the other hand, unlike variable stacks, heaps are not parametric in the choice of program values. In other words, we do not allow the underlying memory to have pointers into the library and variable stacks are the sole point of interaction between the client and the library.

WLogic Instance (Parameter 11)

Definition 25 (WL program states)**.** Let $\mathrm{V} \supseteq \mathrm{WPVal}$ denote a generic extension of WL program values (Def. 23).

The set of *parametric variable stacks* is:

$$\sigma \in \textsc{Stack}\langle \mathrm{V} \rangle \triangleq \textsc{PVar} \xrightarrow{\text{fin}} \mathrm{V}$$

The set of *heaps* is $h \in \textsc{Heap} \triangleq \mathbb{N}^+ \xrightarrow{\text{fin}} \textsc{WPVal}$.
The set of WL *parametric program states* is:

$$\textsc{WPState}\langle \mathrm{V} \rangle \triangleq (\textsc{Stack}\langle \mathrm{V} \rangle \times \textsc{Heap}) \uplus \{\lightning\}$$

The set of WL *program states* is:

$$s \in \textsc{WPState} \triangleq \textsc{WPState}\langle \textsc{WPVal} \rangle$$

Example 2 (WL$_{\mathbb{L}}$ program states)**.** Given the WL$_{\mathbb{L}}$ program values $\textsc{WPVal}_{\mathbb{L}}$ (Example 1), the list program heaps $\textsc{PHeap}_{\mathbb{L}}$ (Def. 7) and the parametric WL program states $\textsc{WPState}\langle . \rangle$ (Def. 25), the WL$_{\mathbb{L}}$ program states are:

$$\textsc{WPState}_{\mathbb{L}} \triangleq \textsc{WPState}\langle \textsc{WPVal}_{\mathbb{L}} \rangle \times \textsc{PHeap}_{\mathbb{L}}$$

Programming language We assume the PL programming language to be defined by an inductive grammar comprising a set of *primitive operations* (e.g. variable assignment in PL) in $\textsc{PrimPLOp}$, as well as a set of *composite operations* (e.g. sequential composition ;) in $\textsc{Op}\langle \mathrm{O} \rangle$, *parametric* in the choice of primitive operations used as the "building blocks" of composite operations. That is, for a generic extension of primitive operations, $\mathrm{O} \supseteq \textsc{PrimPLOp}$, we assume $\textsc{Op}\langle \mathrm{O} \rangle$ to define the set of operations with its "building blocks" (i.e. the base cases in the inductive grammar) drawn from O. In PL$_{\mathbb{A}}$ we extend the primitive operations of PL with those of atomic library operations in $\textsc{Op}_{\mathbb{A}}$ (Par. 6); that is, $\textsc{PrimPLOp}_{\mathbb{A}} \triangleq \textsc{PrimPLOp} \uplus \textsc{Op}_{\mathbb{A}}$. We then instantiate the parametric composite operations to take into account the extended primitive operation set (i.e. inhabit the $\textsc{Op}\langle \textsc{PrimPLOp}_{\mathbb{A}} \rangle$ set).

PLogic Parameter

Parameter 12 (PL operations)**.** Assume a set of *primitive operations* $\textsc{PrimPLOp}$.
Given a generic extension of primitive operations $\mathrm{O} \supseteq \textsc{PrimPLOp}$, assume a set of *parametric operations* $\textsc{Op}\langle \mathrm{O} \rangle$ such that $\mathrm{O} \subseteq \textsc{Op}\langle \mathrm{O} \rangle$.
Assume that the set of PL *operations* is $\mathtt{C} \in \textsc{PLOp} \triangleq \textsc{Op}\langle \textsc{PrimPLOp} \rangle$.

Definition 26 ($\textsc{PLogic}_{\mathbb{A}}$ operations)**.** Given the primitive PL operations $\textsc{PrimPLOp}$ (Par. 12) and the atomic library operations $\textsc{Op}_{\mathbb{A}}$ (Par. 6), the set of *primitive* $\mathrm{PL}_{\mathbb{A}}$ *operations* is $\textsc{PrimPLOp}_{\mathbb{A}} \triangleq \textsc{PrimPLOp} \uplus \textsc{Op}_{\mathbb{A}}$.
The set of $\mathrm{PL}_{\mathbb{A}}$ *operations* is $\mathtt{C} \in \textsc{PLOp}_{\mathbb{A}} \triangleq \textsc{Op}\langle \textsc{PrimPLOp}_{\mathbb{A}} \rangle$.

WL programming language The programming language of WL is a simple while language and consists of i) heap operations; ii) variable stack operations; and iii) standard inductive programming constructs. The heap operations comprise memory allocation and deallocation (`x = alloc()` and `free(x)`), cell dereference (`x = [y]`) and cell update (`[x] = y`). The stack operations comprise variable assignment (`x = e`). The inductive constructs comprise `skip`, scoped variable declaration (`var x in {C}`), sequential composition (`;`), parallel composition (`||`), conditional loop (`while (b) {C}`), and conditional choice (`if (b) then {C} else {C'}`). As per the stipulations delineated in Par. 12, the WL operations are to be categorised as either *primitive* or *composite*. Moreover, as we demonstrate shortly, while the semantics and proof rules of primitive operations must not depend on the library extension (i.e. not refer to the abstract heaps of the library), the semantics and proof rules of composite operations may be lifted to account for the extension as required. Observe that neither stack nor heap operations depend on the abstract program heaps of the library. In other words, the only way to manipulate the abstract program heaps is either via library operations, or the inductive constructs built from library operations. As such, we declare both stack and heap operations as primitive, while categorising the inductive constructs as composite so that their semantics and proof rules are lifted accordingly upon extension with library operations.

WLogic Instance (Parameter 12)

Definition 27 (WL operations)**.** The set of WL *program expressions*, $\mathtt{e} \in \textsc{WExp}$, is defined by the following grammar, where $\mathtt{x} \in \textsc{PVar}$ (Def. 1) and $v \in \textsc{WPVal}$ (Def. 23):

$$\mathtt{e} ::= v \mid \mathtt{x}$$

Let $\mathrm{V} \supseteq \textsc{WPVal}$ denote a generic extension of WL program values. The WL *parametric program expression evaluation function*, $(\!|.|\!)^{(\cdot)} : \textsc{WExp} \times \textsc{Stack}\langle \mathrm{V} \rangle \rightharpoonup \mathrm{V}$, is defined inductively over the structure of program expressions as follows:

$$(\!|v|\!)^{\sigma} \triangleq v \qquad\qquad (\!|\mathtt{x}|\!)^{\sigma} \triangleq \begin{cases} \sigma(\mathtt{x}) & \text{if } \mathtt{x} \in \mathit{dom}(\sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The set of WL *boolean expressions*, $\mathtt{b} \in \textsc{WBExp}$, is defined by the following grammar, where $\mathtt{e} \in \textsc{WExp}$:

$$\mathtt{b} ::= \mathtt{e} \mid !\mathtt{b} \mid \mathtt{b}_1 \&\& \mathtt{b}_2 \mid \mathtt{b}_1 = \mathtt{b}_2$$

The WL *parametric boolean expression evaluation function*, $\langle\!|.|\!\rangle^{(\cdot)} : \textsc{WBExp} \times \textsc{Stack}\langle \mathrm{V} \rangle \rightharpoonup \{\mathsf{true}, \mathsf{false}\}$, is defined inductively over the structure of program expressions as follows:

$$\langle\!|\mathtt{e}|\!\rangle^{\sigma} \triangleq \begin{cases} (\!|\mathtt{e}|\!)^{\sigma} & \text{if } (\!|\mathtt{e}|\!)^{\sigma} \in \{\mathsf{true}, \mathsf{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\langle\!|!\mathtt{b}|\!\rangle^{\sigma} \triangleq \begin{cases} \neg \langle\!|\mathtt{b}|\!\rangle^{\sigma} & \text{if } \langle\!|\mathtt{b}|\!\rangle^{\sigma} \in \{\mathsf{true}, \mathsf{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\langle\!|\mathtt{b}_1 \&\& \mathtt{b}_2|\!\rangle^{\sigma} \triangleq \begin{cases} \langle\!|\mathtt{b}_1|\!\rangle^{\sigma} \wedge \langle\!|\mathtt{b}_1|\!\rangle^{\sigma} & \text{if } \langle\!|\mathtt{b}_1|\!\rangle^{\sigma}, \langle\!|\mathtt{b}_2|\!\rangle^{\sigma} \in \{\mathsf{true}, \mathsf{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llparenthesis \mathtt{b}_1 = \mathtt{b}_2 \rrparenthesis^\sigma \triangleq \begin{cases} (\!|\mathtt{b}_1|\!)^\sigma = (\!|\mathtt{b}_2|\!)^\sigma & \text{if } \mathtt{b}_1, \mathtt{b}_2 \in \text{WE}\textsc{xp} \\ & \text{and } (\!|\mathtt{b}_1|\!)^\sigma, (\!|\mathtt{b}_2|\!)^\sigma \neq \text{undefined} \\ (\!|\mathtt{b}_1|\!)^\sigma = \llparenthesis \mathtt{b}_2 \rrparenthesis^\sigma & \text{if } \mathtt{b}_1 \in \text{WE}\textsc{xp} \\ & \text{and } (\!|\mathtt{b}_1|\!)^\sigma, \llparenthesis \mathtt{b}_2 \rrparenthesis^\sigma \neq \text{undefined} \\ \llparenthesis \mathtt{b}_1 \rrparenthesis^\sigma = (\!|\mathtt{b}_2|\!)^\sigma & \text{if } \mathtt{b}_2 \in \text{WE}\textsc{xp} \\ & \text{and } \llparenthesis \mathtt{b}_1 \rrparenthesis^\sigma, (\!|\mathtt{b}_2|\!)^\sigma \neq \text{undefined} \\ \llparenthesis \mathtt{b}_1 \rrparenthesis^\sigma = \llparenthesis \mathtt{b}_2 \rrparenthesis^\sigma & \text{if } \llparenthesis \mathtt{b}_1 \rrparenthesis^\sigma, \llparenthesis \mathtt{b}_2 \rrparenthesis^\sigma \neq \text{undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The set of *primitive* WLogic *operations*, PrimWOp, is defined by the following grammar:

$$\text{P}\textsc{rim}\text{WO}\textsc{p} \ni \mathtt{C_W} ::= \mathtt{x = alloc()} \mid \mathtt{free(x)} \mid \mathtt{x = [y]} \mid \mathtt{[x] = y} \mid \mathtt{x = e}$$

Given a generic extension of primitive WL operations $\text{O} \supseteq$ PrimWOp, the WL *parametric operations*, $\text{WO}\textsc{p}\langle \text{O} \rangle$, is defined by the following grammar, where $o \in \text{O}$:

$$\begin{aligned} \text{WO}\textsc{p}\langle \text{O} \rangle \ni \mathtt{C} ::= & \; o \mid \mathtt{skip} \mid \mathtt{var\ x\ in\ \{C\}} \mid \mathtt{C;C'} \mid \mathtt{C||C'} \\ & \mid \mathtt{if\,(b)\,then\,\{C\}\,else\,\{C'\}} \mid \mathtt{while\,(b)\,\{C\}} \end{aligned}$$

The set of WLogic *operations* is $\text{WO}\textsc{p} \triangleq \text{WO}\textsc{p}\langle \text{P}\textsc{rim}\text{WO}\textsc{p} \rangle$.

Example 3 (WL$_\mathbb{L}$ operations)**.** Given the list library operations $\text{O}\textsc{p}_\mathbb{L}$ (Def. 16) and the WL primitive operations PrimWOp (Def. 27), the set of WL$_\mathbb{L}$ primitive operations is $\text{P}\textsc{rim}\text{WO}\textsc{p}_\mathbb{L} \triangleq \text{P}\textsc{rim}\text{WO}\textsc{p} \cup \text{O}\textsc{p}_\mathbb{L}$.
Given the WL parametric operations $\text{WO}\textsc{p}\langle . \rangle$ (Def. 27), the set of WL$_\mathbb{L}$ operations is $\text{WO}\textsc{p}_\mathbb{L} \triangleq \text{WO}\textsc{p}\langle \text{P}\textsc{rim}\text{WO}\textsc{p}_\mathbb{L} \rangle$

Semantics We require PL to define the semantics of its operations by associating each PL operation with a *state transformer*. A state transformer is a function describing how a program state may be manipulated by the corresponding operation. As we demonstrate shortly for WL, it is straightforward to redefine the operational semantics of a language (both small-step and big-step) using state transformers. As before, we require the state transformers to follow the structure of primitive and composite PL operations. For $\text{PL}_{\mathbb{A}}$, we modify the low-level semantics of PL and redefine it in terms of the extended program states incorporating abstract program heaps. We further extend the semantics to incorporate the operations of library $\mathbb{A}$. While it is possible for a library to describe the low-level semantics of its operations, the behaviour of library operations is often described axiomatically at a high-level. The abstract specification of library operations is then justified with respect to their implementation rather than their low-level semantics. As such, when this is the case it seems unsuitable to require the invention of a low-level semantics for the library. Therefore, we declare the low-level semantics of library $\mathbb{A}$ operations as an *optional* parameter. As we demonstrate shortly, when no low-level semantics is provided, we describe the semantics of library $\mathbb{A}$ operations denotationally via their high-level axiomatic semantics (Par. 18). We thus delay the formulation of the low-level library semantics until we present its axiomatic semantics.

PLogic Parameter

Parameter 13 (PL semantics)**.** Let $\text{V} \supseteq \text{PLPVal}$ denote a generic extension of PL program values (Par. 10).
Given the set of primitive PL operations PrimPLOp (Par. 12), assume a *primitive semantics function,* $⦇.⦈_{\text{PL}} : \text{PrimPLOp} \to \text{PState}\langle\text{V}\rangle \to \mathcal{P}(\text{PState}\langle\text{V}\rangle)$, that associates each primitive operation in PrimPLOp with a state transformer.
Given the parametric PL program states $\text{PState}\langle\text{V}\rangle$ (Par. 11), let $\text{S} \triangleq \text{PState}\langle\text{V}\rangle \times \text{C}$ denote a generic extension of PL program states, comprising parametric PL program states, as well as library program states captured by C. Given a generic extension of primitive PL operations $\text{O} \supseteq \text{PrimPLOp}$ (Par. 12), assume a *parametric semantics function*,

$$\mathsf{sem}(.) : \big(\text{O} \to \text{S} \to \mathcal{P}(\text{S})\big) \to \text{Op}\langle\text{O}\rangle \to \text{S} \to \mathcal{P}(\text{S})$$

that given a semantics function $f : \text{O} \to \text{S} \to \mathcal{P}(\text{S})$ for primitive operations in O, it associates each operation in $\text{Op}\langle\text{O}\rangle$ with a state transformer, provided that for all $o \in \mathrm{O}$:

$$\mathsf{sem}(f)(o) \triangleq f(o)$$

Assume that the PL *semantics function*, $[\![.]\!]_{\text{PL}} : \text{PLOP} \to \text{PLPSTATES} \to \mathcal{P}(\text{PLPSTATES})$, is defined as follows:

$$[\![.]\!]_{\text{PL}} \triangleq \mathsf{sem}(\langle\!|.|\!\rangle_{\text{PL}})$$

Definition 28 ($\text{PL}_{\mathbb{A}}$ semantics)**.** Given the PL primitive semantics function $\langle\!|.|\!\rangle_{\text{PL}}$ (Par. 13) and the library semantics function $\langle\!|.|\!\rangle_{\mathbb{A}}$ (Par. 20), the $\text{PL}_{\mathbb{A}}$ *primitive semantics function*, $\langle\!|.|\!\rangle_{\text{PL}\mathbb{A}} : \text{PRIMPLOP}_{\mathbb{A}} \to \text{PLPSTATES}_{\mathbb{A}} \to \mathcal{P}(\text{PLPSTATES}_{\mathbb{A}})$, is defined as follows, where $\mathtt{C}_{\text{PL}} \in \text{PRIMPLOP}$, $\mathtt{C}_{\mathbb{A}} \in \text{OP}_{\mathbb{A}}$ and $(s, \mathrm{h}) \in \text{PLPSTATES}_{\mathbb{A}}$:

$$\langle\!|\mathtt{C}_{\text{PL}}|\!\rangle_{\text{PL}\mathbb{A}}(s, \mathrm{h}) \triangleq \{(s', \mathrm{h}) \mid s' \in \langle\!|\mathtt{C}_{\text{PL}}|\!\rangle_{\text{PL}}(s)\}$$
$$\langle\!|\mathtt{C}_{\mathbb{A}}|\!\rangle_{\text{PL}\mathbb{A}}(s, \mathrm{h}) \triangleq \langle\!|\mathtt{C}_{\mathbb{A}}|\!\rangle_{\mathbb{A}}(s, \mathrm{h})$$

Given the PL parametric semantics function $\mathsf{sem}(.)$ (Par. 13), the $\text{PL}_{\mathbb{A}}$ *semantics function*, $[\![.]\!]_{\text{PL}\mathbb{A}} : \text{PLOP}_{\mathbb{A}} \to \text{PLPSTATES}_{\mathbb{A}} \to \mathcal{P}(\text{PLPSTATES}_{\mathbb{A}})$, is defined as follows:

$$[\![.]\!]_{\text{PL}\mathbb{A}} \triangleq \mathsf{sem}(\langle\!|.|\!\rangle_{\text{PL}\mathbb{A}})$$

WL semantics As stipulated by Par. 13, we define a primitive semantics function associating each primitive WL operation in PRIMWOP with a state transformer. We then define a parametric semantics function that given a generic extension of WL primitive operations $\mathrm{O} \supseteq \text{PRIMWOP}$ and a semantics function for the primitive operations in O, it associates each operation in $\text{OP}\langle\mathrm{O}\rangle$ with a state transformer. To do this, we define a small-step transition system, $\to_{(.)}$, capturing the small-step (structural) operational semantics of the operations in $\text{OP}\langle\mathrm{O}\rangle$. That is, given a primitive semantics function f describing the semantics of the primitive operations in O, then $\to_f$ produces a transition system describing the small-step semantics of the operations in $\text{OP}\langle\mathrm{O}\rangle$. Using the small-step transition system $\to_f$, we define a big-step transition system $\rightsquigarrow_f$, capturing the *terminating* traces, i.e. those ending with `skip`. The semantics of an operation is then defined via the big-step transition system.

Recall that scoped variable declaration, `var x in {C}`, defines a local variable `x` for the duration of `C` (i.e. scoped in `C`). As the current scope may already contain a variable named `x`, upon executing `var x in {C}` we i) extend the current stack with a *fresh* variable named `z`; ii) substitute all occurrences of `x` in `C` with `z`; and iii) remove `z` from the stack upon termination of `C`. To describe the small-step semantics of `var x in {C}`, we appeal to an auxiliary operation, `rem(z)`, describing the removal of `z` from the stack upon termination of `C` as described above.

WLogic Instance (Parameter 13)

Definition 29 (WLogic semantics)**.** Let $\mathrm{V} \supseteq \mathrm{WPVal}$ denote a generic extension of WL program values (Def. 23).
The primitive WL semantics function, $\langle\!| . |\!\rangle_{\mathrm{WL}} : \mathrm{PrimWOp} \to \mathrm{PState}\langle \mathrm{V} \rangle \to \mathcal{P}(\mathrm{PState}\langle \mathrm{V} \rangle)$, is defined over the structure of primitive operations as follows, where given a function g, $g[a \mapsto b]$ denotes a function that behaves as g, except that a is mapped to b:

$$\begin{aligned}
\langle\!| \mathtt{x = alloc()} |\!\rangle_{\mathrm{WL}}(\sigma, h) &\triangleq \left\{ (\sigma[\mathtt{x} \mapsto i], h \uplus [i \mapsto v]) \;\middle|\; \begin{array}{l} \mathtt{x} \in dom(\sigma) \\ \wedge\, i \in \mathbb{N}^+ \backslash dom(h) \\ \wedge\, v \in \mathrm{WPVal} \end{array} \right\} \\
&\quad \cup \left\{ ↯ \;\middle|\; \mathtt{x} \notin dom(\sigma) \right\} \\
\langle\!| \mathtt{free(x)} |\!\rangle_{\mathrm{WL}}(\sigma, h) &\triangleq \left\{ (\sigma, h') \;\middle|\; \sigma(\mathtt{x}) {=} i \wedge h {=} h' \uplus [i \mapsto -] \right\} \\
&\quad \cup \left\{ ↯ \;\middle|\; \mathtt{x} \notin dom(\sigma) \vee \sigma(\mathtt{x}) \notin dom(h) \right\} \\
\langle\!| \mathtt{x = [y]} |\!\rangle_{\mathrm{WL}}(\sigma, h) &\triangleq \left\{ (\sigma[\mathtt{x} \mapsto h(i)], h) \;\middle|\; \sigma(\mathtt{y}) {=} i \wedge i \in dom(h) \right\} \\
&\quad \cup \left\{ ↯ \;\middle|\; \begin{array}{l} \mathtt{x} \notin dom(\sigma) \vee \mathtt{y} \notin dom(\sigma) \\ \vee\, \sigma(\mathtt{y}) \notin dom(h) \end{array} \right\} \\
\langle\!| \mathtt{[x] = y} |\!\rangle_{\mathrm{WL}}(\sigma, h) &\triangleq \left\{ (\sigma, h[i \mapsto \sigma(\mathtt{y})]) \;\middle|\; \begin{array}{l} \sigma(\mathtt{x}) {=} i \wedge i \in dom(h) \\ \wedge\, \sigma(\mathtt{y}) \in \mathrm{WPVal} \end{array} \right\} \\
&\quad \cup \left\{ ↯ \;\middle|\; \begin{array}{l} \mathtt{x} \notin dom(\sigma) \vee \mathtt{y} \notin dom(\sigma) \\ \vee\, \sigma(\mathtt{x}) \notin dom(h) \vee \sigma(\mathtt{y}) \notin \mathrm{WPVal} \end{array} \right\}
\end{aligned}$$

$$\frac{s{\neq}(\lightning, -) \quad s' \in f(o)(s)}{(o, s) \rightarrow_f (\texttt{skip}, s')}$$

$$\frac{\sigma'{=}\sigma \uplus [\texttt{z} \mapsto v] \quad v \in \mathrm{V}}{(\texttt{var x in \{C\}}, (\sigma, h), c)) \rightarrow_f (\texttt{C[z/x]}; \texttt{rem(z)}, ((\sigma', h), c))}\ \mathsf{fresh}(\texttt{z})$$

$$\frac{\sigma{=}\sigma' \uplus [\texttt{x} \mapsto v]}{(\texttt{rem(x)}, ((\sigma, h), c)) \rightarrow_f (\texttt{skip}, ((\sigma', h), c))} \qquad \frac{s{=}((\sigma, h), c) \quad \texttt{x} \notin \mathit{dom}(\sigma)}{(\texttt{rem(x)}, s) \rightarrow_f (\texttt{skip}, (\lightning, c))}$$

$$\frac{(\texttt{C}_1, s) \rightarrow_f (\texttt{C}_1', s')}{(\texttt{C}_1; \texttt{C}_2, s) \rightarrow_f (\texttt{C}_1'; \texttt{C}_2, s')} \qquad \frac{s{\neq}(\lightning, -)}{(\texttt{skip}; \texttt{C}, s) \rightarrow_f (\texttt{C}, s)}$$

$$\frac{(\texttt{C}_1, s) \rightarrow_f (\texttt{C}_1', s')}{(\texttt{C}_1 \,||\, \texttt{C}_2, s) \rightarrow_f (\texttt{C}_1' \,||\, \texttt{C}_2, s')} \qquad \frac{s{\neq}(\lightning, -)}{(\texttt{skip} \,||\, \texttt{C}, s) \rightarrow_f (\texttt{C}, s)}$$

$$\frac{(\texttt{C}_2, s) \rightarrow_f (\texttt{C}_2', s')}{(\texttt{C}_1 \,||\, \texttt{C}_2, s) \rightarrow_f (\texttt{C}_1 \,||\, \texttt{C}_2', s')} \qquad \frac{s{\neq}(\lightning, -)}{(\texttt{C} \,||\, \texttt{skip}, s) \rightarrow_f (\texttt{C}, s)}$$

$$\frac{s{=}((\sigma, h), c) \quad \llparenthesis \texttt{b} \rrparenthesis^{\sigma}{=}\mathsf{true}}{(\texttt{if(b)then\{C}_1\texttt{\}else\{C}_2\texttt{\}}, s) \rightarrow_f (\texttt{C}_1, s)}$$

$$\frac{s{=}((\sigma, h), c) \quad \llparenthesis \texttt{b} \rrparenthesis^{\sigma}{=}\mathsf{false}}{(\texttt{if(b)then\{C}_1\texttt{\}else\{C}_2\texttt{\}}, s) \rightarrow_f (\texttt{C}_2, s)}$$

$$\frac{s{=}((\sigma, h), c) \quad \llparenthesis \texttt{b} \rrparenthesis^{\sigma}\text{undefined}}{(\texttt{if (b) then \{C}_1\texttt{\} else \{C}_2\texttt{\}}, s) \rightarrow_f (\texttt{skip}, (\lightning, c))}$$

$$\frac{s{=}((\sigma, h), c) \quad \llparenthesis \texttt{b} \rrparenthesis^{\sigma}{=}\mathsf{true}}{\big(\texttt{while (b) \{C\}}, s\big) \rightarrow_f \big(\texttt{C}; \texttt{while (b) \{C\}}, s\big)}$$

$$\frac{s{=}((\sigma, h), c) \quad \llparenthesis \texttt{b} \rrparenthesis^{\sigma}{=}\mathsf{false}}{\big(\texttt{while (b) \{C\}}, s\big) \rightarrow_f \big(\texttt{skip}, s\big)} \qquad \frac{s{=}((\sigma, h), c) \quad \llparenthesis \texttt{b} \rrparenthesis^{\sigma}\text{undefined}}{\big(\texttt{while (b) \{}a\texttt{\}}, s\big) \rightarrow_f (\texttt{skip}, (\lightning, c))}$$

Figure 3.1.: The WL parametric small-step transitions

$$\llparenthesis \mathtt{x} = \mathtt{e} \rrparenthesis_{\mathrm{WL}}(\sigma, h) \triangleq \left\{ (\sigma[\mathtt{x} \mapsto v], h) \;\middle|\; \begin{array}{l} \llparenthesis \mathtt{e} \rrparenthesis^{\sigma} = v \land v \in \mathrm{V} \\ \land\, \mathtt{x} \in \mathit{dom}(\sigma) \end{array} \right\}$$
$$\cup \left\{ \lightning \;\middle|\; \mathtt{x} \notin \mathit{dom}(\sigma) \lor \llparenthesis \mathtt{e} \rrparenthesis^{\sigma} \;\; \text{undefined} \right\}$$

Given the parametric WL program states $\mathrm{WPS{\scriptstyle TATE}}\langle \mathrm{V} \rangle$ (Def. 25), let $\mathrm{S} \triangleq \mathrm{WPS{\scriptstyle TATE}}\langle \mathrm{V} \rangle \times \mathrm{C}$ denote a generic extension of WL program states, comprising parametric WL program states, as well as library program states captured by C. Let $\mathrm{O} \supseteq \mathrm{P{\scriptstyle RIM}WO{\scriptstyle P}}$ denote a generic extension of primitive WL operations (Def. 27) and let

$$\mathrm{WO{\scriptstyle P}}\langle \mathrm{O} \rangle^{\dagger} \triangleq \mathrm{WO{\scriptstyle P}} \left\langle \mathrm{O} \uplus \left\{ \mathtt{rem(x)} \;\middle|\; \mathtt{x} \in \mathrm{PV{\scriptstyle AR}} \right\} \right\rangle$$

The WL *parametric small-step transition system*,

$$\rightarrow_{(.)} \colon \big(\mathrm{O} \rightarrow \mathrm{S} \rightarrow \mathcal{P}(\mathrm{S})\big) \rightarrow \mathcal{P}\Big((\mathrm{WO{\scriptstyle P}}\langle \mathrm{O} \rangle^{\dagger} \times \mathrm{S}) \times (\mathrm{WO{\scriptstyle P}}\langle \mathrm{O} \rangle^{\dagger} \times \mathrm{S}) \Big)$$

is defined by the rules in Fig. 3.1 for $o \in \mathrm{O}$, $f : \mathrm{O} \rightarrow \mathrm{S} \rightarrow \mathcal{P}(\mathrm{S})$, $c \in \mathrm{C}$, and $s \in \mathrm{S}$.
The WL *parametric big-step transition system*, $\rightsquigarrow_{(.)} \colon \big(\mathrm{O} \rightarrow \mathrm{S} \rightarrow \mathcal{P}(\mathrm{S})\big) \rightarrow \mathcal{P}\Big((\mathrm{WO{\scriptstyle P}}\langle \mathrm{O} \rangle^{\dagger} \times \mathrm{S}) \times \mathrm{S} \Big)$, is defined as follows, for $f : \mathrm{O} \rightarrow \mathrm{S} \rightarrow \mathcal{P}(\mathrm{S})$, $c \in \mathrm{C}$, and $s \in \mathrm{S}$:

$$\frac{}{(\mathtt{skip}, s) \rightsquigarrow_f s} \qquad \frac{(o, s) \rightarrow_f (o', (\lightning, c))}{(o, s) \rightsquigarrow_f (\lightning, c)}$$

$$\frac{(o, s) \rightarrow_f (o'', s'') \quad (o'', s'') \rightsquigarrow_f s'}{(o, s) \rightsquigarrow_f s'}$$

The WL *parametric semantics function*,

$$\mathsf{sem}(.) : \big(\mathrm{O} \rightarrow \mathrm{S} \rightarrow \mathcal{P}(\mathrm{S})\big) \rightarrow \mathrm{WO{\scriptstyle P}}\langle \mathrm{O} \rangle \rightarrow \mathrm{S} \rightarrow \mathcal{P}(\mathrm{S})$$

is defined as follows, for all $f : \mathrm{O} \rightarrow \mathrm{S} \rightarrow \mathcal{P}(\mathrm{S})$, $o \in \mathrm{WO{\scriptstyle P}}\langle \mathrm{O} \rangle$ and $s \in \mathrm{S}$:

$$\mathsf{sem}(f)(o)(s) \triangleq \{ s' \mid (o, s) \rightsquigarrow_f s' \}$$

Example 4 ($\mathrm{WL}_{\mathbb{L}}$ semantics)**.** Given the WL primitive semantics function $\langle\!|.|\!\rangle_{\text{WL}}$ (Def. 29) and the list semantics function $\langle\!|.|\!\rangle_{\mathbb{L}}$ (Def. 42), the WL$_{\mathbb{L}}$ *primitive semantics function*, $\langle\!|.|\!\rangle_{\text{WL}\mathbb{L}} : \textsc{PrimWOp}_{\mathbb{L}} \to \textsc{WPState}_{\mathbb{L}} \to \mathcal{P}(\textsc{WPState}_{\mathbb{L}})$, is defined as follows, for all $\mathtt{C}_{\text{WL}} \in \textsc{PrimWOp}$ (Def. 27), $\mathtt{C}_{\mathbb{L}} \in \textsc{Op}_{\mathbb{L}}$ (Def. 16), and $(s, \text{h}) \in \textsc{WPState}_{\mathbb{L}}$ (Example 2):

$$\langle\!|\mathtt{C}_{\text{WL}}|\!\rangle_{\text{WL}\mathbb{L}}(s, \text{h}) \triangleq \{(s', \text{h}) \mid s' \in \langle\!|\mathtt{C}_{\text{WL}}|\!\rangle_{\text{WL}}(s)\}$$
$$\langle\!|\mathtt{C}_{\mathbb{L}}|\!\rangle_{\text{WL}\mathbb{L}}(s, \text{h}) \triangleq \langle\!|\mathtt{C}_{\mathbb{L}}|\!\rangle_{\mathbb{L}}(s, \text{h})$$

Given the WL program states $\textsc{WPState}_{\mathbb{L}}$ (Example 2), the WL$_{\mathbb{L}}$ operations $\textsc{WOp}_{\mathbb{L}}$ (Example 3) and the WL parametric semantics function $\mathsf{sem}(.)$ (Def. 29), the WL$_{\mathbb{L}}$ *semantics function*, $[\![.]\!]_{\text{WL}\mathbb{L}} : \textsc{WOp}_{\mathbb{L}} \to \textsc{WPState}_{\mathbb{L}} \to \mathcal{P}(\textsc{WPState}_{\mathbb{L}})$, is defined as follows:

$$[\![.]\!]_{\text{WL}\mathbb{L}} \triangleq \mathsf{sem}(\langle\!|.|\!\rangle_{\text{WL}\mathbb{L}})$$

Logical values We assume a set of PLogic logical values denoting those values that may be associated with logical variables. In PLogic$_{\mathbb{A}}$ we extend the set of PLogic logical values with those of library $\mathbb{A}$ in $\textsc{LVal}_{\mathbb{A}}$ (Par. 7).

PLogic Parameter

Parameter 14 (PLogic logical values)**.** Assume a set of PLogic *logical values* $v \in \textsc{PLLVal}$.

Definition 30 (PLogic$_{\mathbb{A}}$ logical values)**.** Given the set of PLogic logical values PLLVal (Par. 14) and the set of library logical values $\textsc{LVal}_{\mathbb{A}}$ (Par. 7), the set of PLogic$_{\mathbb{A}}$ *logical values* is $v \in \textsc{PLLVal}_{\mathbb{A}} \triangleq \textsc{PLLVal} \cup \textsc{LVal}_{\mathbb{A}}$.

WLogic logical values For WLogic, the set of logical values is the extension of WL program values with lists.

WLogic Instance (Parameter 14)

Definition 31 (WLogic logical values)**.** Given the set of WL program values WPVal (Def. 23), the set of WLogic logical values is

$\textsc{WLVal} \triangleq \textsc{WPVal} \cup \textsc{List}\langle\textsc{WLVal}\rangle$.

Example 5 ($\textsc{WLogic}_{\mathbb{L}}$ logical values)**.** Given the set of list logical values $\textsc{LVal}_{\mathbb{L}}$ (Def. 17) and the set of WLogic logical values WLVal (Def. 31), the set of $\textsc{WLogic}_{\mathbb{L}}$ logical values is $\textsc{WLVal}_{\mathbb{L}} \triangleq \textsc{WLVal} \cup \textsc{LVal}_{\mathbb{L}}$.

Logical states Recall that the interaction between the programs written in PL and a library $\mathbb{A}$ are carried out via program variables. As such, we stipulated that PL program states embody a variable store parametric in the choice of program values (see Par. 11) in order to allow for the extension of the set of program values with those of the library upon integration. Analogously, we require that a PLogic logical state embody a high-level representation of the PL variable store parametric in the choice of program values $\mathrm{V} \supseteq \textsc{PLPVal}$, denoting a generic extension of PL program values. As before, this is to allow for the extension of program values (i.e. the values associated with program variables) with those of the library to enable client reasoning for library $\mathbb{A}$. We further require the logical states of PLogic to be modelled as a *parametric partial commutative monoid* of the form $(\textsc{LState}\langle\mathrm{V}\rangle, \circ_{\langle\mathrm{V}\rangle}, \textsc{Unit}\langle\mathrm{V}\rangle)$, where $\textsc{LState}\langle\mathrm{V}\rangle$ denotes the parametric set of states, the $\circ_{\langle\mathrm{V}\rangle}$ denotes the *parametric state composition*, and $\textsc{Unit}\langle\mathrm{V}\rangle$ denotes the *parametric unit set*.

In order to reason about the $\mathbb{A}$ operations, in $\textsc{PLogic}_{\mathbb{A}}$ we instantiate the PLogic states with $\textsc{PLogic}_{\mathbb{A}}$ program values as $\textsc{LState}\langle\textsc{PLPVal}_{\mathbb{A}}\rangle$, and further extend the states to incorporate abstract heaps. That is, we combine the partial commutative monoid of PLogic states with the separation algebra of abstract heaps (Def. 14) and define a program state to be a pair, $(\mathsf{s}, \mathbf{h})$, comprising a PL state $\mathsf{s} \in \textsc{LState}\langle\textsc{PLPVal}_{\mathbb{A}}\rangle$ and an abstract heap $\mathbf{h} \in \textsc{LHeap}_{\mathbb{A}}$.

PLogic Parameter

Parameter 15 (PLogic partial commutative monoid)**.** Given the set of PL program values (Par. 10) and a generic extension of PL program values $\mathrm{V} \supseteq \textsc{PLPVal}$, assume a set of *parametric* PLogic *logical states* $\textsc{LState}\langle\mathrm{V}\rangle$. Assume a *parametric partial com-*

mutative monoid, $\text{PLPCM}\langle V\rangle \triangleq (\text{LS{\scriptsize TATE}}\langle V\rangle, \circ_{\langle V\rangle}, \text{U{\scriptsize NIT}}\langle V\rangle)$, with $\circ_{\langle V\rangle} : \text{LS{\scriptsize TATE}}\langle V\rangle \times \text{LS{\scriptsize TATE}}\langle V\rangle \rightharpoonup \text{LS{\scriptsize TATE}}\langle V\rangle$ and $\text{U{\scriptsize NIT}}\langle V\rangle \in \mathcal{P}(\text{LS{\scriptsize TATE}}\langle V\rangle)$.
Assume that the PLogic *partial commutative monoid* is:

$$(\text{PLLS{\scriptsize TATE}}, \circ, \text{PLU{\scriptsize NIT}}) \triangleq \text{PLPCM}\langle \text{PLPV{\scriptsize AL}}\rangle$$

Definition 32 (PLogic$_{\mathbb{A}}$ partial commutative monoid)**.** Given the set of PL$_{\mathbb{A}}$ program values $\text{PLPV{\scriptsize AL}}_{\mathbb{A}}$ (Def. 22) and the separation algebra of abstract logical heaps $(\text{LH{\scriptsize EAP}}_{\mathbb{A}}, \bullet, \mathbf{0})$, the set of PLogic$_{\mathbb{A}}$ *logical states* is: $\mathsf{w} \in \text{PLLS{\scriptsize TATE}}_{\mathbb{A}} \triangleq \text{LS{\scriptsize TATE}}\langle \text{PLPV{\scriptsize AL}}_{\mathbb{A}}\rangle \times \text{LH{\scriptsize EAP}}_{\mathbb{A}}$. The PLogic$_{\mathbb{A}}$ *logical state composition*, $+ : \text{PLLS{\scriptsize TATE}}_{\mathbb{A}} \times \text{PLLS{\scriptsize TATE}}_{\mathbb{A}} \rightharpoonup \text{PLLS{\scriptsize TATE}}_{\mathbb{A}}$, is defined component-wise as $+ \triangleq (\circ_{\langle \text{PLPV{\scriptsize AL}}_{\mathbb{A}}\rangle}, \bullet)$ and is not defined if composition on either component is undefined. The PLogic$_{\mathbb{A}}$ *unit set* is $\text{PLU{\scriptsize NIT}}_{\mathbb{A}} \triangleq \{(\mathsf{s}, \mathbf{0}) \mid \mathsf{s} \in \text{U{\scriptsize NIT}}\langle \text{PLPV{\scriptsize AL}}_{\mathbb{A}}\rangle\}$. The PLogic$_{\mathbb{A}}$ *partial commutative monoid* is:

$$\text{PLPCM}_{\mathbb{A}} \triangleq (\text{PLLS{\scriptsize TATE}}_{\mathbb{A}}, +, \text{PLU{\scriptsize NIT}}_{\mathbb{A}})$$

WLogic logical states The WLogic logical states are those of WL program states excluding the fault state. That is, a WLogic logical state is a pair comprising a variable stack and a heap. WLogic state composition is defined component-wise as $(\uplus, \uplus)$ where $\uplus$ denotes the standard disjoint function union. The empty WLogic state is defined as a pair comprising two functions with empty domains. As before, variable stacks are parametric in the choice of program values whereas heaps are not. That is, variable stacks are the sole point of interaction between the client and the library and hence we do not allow the underlying heap (memory) to have pointers into the library. Instead, we allow the values of program variables to extend beyond WL program values to include library values.

WLogic Instance (Parameter 15)

Definition 33 (WLogic partial commutative monoid)**.** Let $\mathrm{V} \supseteq \textsc{WPVal}$ denote a generic extension of WL program values (Def. 23). Given the set of WL parametric variable stacks $\textsc{Stack}\langle \mathrm{V}\rangle$ and the set of WL heaps $\textsc{Heap}$ (Def. 25), the set of *parametric* WLogic *logical states* is $\mathsf{s} \in \textsc{WLState}\langle \mathrm{V}\rangle \triangleq \textsc{Stack}\langle \mathrm{V}\rangle \times \textsc{Heap}$.
The *parametric* WLogic *state composition*, $\circ_{\langle \mathrm{V}\rangle} : \textsc{WLState}\langle \mathrm{V}\rangle \times \textsc{WLState}\langle \mathrm{V}\rangle \rightharpoonup \textsc{WLState}\langle \mathrm{V}\rangle$, is defined component-wise as $\circ_{\langle \mathrm{V}\rangle} \triangleq (\uplus, \uplus)$, where $\uplus$ denotes the standard disjoint function union, and $\circ_{\langle \mathrm{V}\rangle}$ is not defined when composition on either component is undefined. The *parametric* WLogic *unit set*, $\textsc{WUnit}\langle \mathrm{V}\rangle \in \mathcal{P}(\textsc{WLState}\langle \mathrm{V}\rangle)$, is $\textsc{WUnit}\langle \mathrm{V}\rangle \triangleq \{(\mathbf{0}, \mathbf{0})\}$, where $\mathbf{0}$ denotes a function with an empty domain. The *parametric* WLogic *partial commutative monoid* is $\mathrm{WPCM}\langle \mathrm{V}\rangle \triangleq (\textsc{WLState}\langle \mathrm{V}\rangle, \circ_{\langle \mathrm{V}\rangle}, \textsc{WUnit}\langle \mathrm{V}\rangle)$.
The WLogic *partial commutative monoid* is $(\textsc{WLState}, \circ_{\mathrm{W}}, \textsc{WUnit}) \triangleq \mathrm{WPCM}\langle \textsc{WPVal}\rangle$.

Example 6 (WLogic$_{\mathbb{L}}$ logical states)**.** Given the WL$_{\mathbb{L}}$ program values $\textsc{WPVal}_{\mathbb{L}}$ (Example 1), the list logical heaps $\textsc{LHeap}_{\mathbb{L}}$ (Def. 14) and the parametric WLogic program states $\textsc{WLState}\langle . \rangle$ (Def. 25), the WLogic$_{\mathbb{L}}$ logical states are:

$$\textsc{WLState}_{\mathbb{L}} \triangleq \textsc{WLState}\langle \textsc{WPVal}_{\mathbb{L}}\rangle \times \textsc{LHeap}_{\mathbb{L}}$$

Given the parametric WLogic composition $\circ_{\langle . \rangle}$ (Def. 25) and the list logical heap composition $\bullet_{\mathbb{L}}$ (Def. 14), the WLogic$_{\mathbb{L}}$ logical state composition is defined component-wise as $+ \triangleq (\circ_{\langle \textsc{WPVal}_{\mathbb{L}}\rangle}, \bullet_{\mathbb{L}})$, and is not defined if the composition on either component is undefined.

Given the parametric WLogic unit set $\textsc{WUnit}\langle . \rangle$ (Def. 25) and the list unit element $\mathbf{0}_{\mathbb{L}}$ (Def. 14), the WLogic$_{\mathbb{L}}$ unit set is:

$$\textsc{WUnit}_{\mathbb{L}} \triangleq \{(\mathsf{s}, \mathbf{0}_{\mathbb{L}}) \mid \mathsf{s} \in \textsc{WUnit}\langle \textsc{WPVal}_{\mathbb{L}}\rangle\}$$

The WLogic$_{\mathbb{L}}$ partial commutative monoid is: $(\textsc{WLState}_{\mathbb{L}}, +, \textsc{WUnit}_{\mathbb{L}})$.

Logical expressions We require PLogic to define a set of logical expressions that includes logical variables and values. We assume the PLogic logical expressions to be defined by an inductive grammar comprising a set of *primitive logical expressions* in PrimPLLExp, as well as a set of *composite logical expressions* in $\text{LExp}\langle \text{E} \rangle$, parametric in the choice of primitive expressions used as the "building blocks" of logical expressions. We further require PLogic to define an evaluation function for logical expressions. Logical expressions are often evaluated with respect to a *logical environment* (Def. 3). However, in order to evaluate the logical expressions, PLogic may need additional information besides that provided by the logical environment. For instance, as we demonstrate later in §5, the logical expressions of the JavaScript program logic in [21] are evaluated with respect to a logical environment *and* a scope chain. To capture this, we require PLogic to define an *evaluation environment* which must comprise a logical environment, and may include additional information required for expression evaluation. In $\text{PLogic}_{\mathbb{A}}$ we extend the primitive logical expressions of PLogic with those of the library (Par. 8), and instantiate the parametric composite expressions with the extended primitive expressions. Accordingly, we lift the expression evaluation function of PLogic to $\text{PLogic}_{\mathbb{A}}$ expressions.

PLogic Parameter

Parameter 16 (PLogic logical expressions)**.** Let $\text{L} \supseteq \text{PLLVal}$ denote a generic extension of PLogic logical values (Par. 7).
Assume a set of *primitive logical expressions*, $e \in \text{PrimPLLExp}$.
Let $\text{E} \supseteq \text{PrimPLLExp}$ denote a generic extension of PLogic primitive logical expressions.
Given the set of logical variables LVar (Def. 2), assume a set of *parametric logical expressions* $\text{LExp}\langle \text{E} \rangle$ such that $\text{LVar} \cup \text{E} \subseteq \text{LExp}\langle \text{E} \rangle$.
Assume that the set of PLogic *logical expressions* is:

$$E \in \text{PLLExp} \triangleq \text{LExp}\langle \text{PrimPLLExp} \rangle$$

Given the set of logical environments $\text{Env}\langle \text{L} \rangle$ (Def. 3), assume a *parametric evaluation environment*, $\text{Env}\langle \text{L} \rangle \triangleq \text{LEnv}\langle \text{L} \rangle \times \text{B}$, com-

prising a parametric logical environment, as well as additional information captured by B.
Assume a *primitive logical expression evaluation function*, $(\!|.|\!)^{(.)}_{\text{PL}}$: $\textsc{PrimPLLExp} \times \textsc{Env}\,\langle\text{PLLVal}\rangle \rightharpoonup \text{PLLVal}$.
Assume a *generic evaluation function*,

$$\mathsf{eval}(.) : (\text{E} \times \textsc{Env}\,\langle\text{L}\rangle \rightharpoonup \text{L}) \to (\textsc{LExp}\,\langle\text{E}\rangle \times \textsc{Env}\,\langle\text{L}\rangle \rightharpoonup \text{L})$$

that given an evaluation function for primitive expressions in E, it produces an evaluation function for expressions in $\textsc{LExp}\,\langle\text{E}\rangle$, provided that for all $e \in \text{E}$, $\epsilon{=}(\Gamma, b) \in \textsc{Env}\,\langle\text{L}\rangle$, $\textsc{x} \in \textsc{LVar}$ and $f_p \in \text{E} \times \textsc{Env}\,\langle\text{L}\rangle \rightharpoonup \text{L}$, it satisfies the following where $f \triangleq \mathsf{eval}(f_p)$:

$$f(e, \epsilon) = f_p(e, \epsilon) \qquad\qquad (\!|\textsc{x}|\!)^{(\Gamma,b)}_{\text{PL}} = \Gamma(\textsc{x})$$

Assume that the PLogic *logical expression evaluation function*, $\|.\|^{(.)}_{\text{PL}}$: $\textsc{PLLExp} \times \textsc{Env}\,\langle\text{PLLVal}\rangle \rightharpoonup \text{PLLVal}$, is defined as follows:

$$\|.\|^{(.)}_{\text{PL}} \triangleq \mathsf{eval}((\!|.|\!)^{(.)}_{\text{PL}})$$

Definition 34 (PLogic$_{\mathbb{A}}$ logical expressions)**.** Given the PLogic primitive logical expressions PrimPLLExp (Par. 16) and the library logical expressions $\textsc{LExp}_{\mathbb{A}}$ (Par. 8), the set of PLogic$_{\mathbb{A}}$ *primitive logical expressions* is $e \in \textsc{PrimPLLExp}_{\mathbb{A}} \triangleq \textsc{PrimPLLExp} \cup \textsc{LExp}_{\mathbb{A}}$.

The set of PLogic$_{\mathbb{A}}$ *logical expressions* is defined as $E \in \textsc{PLLExp}_{\mathbb{A}} \triangleq \textsc{LExp}\,\langle\textsc{PrimPLLExp}_{\mathbb{A}}\rangle$.

Given the library expression evaluation function $(\!|.|\!)^{(.)}_{\mathbb{A}}$ (Par. 8), the PLogic parametric evaluation environment $\textsc{Env}\,\langle\text{L}\rangle$ and the primitive expression evaluation function $(\!|.|\!)^{(.)}_{\text{PL}}$ (Par. 16), the PLogic$_{\mathbb{A}}$ *primitive expression evaluation function*, $(\!|.|\!)^{(.)}_{\text{PL}\mathbb{A}}$: $\textsc{PrimPLLExp}_{\mathbb{A}} \times \textsc{Env}\,\langle\textsc{PLLVal}_{\mathbb{A}}\rangle \rightharpoonup \textsc{PLLVal}_{\mathbb{A}}$, is defined inductively over the structure of $\textsc{PrimPLLExp}_{\mathbb{A}}$ as follows, for $e_{\mathbb{A}} \in \textsc{LExp}_{\mathbb{A}}$, $e_{\text{PL}} \in \textsc{PrimPLLExp}$ and $(\Gamma, b) \in \textsc{Env}\,\langle\textsc{PLLVal}_{\mathbb{A}}\rangle$, where $\Gamma_p \triangleq \Gamma{\downarrow}_{\textsc{PLLVal}}$ and $\Gamma_a \triangleq \Gamma{\downarrow}_{\textsc{LVal}_{\mathbb{A}}}$ (see Def. 3):

$$(\!|e_{\text{PL}}|\!)^{(\Gamma,b)}_{\text{PL}\mathbb{A}} \triangleq (\!|e_{\text{PL}}|\!)^{(\Gamma_p,b)}_{\text{PL}} \qquad\qquad (\!|e_{\mathbb{A}}|\!)^{(\Gamma,b)}_{\text{PL}\mathbb{A}} \triangleq (\!|e_{\mathbb{A}}|\!)^{\Gamma_a}_{\mathbb{A}}$$

Given the generic expression evaluation function $\mathsf{eval}(.)$ (Par. 16), the *logical expression evaluation function for* $\textsc{PLogic}_{\mathbb{A}}$, $\|.\|^{(.)}_{\mathrm{PL}\mathbb{A}} : \textsc{PLLExp}_{\mathbb{A}} \times \textsc{Env}\langle\textsc{PLLVal}_{\mathbb{A}}\rangle \rightharpoonup \textsc{PLLVal}_{\mathbb{A}}$, is defined as follows:

$$\|.\|^{(.)}_{\mathrm{PL}\mathbb{A}} \triangleq \mathsf{eval}(\llparenthesis . \rrparenthesis^{(.)}_{\mathrm{PL}\mathbb{A}})$$

WLogic logical expressions The logical expressions of WLogic comprise logical variables and values, the empty list $[\,]$, list cons written $E_1 : E_2$ and list concatenation written $E_1{+\!\!+}E_2$.

WLogic Instance (Parameter 16)

Definition 35 (WLogic logical expressions)**.** Given the set of logical values $\textsc{WLVal}$ (Def. 31), the set of WLogic *primitive logical expressions* is $e \in \textsc{PrimWLExp} \triangleq \textsc{WLVal}$.
Let $\mathrm{PE} \supseteq \textsc{PrimWLExp}$ denote a generic extension of WLogic primitive logical expressions.
Given the set of logical variables $\textsc{LVar}$ (Def. 2), the set of *parametric logical expressions* $\textsc{LExp}\langle\mathrm{PE}\rangle$ is defined by the following grammar, where $e \in \mathrm{PE}$ and $\mathrm{x} \in \textsc{LVar}$:

$$\textsc{WLExp}\langle\mathrm{E}\rangle \ni E ::= e \mid \mathrm{x} \mid E_1{:}E_2 \mid E_1{+\!\!+}E_2$$

The set of WLogic *logical expressions* is $\textsc{WLExp} \triangleq \textsc{WLExp}\langle\textsc{PrimWLExp}\rangle$.
Given the set of logical values $\textsc{WLVal}$ (Def. 31) and the set of logical environments $\textsc{LEnv}\langle\textsc{WLVal}\rangle$ (Def. 3), the set of *parametric* WLogic *evaluation environments* is $\Gamma \in \textsc{LEnv}\langle\textsc{WLVal}\rangle$.
Given an evaluation environment $\Gamma \in \textsc{LEnv}\langle\textsc{WLVal}\rangle$, the *primitive logical expression evaluation function*, $\llparenthesis . \rrparenthesis^{(.)}_{\mathrm{WL}} : \textsc{PrimWLExp} \times \textsc{LEnv}\langle\textsc{WLVal}\rangle \rightharpoonup \textsc{WLVal}$, is defined inductively over the structure of primitive expressions as:

$$\llparenthesis v \rrparenthesis^{\Gamma}_{\mathrm{WL}} \triangleq v$$

Given an extension of WLogic logical values $\mathrm{L} \supseteq \textsc{WLVal}$ and an extension of WLogic primitive logical expressions $\mathrm{PE} \supseteq \textsc{PrimWLExp}$, the *parametric evaluation function*,

$$\mathsf{eval}(.) : (\text{PE} \times \text{LE}\textsc{nv}\,\langle \text{L}\rangle \rightharpoonup \text{L}) \to (\text{LE}\textsc{xp}\,\langle \text{PE}\rangle \times \text{LE}\textsc{nv}\,\langle \text{L}\rangle \rightharpoonup \text{L})$$

is defined as follows, for all $e \in \text{PE}$, $\Gamma \in \text{LE}\textsc{nv}\,\langle \text{L}\rangle$ and $f_p : \text{PE} \times \text{LE}\textsc{nv}\,\langle \text{L}\rangle \rightharpoonup \text{L}$ where $f \triangleq \mathsf{eval}(f_p)$:

$$f(e, \Gamma) \triangleq f_p(e, \Gamma) \qquad\qquad f(\text{x}, \Gamma) \triangleq \Gamma(\text{x})$$

$$f(E_1 : E_2, \Gamma) \triangleq \begin{cases} v : L & \text{if } f(E_1, \Gamma) = v \text{ and } f(E_2, \Gamma) = L \\ & \quad \text{and } L \in \textsc{List}\langle \text{L}\rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f(E_1 {+\!\!+} E_2, \Gamma) \triangleq \begin{cases} L_1 {+\!\!+} L_2 & \text{if } f(E_1, \Gamma) = L_1 \text{ and } f(E_2, \Gamma) = L_2 \\ & \quad \text{and } L_1, L_2 \in \textsc{List}\langle \text{L}\rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

The WL<span style="font-variant:small-caps">ogic</span> *logical expression evaluation function*, $\|.\|^{(.)}_{\text{WL}} : (\text{WLE}\textsc{xp} \times \text{LE}\textsc{nv}\,\langle \text{WLV}\textsc{al}\rangle) \rightharpoonup \text{WLV}\textsc{al}$, is defined as follows:

$$\|.\|^{(.)}_{\text{WL}} \triangleq \mathsf{eval}(\llparenthesis . \rrparenthesis^{(.)}_{\text{WL}})$$

Example 7 (WLogic$_{\mathbb{L}}$ logical expressions)**.** Given the logical expressions for lists LExp$_{\mathbb{L}}$ (Def. 18) and the WLogic primitive logical expressions PrimWLExp (Def. 35), the WLogic$_{\mathbb{L}}$ primitive expressions are:

$$\textsc{PrimWLExp}_{\mathbb{L}} \triangleq \textsc{PrimWLExp} \cup \textsc{LExp}_{\mathbb{L}}$$

Given the WLogic parametric expressions WLExp $\langle . \rangle$ (Def. 35), the WLogic$_{\mathbb{L}}$ logical expressions are WLExp$_{\mathbb{L}} \triangleq$ WLExp $\langle \text{PrimWLExp}_{\mathbb{L}} \rangle$.

Given the WLogic$_{\mathbb{L}}$ logical values WLVal$_{\mathbb{L}}$ (Example 5), the set of logical environments LEnv $\langle \text{WLVal}_{\mathbb{L}} \rangle$ (Def. 3), the WLogic primitive evaluation function $\llparenthesis . \rrparenthesis^{(.)}_{\text{WL}}$ (Par. 35) and the list evaluation function $\llparenthesis . \rrparenthesis^{(.)}_{\mathbb{L}}$ (Par. 18), the WLogic *primitive evaluation function*, $\llparenthesis . \rrparenthesis^{(.)}_{\text{WL}\mathbb{L}} : \textsc{PrimWLExp}_{\mathbb{L}} \times \text{LEnv}\,\langle \text{WLVal}_{\mathbb{L}} \rangle \rightharpoonup \text{WLVal}_{\mathbb{L}}$, is defined follows, for all $e_{\mathbb{L}} \in \text{LExp}_{\mathbb{L}}$ (Def. 18),

$e_{\text{WL}} \in \textsc{PrimWLExp}$ (Def. 35) and $\Gamma \in \text{LEnv}\langle \text{WLVal}_{\mathbb{L}} \rangle$, where $\Gamma_w \triangleq \Gamma{\downarrow_{\text{WLVal}}}$ and $\Gamma_l \triangleq \Gamma{\downarrow_{\text{LVal}_{\mathbb{L}}}}$ (see Def. 3):

$$(\!|e_{\text{WL}}|\!)^{\Gamma}_{\text{WL}\mathbb{L}} \triangleq (\!|e_{\text{WL}}|\!)^{\Gamma_w}_{\text{WL}} \qquad\qquad (\!|e_{\mathbb{L}}|\!)^{\Gamma}_{\text{WL}\mathbb{L}} \triangleq (\!|e_{\mathbb{L}}|\!)^{\Gamma_l}_{\mathbb{L}}$$

Given the parametric WLogic evaluation function $\mathsf{eval}(.)$ (Par. 16), the $\text{WLogic}_{\mathbb{L}}$ *evaluation function,* $\|.\|^{(.)}_{\text{WL}\mathbb{L}}$:$\text{WLExp}_{\mathbb{L}} \times \text{LEnv}\langle \text{WLVal}_{\mathbb{L}} \rangle \rightharpoonup \text{WLVal}_{\mathbb{L}}$, is defined as follows:

$$\|.\|^{(.)}_{\text{WL}\mathbb{L}} \triangleq \mathsf{eval}((\!|.|\!)^{(.)}_{\text{WL}\mathbb{L}})$$

Assertions and their semantics We assume that the assertions of PLogic, $P \in \text{PLAst}$, include: i) standard classical assertions (e.g. $P \Rightarrow Q$); ii) standard boolean assertions (e.g. $E_1{=}E_2$); iii) standard SL assertions (e.g. $P * Q$); and iv) assertions to describe the underlying PL-specific variable store. We assume that the PLogic assertions are interpreted via a satisfiability relation and that the classical, boolean and SL assertions have their standard semantics. We require that the PLogic assertion language be defined by an inductive grammar comprising primitive and composite assertions. In $\text{PLogic}_{\mathbb{A}}$, we extend the primitive assertions of PLogic with those of SSL assertions (Def. 19), and accordingly lift the satisfiability relation to $\text{PLogic}_{\mathbb{A}}$ assertions.

PLogic Parameter

Parameter 17 (PLogic assertions and their semantics)**.** Assume a set of PLogic *primitive assertions* $p \in \text{PrimPLAst}$.
Let $\text{A} \supseteq \text{PrimPLAst}$ denote an extension of primitive PLogic assertions. Let $\text{V} \supseteq \text{PLPVal}$ denote an extension of PL program values (Par. 10) and $\text{L} \supseteq \text{PLLVal}$ denote an extension of PLogic logical values (Par. 14). Let $\text{W} \triangleq (\text{LState}\langle \text{V} \rangle \times \text{D})$ denote an extension of parametric PLogic logical states (Par. 15), comprising PLogic logical states in $\text{LState}\langle \text{V} \rangle$, as well as library logical states captured by D. Let $\text{PE} \supseteq \text{PrimPLLExp}$ denote an extension of PLogic logical expressions and $\text{E} \triangleq \text{LExp}\langle \text{PE} \rangle$ (Par. 16).
Given the parametric evaluation environment $\text{Env}\langle . \rangle$ (Par. 16), assume a *primitive satisfaction relation,* $\models_p \subseteq (\text{Env}\langle \text{L} \rangle \times \text{LState}\langle \text{V} \rangle) \times$

PrimPLAst.
Assume a set of *parametric assertions*, $\textsc{Ast}\langle \mathrm{A}, \mathrm{E}\rangle$, defined by the following inductive grammar where $a \in \mathrm{A}$, $E_1, E_2 \in \mathrm{E}$, $\mathtt{x}_1 \cdots \mathtt{x}_n \in$ $\textsc{PVar}$, $\mathrm{x}, \mathrm{v}_1 \cdots \mathrm{v}_n \in \textsc{LVar}$, and $P, Q, \in \textsc{Ast}\langle \mathrm{A}, \mathrm{E}\rangle$:

$$\begin{aligned}\textsc{Ast}\langle \mathrm{A}, \mathrm{E}\rangle \ni P, Q ::= &\ \cdots \mid a \mid \mathsf{false} \mid P \Rightarrow Q \mid \exists \mathrm{x}.\ P\\ &\mid E_1 \ominus E_2 \ \text{ where } \ominus \in \{=, <\}\\ &\mid \mathsf{emp} \mid P * Q \mid P \mathrel{-\!\!*} Q\\ &\mid \mathsf{vars}(\overline{\mathtt{x}_i : \mathrm{v}_i}^{i=1\ldots n})\end{aligned}$$

Given the set of PLogic logical expressions (Par. 16), assume that the set of PLogic *assertions* is:

$$\textsc{PLAst} \triangleq \textsc{Ast}\langle \textsc{PrimPLAst}, \textsc{PLLExp}\rangle$$

Assume a parametric ordering on PLogic states, $\preccurlyeq_{\langle \mathrm{V}\rangle} \subseteq$ $\textsc{LState}\langle \mathrm{V}\rangle \times \textsc{LState}\langle \mathrm{V}\rangle$.
Given the parametric evaluation environment $\textsc{Env}\langle . \rangle$ (Par. 16), assume a *generic satisfaction relation*,

$$\mathsf{sat}(.) : \begin{pmatrix} \ \ \mathcal{P}(\mathrm{W}{\times}\mathrm{W}) \\ \times (\mathrm{W}{\times}\mathrm{W} \rightharpoonup \mathrm{W}) \\ \times \mathcal{P}(\mathrm{W}) \\ \times \mathcal{P}((\textsc{Env}\langle \mathrm{L}\rangle \times \mathrm{W}){\times}\mathrm{A}) \\ \times (\textsc{Env}\langle \mathrm{L}\rangle \times \mathrm{E} \rightharpoonup \mathrm{L}) \end{pmatrix} \rightarrow \mathcal{P}((\textsc{Env}\langle \mathrm{L}\rangle \times \mathrm{W}){\times}\textsc{Ast}\langle \mathrm{A}, \mathrm{E}\rangle)$$

that given an ordering on states of W, a composition operator on the states in W, a unit set for W, a satisfaction relation for primitive assertions in A, and an expression evaluation function for expressions in E, it produces a satisfaction relation for assertions in $\textsc{Ast}\langle \mathrm{A}, \mathrm{E}\rangle$, with the proviso that for all $R_{ord} \in \mathrm{W} \times \mathrm{W}$, $\oplus : (\mathrm{W}{\times}\mathrm{W} \rightharpoonup \mathrm{W})$, $U \in \mathcal{P}(\mathrm{W})$, $R_p \in (\textsc{Env}\langle \mathrm{L}\rangle \times \mathrm{W}) \times \mathrm{A}$, $f \in \textsc{Env}\langle \mathrm{L}\rangle \times \mathrm{E} \rightharpoonup \mathrm{L}$, $\epsilon{=}(\Gamma, b) \in \textsc{Env}\langle \mathrm{L}\rangle$, $\mathrm{x} \in \textsc{LVar}$, $a \in \mathrm{A}$, $w \in \mathrm{W}$, $P, Q \in \textsc{Ast}\langle \mathrm{A}, \mathrm{E}\rangle$, and $E_1, E_2 \in \mathrm{E}$, it satisfies the following conditions where $R \triangleq \mathsf{sat}(R_{ord}, \oplus, U, R_p, f)$:

$$\begin{array}{lll}
(\epsilon, w)\ R\ a & \text{iff} & (\epsilon, w)\ R_p\ a \\
(\epsilon, w)\ R\ \mathsf{false} & & \text{never} \\
(\epsilon, w)\ R\ (P \Rightarrow Q) & \text{iff} & (\epsilon, w)\ R\ P\ \Rightarrow\ \exists w'.\, (w, w') \in R_{ord} \\
 & & \qquad\qquad \land (\epsilon, w')\ R\ Q \\
((\Gamma, b), w)\ R\ (\exists \mathrm{x}.\ P) & \text{iff} & \exists x.\ ((\Gamma[\mathrm{x} \mapsto x], b), w)\ R\ P \\
(\epsilon, w)\ R\ E_1 \ominus E_2 & \text{iff} & f(\epsilon, E_1) \ominus f(\epsilon, E_2) \\
(\epsilon, w)\ R\ \mathsf{emp} & \text{iff} & w \in U \\
(\epsilon, w)\ R\ (P * Q) & \text{iff} & \exists w_1, w_2.\ w{=}w_1{\oplus}w_2 \land (\epsilon, w_1)\ R\ P \\
 & & \qquad \land (\epsilon, w_2)\ R\ Q \\
(\epsilon, w)\ R\ (P \mathrel{-\!\!*} Q) & \text{iff} & \forall w'.\, (\epsilon, w')\ R\, P \Rightarrow \big(\epsilon, (w{+}w')\big)\ R\ Q
\end{array}$$

Given the ($\text{PLLS}\textsc{tate}, \circ, \text{PLU}\textsc{nit}$) $\text{PL}\textsc{ogic}$ partial commutative monoid (Par. 15), and the $\text{PL}\textsc{ogic}$ expression evaluation function $\|.\|^{(.)}_{\text{PL}}$ (Par. 16), assume that the $\text{PL}\textsc{ogic}$ *satisfaction relation*, $\models_{\text{PL}}$: $\big((\textsc{Env}\,\langle \text{PLLV}\textsc{al}\rangle \times \text{PLLS}\textsc{tate}) \times \text{PLA}\textsc{st}\big)$, is:

$$\models_{\text{PL}} \triangleq \mathsf{sat}(\preccurlyeq_{\langle \text{PLPV}\textsc{al}\rangle}, \circ, \text{PLU}\textsc{nit}, \models_p, \|.\|^{(.)}_{\text{PL}})$$

The semantics of boolean and classical assertions (except $\Rightarrow$) are standard. We generalise the semantics of $\Rightarrow$ to incorporate an ordering relation on the underlying states. For the standard semantics of $\Rightarrow$, this ordering is the identity relation. As we demonstrate shortly, we use this generalisation to incorporate the abstract (de)allocation relation on logical heaps (Def. 15) into the semantics of $\Rightarrow$. This is captured in Lemma 1 below. The $\mathsf{vars}(\overline{\mathtt{x}_i : \mathrm{v}_i}^{i=1\dots n})$ describes a variable store in PL, where variables $\mathtt{x}_1 \cdots \mathtt{x}_n$ have values $\mathrm{v}_1 \cdots \mathrm{v}_n$, respectively. The emp assertion describes an empty logical state denoted by a unit element of the underlying monoid (i.e. $w \in U$). The $P * Q$ describes a state that can be split (via the composition operator $\oplus$ of the underlying monoid) into two substates satisfying P and Q. The $\mathrel{-\!\!*}$ connective is the right adjunct of $*$, i.e. $P * (P \mathrel{-\!\!*} Q) \Rightarrow Q$. Informally, $P \mathrel{-\!\!*} Q$ describes *subtracting* P from Q (i.e. $Q - P$): a state that satisfies $P \mathrel{-\!\!*} Q$ is one that is *missing* P, and when combined with P, it satisfies Q. That is, a state w satisfies $P \mathrel{-\!\!*} Q$ if and only if for any state w' satisfying P, the combined state $w \oplus w'$ satisfies Q. For a binary operator $\ominus$, we write $E_1 \dot{\ominus} E_2$ as a shorthand for $E_1 \ominus E_2 \wedge \mathsf{emp}$.

Definition 36 ($\text{PLogic}_{\mathbb{A}}$ assertions and their semantics)**.** Given the set of PLogic primitive assertions PrimPLAst (Par. 17) and the set of SSL heap assertions $\Theta \in \text{HAst}_{\mathbb{A}}$ (Def. 19), the set of $\text{PLogic}_{\mathbb{A}}$ *primitive assertions* is $\text{PrimPLAst}_{\mathbb{A}} \triangleq \text{PrimPLAst} \cup \text{HAst}_{\mathbb{A}}$.

Given the set of PLogic logical values PLLVal (Par. 14), the parametric evaluation environment $\text{Env}\langle . \rangle$ (Par. 16), the PLogic primitive satisfaction relation $\models_p$ (Par. 17), the set of library logical values $\text{LVal}_{\mathbb{A}}$ (Par. 7), the SSL satisfaction relation $\models_{\mathbb{A}}$ (Def. 20) and the set of $\text{PLogic}_{\mathbb{A}}$ logical values $\text{PLLVal}_{\mathbb{A}}$ (Def. 30), the $\text{PLogic}_{\mathbb{A}}$ *primitive satisfaction relation,* $\models_{p\mathbb{A}} \subseteq \big(\text{Env}\langle \text{PLLVal}_{\mathbb{A}} \rangle \times \text{PLLState}_{\mathbb{A}}\big) \times \text{PrimPLAst}_{\mathbb{A}}$, is defined by induction on the structure of $\text{PLogic}_{\mathbb{A}}$ primitive assertion as follows, where $p \in \text{PrimPLAst}$, $\Theta \in \text{HAst}_{\mathbb{A}}$, $\Gamma_p = \Gamma\downarrow_{\text{PLLVal}}$ and $\Gamma_a = \Gamma\downarrow_{\text{LVal}_{\mathbb{A}}}$ (Def. 3):

$$((\Gamma, b), (\mathsf{s}, \mathbf{h})) \models_{p\mathbb{A}} p \quad \text{iff} \quad ((\Gamma_p, b), \mathsf{s}) \models_p p \wedge \mathbf{h} = \mathbf{0}$$
$$((\Gamma, b), (\mathsf{s}, \mathbf{h})) \models_{p\mathbb{A}} \Theta \quad \text{iff} \quad \mathsf{s} \in \text{PLUnit} \wedge \Gamma_a, \mathbf{h} \models_{\mathbb{A}} \Theta$$

The set of $\text{PLogic}_{\mathbb{A}}$ *assertions* is $P \in \text{PLAst}_{\mathbb{A}} \triangleq \text{Ast}\langle \text{PrimPLAst}_{\mathbb{A}} \rangle$.

Given the parametric $\preccurlyeq_{\langle . \rangle}$ relation on PLogic states (Par. 17) and the abstract allocation relation $\approx$ on abstract heaps (Def. 15), the *ordering on* $\text{PLogic}_{\mathbb{A}}$ *states,* $\preccurlyeq_{\mathbb{A}} \subseteq \text{PLLState}_{\mathbb{A}} \times \text{PLLState}_{\mathbb{A}}$*,* is defined component-wise as $\preccurlyeq_{\mathbb{A}} \triangleq (\preccurlyeq_{\langle \text{PLPVal}_{\mathbb{A}} \rangle}, \approx)$.

Given the PLogic partial commutative monoid $(\text{PLLState}_{\mathbb{A}}, +, \text{PLUnit}_{\mathbb{A}})$ in Def. 32, the primitive satisfaction relation $\models_{p\mathbb{A}}$ and the expression evaluation function $\|.\|^{(.)}_{\text{PL}\mathbb{A}}$ (Def. 34), the *satisfaction relation for* $\text{PLogic}_{\mathbb{A}}$ *assertions*, $\models_{\text{PL}\mathbb{A}}$: $((\text{LEnv}\langle \text{PLLVal}_{\mathbb{A}} \rangle \times \text{B}) \times \text{PLLState}_{\mathbb{A}}) \times \text{PLAst}_{\mathbb{A}}$, is defined as follows:

$$\models_{\text{PL}\mathbb{A}} \triangleq \mathsf{sat}(\preccurlyeq_{\mathbb{A}}, +, \text{PLUnit}_{\mathbb{A}}, \models_{p\mathbb{A}}, \|.\|^{(.)}_{\text{PL}\mathbb{A}})$$

The $\text{PLogic}_{\mathbb{A}}$ assertions are interpreted as sets of $\text{PLogic}_{\mathbb{A}}$ logical states (Def. 32). As described above, classical and boolean assertions (except $\Rightarrow$) are standard. The $\Rightarrow$ integrates logical implication with the ordering relation $\preccurlyeq_{\mathbb{A}} \triangleq (\preccurlyeq_{\langle \text{PLPVal}_{\mathbb{A}} \rangle}, \approx)$ on programs states. In particular, as stated in Lemma 1, the semantics of $\Rightarrow$ incorporates the abstract (de)allocation relation $\approx$ on abstract heaps (Def. 15). The Π describes states of the form $(\mathsf{s}, \mathbf{0})$ where s satisfies Π. Dually, the Θ describes states of the form $(\mathsf{s}, \mathbf{h})$ where s is in the unit set and $\mathbf{h}$ satisfies Θ.

Lemma 1 (Abstract (de)allocation)**.** *For all* $\alpha, \beta \in$ LVar *(Def. 2)* , $\Delta_1, \Delta_2 \in$ DAst$_{\mathbb{A}}$ *(Def. 19) and* $P, Q \in$ PLAst$_{\mathbb{A}}$ *(Def. 36):*

$$\alpha \mapsto \Delta_1 \diamond_\beta \Delta_2 \;\Rightarrow \exists\beta.\, \alpha \mapsto \Delta_1 * \beta \mapsto \Delta_2 \tag{3.1}$$

$$\alpha \mapsto (\Delta_1 \wedge \Diamond\beta) * \beta \mapsto \Delta_2 \;\Rightarrow \alpha \mapsto \Delta_1 \diamond_\beta \Delta_2 \tag{3.2}$$

Proof. Follows immediately from the semantics of the assertions. $\square$

Parts (3.1) and (3.2) describe abstract allocation and deallocation, respectively.

WLogic assertions and their semantics The assertions of WLogic comprise i) the heap cell assertion $\mathrm{x} \mapsto \mathrm{v}$; ii) variable stack assertions comprising the variable cell assertion $\mathtt{x} \Rrightarrow \mathrm{v}$, expression evaluation assertion $\mathtt{e}{\Downarrow}\mathrm{v}$, and boolean expression filters $\mathsf{btrue}(\mathtt{b})$ and $\mathsf{bfalse}(\mathtt{b})$; and iii) inductive assertions comprising the standard classical, boolean and separation logic assertions. WLogic assertions are interpreted over sets of WLogic logical states, with the classical, boolean and SL assertions interpreted in the standard way. The $\mathrm{x} \mapsto \mathrm{v}$ assertion describes a single-cell heap at address x with its value denoted by v. Similarly, the $\mathtt{x} \Rrightarrow \mathrm{v}$ assertion describes a single-cell stack where variable $\mathtt{x}$ is associated with the value denoted by v. The $\mathtt{e}{\Downarrow}\mathrm{v}$ assertion states that in the current stack the program expression $\mathtt{e}$ evaluates to the value denoted by v. Similarly, the $\mathsf{btrue}(\mathtt{b})$ and $\mathsf{bfalse}(\mathtt{b})$ filters assert that in the current stack the boolean expression $\mathtt{b}$ evaluates to true and false, respectively. We write $\mathsf{bsafe}(\mathtt{b})$ for $\mathsf{btrue}(\mathtt{b}) \vee \mathsf{bfalse}(\mathtt{b})$, to denote that $\mathtt{b}$ safely evaluates to a boolean value. Similarly, we write $\mathsf{vsafe}(\mathtt{e})$ for $\exists \mathrm{v}.\ \mathtt{e}{\Downarrow}\mathrm{v}$, to denote that the program expression $\mathtt{e}$ safely evaluates to a value.

As per Par. 12, the WLogic assertions are to be categorised as either *primitive* or *parametric*. Moreover, while the semantics of primitive assertions must not depend on the library extension (i.e. not refer to the abstract logical heaps of the library), the semantics of composite assertions may be lifted to account for the extension. Observe that neither of heap or stack assertions depend on the library heaps and the only assertions that may refer to the logical heaps of the library are SSL assertions and the inductive assertions built from SSL assertions. As such, we declare the heap and stack assertions as primitive, while categorising the inductive assertions as composite.

WLogic Instance (Parameter 17)

Definition 37 (WLogic assertions and their semantics)**.** The set of *primitive* WLogic *assertions* $p \in \textsc{PrimWAst}$, is defined by the following grammar, where $\mathrm{X}, \mathrm{V} \in \textsc{LVar}$ (Def. 2), $\mathtt{x} \in \textsc{PVar}$ (Def. 1), $\mathtt{e} \in \textsc{WExp}$ and $\mathtt{b} \in \textsc{WBExp}$ (Def. 27):

$$p ::= \mathrm{X} \mapsto \mathrm{V} \mid \mathtt{x} \Rrightarrow \mathrm{V} \mid \mathtt{e}{\downarrow}\mathrm{V} \mid \mathsf{btrue}(\mathtt{b}) \mid \mathsf{bfalse}(\mathtt{b})$$

Let $\mathrm{V} \supseteq \textsc{WPVal}$ denote a generic extension of WLogic program values (Def. 23) and $\mathrm{L} \supseteq \textsc{WLVal}$ denote a generic extension of WLogic logical values (Def. 31).
Given a logical environment $\Gamma \in \textsc{LEnv}\langle \mathrm{L} \rangle$ (Def. 3), the *primitive satisfaction relation,* $\models_p$: $(\textsc{LEnv}\langle \mathrm{L} \rangle \times \textsc{WLState}\langle \mathrm{V} \rangle) \times \textsc{PrimWAst}$, is defined as follows, where $\Gamma \in \textsc{LEnv}\langle \mathrm{L} \rangle$, $(\sigma, h) \in \textsc{WLState}\langle \mathrm{V} \rangle$ and $\mathbf{0}$ denotes a function with an empty domain:

$$\begin{array}{lll}
(\Gamma, (\sigma, h)) \models_p \mathrm{X} \mapsto \mathrm{V} & \text{iff} & \sigma{=}\mathbf{0} \wedge \exists x, v.\, \Gamma(\mathrm{X}){=}x \wedge \Gamma(\mathrm{V}){=}v \\
 & & \qquad \wedge\, \mathit{dom}(h){=}\{x\} \wedge h(x){=}v \\
(\Gamma, (\sigma, h)) \models_p \mathtt{x} \Rrightarrow \mathrm{V} & \text{iff} & h{=}\mathbf{0} \wedge \exists v.\, \Gamma(\mathrm{V}){=}v \\
 & & \qquad \wedge\, \mathit{dom}(\sigma){=}\{\mathtt{x}\} \wedge \sigma(\mathtt{x}){=}v \\
(\Gamma, (\sigma, h)) \models_p \mathtt{e}{\downarrow}\mathrm{V} & \text{iff} & \exists v.\, \Gamma(\mathrm{V}){=}v \wedge (\!| \mathtt{e} |\!)^{\sigma} {=} v \\
(\Gamma, (\sigma, h)) \models_p \mathsf{btrue}(\mathtt{b}) & \text{iff} & (\!| \mathtt{b} |\!)^{\sigma} = \mathsf{true} \\
(\Gamma, (\sigma, h)) \models_p \mathsf{bfalse}(\mathtt{b}) & \text{iff} & (\!| \mathtt{b} |\!)^{\sigma} = \mathsf{false}
\end{array}$$

Let $\mathrm{PE} \supseteq \textsc{PrimWLExp}$ denote a generic extension of WLogic primitive logical expressions with $\mathrm{E} \triangleq \textsc{WLExp}\langle \mathrm{PE} \rangle$ (Def. 35), and let $\mathrm{A} \supseteq \textsc{PrimWAst}$ denote a generic extension of primitive WLogic assertions.

The set of WLogic *parametric assertions*, $\textsc{WAst}\langle \mathrm{A}, \mathrm{E}\rangle$, is defined as follows, where $a \in \mathrm{A}$, $E_1, E_2 \in \mathrm{E}$, $\ominus \in \{=, <\}$ and $\mathrm{x} \in \textsc{LVar}$ (Def. 2):

$$\begin{aligned} \textsc{Ast}\langle \mathrm{A}, \mathrm{E}\rangle \ni P, Q ::= \; & a \mid \mathsf{false} \mid P \Rightarrow Q \mid \exists \mathrm{x}.\ P \mid E_1 \ominus E_2 \\ & \mid \mathsf{emp} \mid P * Q \mid P \mathrel{-\!\!*} Q \end{aligned}$$

The WLogic *assertion set* is $P \in \textsc{WAst} \triangleq \textsc{WAst}\langle \textsc{PrimWAst}, \textsc{WLExp}\rangle$. The *parametric ordering relation on* WLState is $\mathsf{id}\langle \mathrm{V}\rangle \subseteq \textsc{WLState}\langle \mathrm{V}\rangle \times \textsc{WLState}\langle \mathrm{V}\rangle$, where id denotes the identity relation.
Let $\mathrm{W} \triangleq (\textsc{WLState}\langle \mathrm{V}\rangle \times \mathrm{D})$ denote a generic extension of parametric WLogic logical states (Def. 33), comprising WLogic logical states in $\textsc{WLState}\langle \mathrm{V}\rangle$, as well as library logical states captured by D. The *parametric satisfaction relation*,

$$\mathsf{sat}(.): \begin{pmatrix} \mathcal{P}(\mathrm{W}\times\mathrm{W}) \\ \times (\mathrm{W}\times\mathrm{W} \rightharpoonup \mathrm{W}) \\ \times \mathcal{P}(\mathrm{W}) \\ \times \mathcal{P}((\textsc{LEnv}\langle \mathrm{L}\rangle \times \mathrm{W}) \times \mathrm{A}) \\ \times (\textsc{LEnv}\langle \mathrm{L}\rangle \times \mathrm{E} \rightharpoonup \mathrm{L}) \end{pmatrix} \rightarrow \mathcal{P}((\textsc{LEnv}\langle \mathrm{L}\rangle \times \mathrm{W}) \times \textsc{Ast}\langle \mathrm{A}, \mathrm{E}\rangle)$$

is defined as follows for $R_{ord} \in \mathrm{W} \times \mathrm{W}$, $\oplus : (\mathrm{W}\times\mathrm{W} \rightharpoonup \mathrm{W})$, $U \in \mathcal{P}(\mathrm{W})$, $R_p \in (\textsc{LEnv}\langle \mathrm{L}\rangle \times \mathrm{W}) \times \mathrm{A}$, $f \in \textsc{LEnv}\langle \mathrm{L}\rangle \times \mathrm{E} \rightharpoonup \mathrm{L}$, $\Gamma \in \textsc{LEnv}\langle \mathrm{L}\rangle$, $\mathrm{x} \in \textsc{LVar}$, $a \in \mathrm{A}$, $w{=}((\sigma, h), d) \in \mathrm{W}$, $P, Q \in \textsc{Ast}\langle \mathrm{A}, \mathrm{E}\rangle$ and $E_1, E_2 \in \mathrm{E}$, where $R \triangleq \mathsf{sat}(R_{ord}, \oplus, U, R_p, f)$:

$$\begin{array}{lll} (\Gamma, w)\ R\ a & \text{iff} & (\Gamma, w)\ R_p\ a \\ (\Gamma, w)\ R\ \mathsf{false} & & \text{never} \\ (\Gamma, w)\ R\ (P \Rightarrow Q) & \text{iff} & (\Gamma, w)\ R\ P \;\Rightarrow\; \exists w'.\,(w, w') \in R_{ord} \\ & & \qquad\qquad\qquad \wedge (\Gamma, w')\ R\ Q \\ (\Gamma, w)\ R\ (\exists \mathrm{x}.\ P) & \text{iff} & \exists x.\ (\Gamma[\mathrm{x} \mapsto x], w)\ R\ P \\ (\Gamma, w)\ R\ E_1 \ominus E_2 & \text{iff} & \|E_1\|^{\Gamma}_{\mathrm{WL}} \ominus \|E_2\|^{\Gamma}_{\mathrm{WL}} \\ (\Gamma, w)\ R\ \mathsf{emp} & \text{iff} & w \in U \end{array}$$

$$
\begin{array}{lll}
(\Gamma, w)\ R\ (P * Q) & \text{iff} & \exists w_1, w_2.\, w{=}w_1{\oplus}w_2 \wedge (\Gamma, w_1)\ R\ P \\
& & \qquad \wedge\, (\Gamma, w_2)\ R\ Q \\
(\Gamma, w)\ R\ (P \mathrel{-\!\!*} Q) & \text{iff} & \forall w'.\, (\epsilon, w')\ R\ P \Rightarrow (\Gamma, w \oplus w')\ R\ Q
\end{array}
$$

The WLogic *satisfaction relation* is $\models_{\mathrm{W}}$: $\big((\text{LEnv}\,\langle \text{WLVal}\rangle \times \text{WLState}) \times \text{WAst}\big) \triangleq \mathsf{sat}(\mathsf{id}, \circ_{\mathrm{W}}, \text{WUnit}, \models_p, \|.\|^{(.)}_{\mathrm{WL}})$.

The $\mathsf{vars}(\overline{\mathtt{x}_i : \mathrm{v}_i}^{i=1\ldots n})$ assertion is a derived assertion in WLogic defined as follows where the $\circledast$ quantifier denotes iterated $*$ (i.e. the finite, multiplicative analogue of $\forall$):

$$\mathsf{vars}(\overline{\mathtt{x}_i : \mathrm{v}_i}^{i=1\ldots n}) \triangleq \underset{i=1\ldots n}{\circledast}\ \mathtt{x}_i \Rightarrow \mathrm{v}_i$$

Example 8 (WLogic$_{\mathbb{L}}$ assertions and their semantics)**.** Given the set of WLogic primitive assertions PrimWAst (Def. 37) and the set of SSL list heap assertions HAst$_{\mathbb{L}}$ (Def. 19), the set of WLogic$_{\mathbb{L}}$ *primitive assertions* is PrimWAst$_{\mathbb{L}} \triangleq$ PrimWAst $\cup$ HAst$_{\mathbb{L}}$.

Given the WLogic logical values WLVal (Def. 37), the list logical values LVal$_{\mathbb{L}}$ (Def. 17), the WLogic$_{\mathbb{L}}$ logical values WLVal$_{\mathbb{L}}$ (Example 5), the logical environments LEnv $\langle$WLVal$_{\mathbb{L}}\rangle$ (Def. 3), the WLogic$_{\mathbb{L}}$ logical states WLState$_{\mathbb{L}}$ (Example 6), the WLogic primitive satisfaction relation $\models_{\mathsf{p}}$ (Def. 37) and the list satisfaction relation $\models_{\mathbb{L}}$ (Def. 20), the WLogic$_{\mathbb{L}}$ *primitive satisfaction relation*, $\models_{\mathsf{p}\mathbb{L}} \subseteq \big(\text{LEnv}\,\langle \text{WLVal}_{\mathbb{L}}\rangle \times \text{WLState}_{\mathbb{L}}\big) \times \text{PrimWAst}_{\mathbb{L}}$, is defined as follows, for all $p \in$ PrimWAst, $\Theta \in$ HAst$_{\mathbb{L}}$ and $(\mathsf{s}, \mathbf{h}) \in$ WLState$_{\mathbb{L}}$, where $\Gamma_w{=}\Gamma{\downarrow}_{\text{WLVal}}$ and $\Gamma_l{=}\Gamma{\downarrow}_{\text{LVal}_{\mathbb{L}}}$ (Def. 3):

$$
\begin{array}{lll}
\Gamma, (\mathsf{s}, \mathbf{h}) \models_{\mathsf{p}\mathbb{L}} p & \text{iff} & \Gamma_p, \mathsf{s} \models_p p \wedge \mathbf{h}{=}\mathbf{0}_{\mathbb{L}} \\
\Gamma, (\mathsf{s}, \mathbf{h}) \models_{\mathsf{p}\mathbb{L}} \Theta & \text{iff} & \mathsf{s}{=}(\mathbf{0}, \mathbf{0}) \wedge \Gamma_a, \mathbf{h} \models_{\mathbb{L}} \Theta
\end{array}
$$

Given the parametric WLogic assertions WAst $\langle.\rangle$ (Def. 37), the set of WLogic$_{\mathbb{L}}$ *assertions* is $P \in$ WAst$_{\mathbb{L}} \triangleq$ WAst $\langle$PrimWAst$_{\mathbb{L}}\rangle$.

Given the parametric ordering relation on WLogic states $\mathsf{id}\,\langle.\rangle$ (Def. 37) and the abstract allocation relation $\approx$ on logical heaps (Def. 15), the *order-*

ing on $\text{WLogic}_{\mathbb{L}}$ *states*, $\preccurlyeq_{\mathbb{L}} \subseteq \text{WLState}_{\mathbb{L}} \times \text{WLState}_{\mathbb{L}}$, is defined componentwise as $\preccurlyeq_{\mathbb{L}} \triangleq (\mathsf{id}\langle \text{WPVal}_{\mathbb{L}} \rangle, \approx)$.

Given the $\text{WLogic}_{\mathbb{L}}$ partial commutative monoid $(\text{WLState}_{\mathbb{L}},+,\text{WUnit}_{\mathbb{L}})$ in Example 6, the expression evaluation function $\|.\|^{(.)}_{\text{WL}\mathbb{L}}$ (Example 7) and the WLogic parametric satisfaction relation $\mathsf{sat}(.)$, the *satisfaction relation for* $\text{WLogic}_{\mathbb{L}}$ *assertions*, $\models_{\text{WL}\mathbb{L}}$: $(\text{LEnv}\langle \text{WLVal}_{\mathbb{L}} \rangle \times \text{WLState}_{\mathbb{L}}) \times \text{WAst}_{\mathbb{L}}$, is:

$$\models_{\text{WL}\mathbb{L}} \triangleq \mathsf{sat}(\preccurlyeq_{\mathbb{L}}, +, \text{WUnit}_{\mathbb{L}}, \models_{\mathsf{p}\mathbb{L}}, \|.\|^{(.)}_{\text{WL}\mathbb{L}})$$

Library specification The axiomatic specification of the library $\mathbb{A}$ operations, $\text{Axiom}_{\mathbb{A}}$, are parameterised and must be provided alongside the other SSL parameters studied so far. For instance, the axiomatic specification of the list library studied in §2.1.3 is given in Fig. 2.2.

SSL Parameter

Parameter 18 (Library specifications)**.** Assume a set of *operation axioms*, $\text{Axiom}_{\mathbb{A}} : \text{PLAst}_{\mathbb{A}} \times \text{Op}_{\mathbb{A}} \times \text{PLAst}_{\mathbb{A}}$.

SSL $\mathbb{L}$ Instance (Parameter 18)

Definition 38 (List axioms)**.** The *axioms of list operations*, $\text{Axiom}_{\mathbb{L}}$, are given in Fig. 2.2.

Proof rules and soundness We assume the proof rules of PLogic to follow an inductive pattern with a set of proof rules for the primitive PL operations, as well as a set of proof rules for composite operations and other assertion rewriting rules (e.g. rule of consequence, frame rule, etc.). The proof rules of $\text{PLogic}_{\mathbb{A}}$ are those of PLogic where i) the operations in the premises and conclusions of proof rules are lifted from PL operations PLOp (Par. 12), to corresponding $\text{PL}_{\mathbb{A}}$ operations in $\text{PLOp}_{\mathbb{A}}$ (Def. 26); and ii) the assertions in the premises and conclusions of proof rules are lifted from PLogic assertions PLAst (Par. 17), to corresponding $\text{PLogic}_{\mathbb{A}}$ assertions in $\text{PLAst}_{\mathbb{A}}$ (Def. 36). We further extend the primitive proof rules of PLogic with the axiomatic library $\mathbb{A}$ specification in $\text{Axiom}_{\mathbb{A}}$

(Par. 18). That is, we extend the proof rules with:

$$\frac{(P, \mathtt{C}, Q) \in \textsc{Axiom}_{\mathbb{A}}}{\{P\}\ \mathtt{C}\ \{Q\}}\ (\textsc{Ax})$$

The soundness of a program logic is typically established by relating the high-level syntactic proof rules to the low-level semantics of the language, where high-level logical states are related to low-level program states via a *reification* function. We thus require that PLogic define a reification function mapping PLogic logical states (Par. 15) onto sets of PL program states (Par. 11). We then modify the definition of state reification and redefine it as a mapping of the extended logical states (containing abstract logical heaps) onto extended program states (containing abstract program heaps). The reification of an extended logical state $(\mathsf{s}, \mathbf{h})$ in $\textsc{PLogic}_{\mathbb{A}}$ is defined component-wise with s reified via the PLogic reification function, and $\mathbf{h}$ reified by *completion*. That is, the reification of an abstract heap $\mathbf{h}$ is the set of all its *completions* in $\text{comp}(\mathbf{h})$ (Def. 13).

In order to establish the soundness of $\textsc{PLogic}_{\mathbb{A}}$, we must show that the $\textsc{PLogic}_{\mathbb{A}}$ triples are *safe* with respect to the PL semantics. That is, if running $\mathtt{C}$ from a state that satisfies P terminates, then the resulting state must satisfy Q. Moreover, the execution of $\mathtt{C}$ must be *frame-preserving*: when run on a bigger state $(P * R)$, the execution of $\mathtt{C}$ must leave R unchanged with its effect contained in P. Put formally, given a $\textsc{PLogic}_{\mathbb{A}}$ triple $\{P\}\ \mathtt{C}\ \{Q\}$, a logical state $\mathsf{w}{=}(\mathsf{s}, \mathbf{h})$ satisfying P, a logical state w_r that is compatible with w (i.e. their composition $\mathsf{w} + \mathsf{w}_r$ is defined), and a program state w in the reification of $\mathsf{w} + \mathsf{w}_r$, then for any program state w' resulting from running $\mathtt{C}$ on w, there must exist a logical state w' such that i) w' satisfies Q; and ii) w' is contained in the reification of $\mathsf{w}' + \mathsf{w}_r$. We formalise this in the upcoming definition of safe triples (Def. 41).

To show the soundness of $\textsc{PLogic}_{\mathbb{A}}$, we then require the PLogic triples to be safe as described above. The soundness of $\textsc{PLogic}_{\mathbb{A}}$ rules is straightforward and follows the same pattern as that of PLogic. In particular, in case of the existing rules lifted from PLogic, the soundness argument remains largely unaffected and is simply lifted to $\textsc{PLogic}_{\mathbb{A}}$. The soundness of the (Ax) rule above is established in a similar way with respect to the semantics of $\mathbb{A}$ operations. Recall that we declare the low-level semantics of library $\mathbb{A}$ operations as an *optional* parameter. When the semantics of the library is provided, we further require the library axioms to be sound with respect to their semantics. On the other hand, as mentioned earlier when the low-level semantics of the library operations is not provided, we describe the default semantics of library $\mathbb{A}$ operations denotationally via their high-level axiomatic semantics (Par. 18). The soundness of library axioms with respect to their semantics is then immediate from the definition of the default library semantics.

PLogic Parameter

Parameter 19 (PLogic reification)**.** Let $\mathrm{V} \supseteq \mathrm{PLPVal}$ denote a generic extension of PL program values (Par. 10).
Given the sets of parametric PL program states $\mathrm{PState}\langle . \rangle$ (Par. 11) and parametric PLogic logical states $\mathrm{LState}\langle . \rangle$ (Par. 15), assume a *parametric reification function*, $\lfloor . \rfloor_{\langle \mathrm{V} \rangle} : \mathrm{LState}\langle \mathrm{V} \rangle \to \mathcal{P}(\mathrm{PState}\langle \mathrm{V} \rangle)$, that maps a logical state in $\mathrm{LState}\langle \mathrm{V} \rangle$ onto a set of program states in $\mathcal{P}(\mathrm{PState}\langle \mathrm{V} \rangle)$. Assume that the PLogic *reification function* is $\lfloor . \rfloor_{\mathrm{PL}} \triangleq \lfloor . \rfloor_{\langle \mathrm{PLPVal} \rangle}$.

Definition 39 (PLogic$_{\mathbb{A}}$ reification)**.** Given the PLogic$_{\mathbb{A}}$ *program states* $\mathrm{PLPStates}_{\mathbb{A}}$ (Def. 24), the PLogic$_{\mathbb{A}}$ *logical states* $\mathrm{PLLState}_{\mathbb{A}}$ (Def. 32), the parametric PLogic reification function $\lfloor . \rfloor_{\langle . \rangle}$ (Par. 19), the PLogic$_{\mathbb{A}}$ logical values $\mathrm{PLLVal}_{\mathbb{A}}$ (Def. 30), the heap completion function comp(.) and the heap collapse function collapse(.) (Def. 13), the PLogic$_{\mathbb{A}}$ *reification function*, $\lfloor . \rfloor_{\mathrm{PL}\mathbb{A}} : \mathrm{PLLState}_{\mathbb{A}} \to \mathcal{P}(\mathrm{PLPStates}_{\mathbb{A}})$, is defined as follows:

$$\lfloor (\mathsf{s}, \mathbf{h}) \rfloor_{\mathrm{PL}\mathbb{A}} \triangleq \Big\{ (s, \mathrm{collapse}(\mathbf{h}')) \mid s \in \lfloor \mathsf{s} \rfloor_{\langle \mathrm{PLPVal}_{\mathbb{A}} \rangle} \land \mathbf{h}' \in \mathrm{comp}(\mathbf{h}) \Big\}$$

WLogic reification Recall that the WLogic logical states are those of WL program states, excluding the fault state $\lightning$. As such, the WLogic reification function simply relates each program state to the singleton set containing the same state.

WLogic Instance (Parameter 19)

Definition 40 (WLogic reification)**.** Let $\mathrm{V} \supseteq \mathrm{WPVal}$ denote an extension of WL program values (Par. 10).
Given the parametric WL program states $\mathrm{WPState}\langle . \rangle$ (Def. 25) and the parametric WLogic logical states $\mathrm{WLState}\langle . \rangle$ (Def. 33), the *parametric* WLogic *reification function*, $\lfloor . \rfloor_{\langle \mathrm{V} \rangle} : \mathrm{WLState}\langle \mathrm{V} \rangle \to \mathcal{P}(\mathrm{WPState}\langle \mathrm{V} \rangle)$, is defined as follows, for $(\sigma, h) \in \mathrm{WLState}\langle \mathrm{V} \rangle$:

$$\lfloor (\sigma, h) \rfloor_{\langle \mathrm{V} \rangle} \triangleq \{(\sigma, h)\}$$

The WLogic *reification function* is $\lfloor . \rfloor_{\mathrm{W}} \triangleq \lfloor . \rfloor_{\langle \mathrm{WPVal} \rangle}$.

Example 9 ($\mathrm{WLogic}_{\mathbb{L}}$ reification)**.** Given the $\mathrm{WLogic}_{\mathbb{L}}$ *program states* $\mathrm{WPState}_{\mathbb{L}}$ (Example 2), the set of $\mathrm{WLogic}_{\mathbb{L}}$ *logical states* $\mathrm{WLState}_{\mathbb{L}}$ (Example 6), the parametric WLogic reification function $\lfloor . \rfloor_{\langle . \rangle}$ (Def. 40), the $\mathrm{WLogic}_{\mathbb{L}}$ logical values $\mathrm{WLVal}_{\mathbb{L}}$ (Example 5), the heap completion function comp(.) and the heap collapse function collapse(.) (Def. 13), the $\mathrm{WLogic}_{\mathbb{L}}$ *reification function*, $\lfloor . \rfloor_{\mathrm{WL}\mathbb{L}} : \mathrm{WLState}_{\mathbb{L}} \to \mathcal{P}(\mathrm{WPState}_{\mathbb{L}})$, is defined as follows, for all $(\mathsf{s}, \mathbf{h}) \in \mathrm{WLState}_{\mathbb{L}}$:

$$\lfloor (\mathsf{s}, \mathbf{h}) \rfloor_{\mathrm{WL}\mathbb{L}} \triangleq \Big\{ (s, \mathrm{collapse}(\mathbf{h}')) \mid s \in \lfloor \mathsf{s} \rfloor_{\langle \mathrm{WLVal}_{\mathbb{L}} \rangle} \land \mathbf{h}' \in \mathrm{comp}(\mathbf{h}) \Big\}$$

Definition 41 (Safe triples)**.** Let A denote a set of assertions, E denote an evaluation environment, O denote a set of operations, S denote a set of program states, W denote a set of logical states, and let $\mathrm{Triples}\langle \mathrm{A}, \mathrm{O} \rangle \triangleq \mathrm{A} \times \mathrm{O} \times \mathrm{A}$. Given a semantics function $[\![.]\!] : \mathrm{O} \to \mathrm{S} \to \mathcal{P}(\mathrm{S})$, a reification function $\lfloor . \rfloor : \mathrm{W} \to \mathcal{P}(\mathrm{S})$, a composition operator $\oplus : \mathrm{W} \times \mathrm{W} \rightharpoonup \mathrm{W}$ and a satisfaction relation $\models \subseteq (\mathrm{E} \times \mathrm{W}) \times \mathrm{A}$, a triple $(a, o, a) \in \mathrm{Triples}\langle \mathrm{A}, \mathrm{O} \rangle$ is *safe with respect to* $[\![.]\!]$, $\lfloor . \rfloor$, $\oplus$ and $\models$, written $\mathsf{safe}((a, o, a), [\![.]\!], \lfloor . \rfloor, \oplus, \models)$, if and only if:

$$\begin{aligned} \mathsf{safe}((a, o, a'), [\![.]\!], \lfloor . \rfloor, \oplus, \models) \overset{\text{def}}{\iff} & \ \forall \epsilon \in \mathrm{E}, s, s' \in \mathrm{S}, w, r \in \mathrm{W}. \\ & \ \epsilon, w \models a \land s \in \lfloor w \oplus r \rfloor \land s' \in [\![o]\!](s) \Rightarrow \\ & \ \exists w' \in \mathrm{W}.\ \epsilon, w' \models a' \land s' \in \lfloor w' \oplus r \rfloor \end{aligned}$$

Triple safety is lifted to sets of triples $T \subseteq \textsc{Triples}\langle \mathrm{A}, \mathrm{O}\rangle$ as follows:

$$\mathsf{safe}(T, [\![.]\!], \lfloor . \rfloor, \oplus, \models) \overset{\text{def}}{\iff} \forall t \in T.\, \mathsf{safe}(t, [\![.]\!], \lfloor . \rfloor, \oplus, \models)$$

SSL Parameter

Parameter 20 (Library semantics)**.** Assume an *optional library semantics function,* $(\!|.|\!)_{\mathbb{A}} : \textsc{Op}_{\mathbb{A}} \to \mathrm{PLPS}\textsc{tates}_{\mathbb{A}} \to \mathcal{P}(\mathrm{PLPS}\textsc{tates}_{\mathbb{A}})$, that associates each library $\mathbb{A}$ operation in $\textsc{Op}_{\mathbb{A}}$ with a state transformer, provided that the set of library axioms $\textsc{Axiom}_{\mathbb{A}}$ (Par. 18) is safe with respect to the semantics function $(\!|.|\!)_{\mathbb{A}}$, reification function $\lfloor . \rfloor_{\mathrm{PL}\mathbb{A}}$ (Def. 39), the composition operator $+$ (Def. 32) and the satisfiability relation $\models_{\mathrm{PL}\mathbb{A}}$ (Def. 36):

$$\mathsf{safe}(\textsc{Axiom}_{\mathbb{A}}, (\!|.|\!)_{\mathbb{A}}, \lfloor . \rfloor_{\mathrm{PL}\mathbb{A}}, +, \models_{\mathrm{PL}\mathbb{A}})$$

When the library semantics function $(\!|.|\!)_{\mathbb{A}}$ is not provided, define the *default semantics* $(\!|.|\!)_{\mathbb{A}}$ as follows, for $\mathtt{C}_{\mathbb{A}} \in \textsc{Op}_{\mathbb{A}}$ (Par. 6) and $w \in \mathrm{PLPS}\textsc{tates}_{\mathbb{A}}$ (Def. 24):

$$(\!|\mathtt{C}_{\mathbb{A}}|\!)_{\mathbb{A}}(w) \triangleq \left\{ w' \;\middle|\; \begin{array}{l} \forall P, Q, \Gamma, \mathsf{w}_1, \mathsf{w}_r. \\ (P, \mathtt{C}_{\mathbb{A}}, Q) \in \textsc{Axiom}_{\mathbb{A}} \\ \wedge\, \Gamma, \mathsf{w}_1 \models_{\mathrm{PL}\mathbb{A}} P \wedge w \in \lfloor \mathsf{w}_1 + \mathsf{w}_r \rfloor_{\langle \mathrm{PLPV}\textsc{al}_{\mathbb{A}}\rangle} \\ \qquad \Longrightarrow \\ \exists \mathsf{w}_2.\, \Gamma, \mathsf{w}_2 \models_{\mathrm{PL}\mathbb{A}} Q \wedge w' \in \lfloor \mathsf{w}_2 \circ \mathsf{w}_r \rfloor_{\langle \mathrm{PLPV}\textsc{al}_{\mathbb{A}}\rangle} \end{array} \right\}$$

Observe that when the semantics of the library is not provided, the semantics of a library operation is defined denotationally by simply reifying the high-level logical states of its axiomatic specification into their low-level program state counterparts, provided that the axiomatic specification is safe with respect to its semantics. In other words, when the low-level semantics of the library operations are not provided, we simply define a denotational interpretation $[\![.]\!]_{\mathrm{PL}\mathbb{A}}$. Observe that the soundness of axiomatic specification of the library in $\textsc{Axiom}_{\mathbb{A}}$ then follows immediately from the definition of the default semantics.

For the list library $\mathbb{L}$, we do not devise a low-level semantics and opt instead for the default semantics provided by Par. 20. As we mentioned earlier, when the low-level semantics of a library is not provided, the axiomatic specification of the library operations is typically justified against an implementation of the library operations. In §6 we provide an implementation of the list library $\mathbb{L}$ operations and describe how to justify the correctness of the axiomatic list specification (given in Fig. 2.2 of §2) with respect to this implementation.

SSL $\mathbb{L}$ Instance (Parameter 20)

Definition 42 (List semantics)**.** The semantics of the list library operations, $\langle\!|.|\!\rangle_{\mathbb{L}} : \textsc{Op}_{\mathbb{L}} \to \text{PLPS}\textsc{tates}_{\mathbb{A}} \to \mathcal{P}(\text{PLPS}\textsc{tates}_{\mathbb{A}})$, is given by the default library semantics in Par. 20.

PLogic Parameter

Parameter 21 (PLogic triples)**.** Given the primitive PL operations PrimPLOp (Par. 12), PLogic assertions PLAst and the satisfiability relation $\models_{\text{PL}}$ (Par. 17), the PL semantics function $[\![.]\!]_{\text{PL}}$ (Par. 13), the PLogic reification function $\lfloor.\rfloor_{\text{PL}}$ (Par. 19) and the PLogic composition operator $\circ$ (Par. 15), assume a set of PLogic *primitive triples*, PrimPLTriples $\subseteq$ Triples $\langle$PLAst, PrimPLOp$\rangle$, such that $\mathsf{safe}(\text{PrimPLTriples}, [\![.]\!]_{\text{PL}}, \lfloor.\rfloor_{\text{PL}}, \circ, \models_{\text{PL}})$ holds (Def. 41).
Let $\text{V} \supseteq \text{PLPVal}$ denote an extension of PL program values (Par. 10) and $\text{L} \supseteq \text{PLLVal}$ denote an extension of PLogic logical values (Par. 14).
Let $\text{O} \supseteq \text{PrimPLOp}$ denote an extension of primitive PL operations (Par. 12).
Let $\text{S} \triangleq \text{PState}\langle\text{V}\rangle \times \text{C}$ denote an extension of parametric PL program states (Par. 11), comprising PL programs states in $\text{PState}\langle\text{V}\rangle$, as well as library program states captured by C.
Let $\text{W} \triangleq \text{LState}\langle\text{V}\rangle \times \text{D}$ denote an extension of PLogic logical states (Par. 15), comprising PLogic logical states in $\text{LState}\langle\text{V}\rangle$, as well as library logical states captured by D.

Let $\text{PE} \supseteq \textsc{PrimPLLExp}$ denote an extension of $\textsc{PLogic}$ primitive expressions with $\text{E} \triangleq \textsc{LExp}\langle\text{PE}\rangle$ (Par. 16).
Let $\text{PA} \supseteq \textsc{PrimPLAst}$ denote an extension of $\textsc{PLogic}$ primitive assertions with $\text{A} \triangleq \textsc{Ast}\langle\text{PA}, \text{E}\rangle$ (Par. 17).
Assume a *parametric triple making function*,

$$tf(.) : \begin{pmatrix} \quad \mathcal{P}(\textsc{Triples}\langle\text{A}, \text{O}\rangle) \\ \times \big(\textsc{Op}\langle\text{O}\rangle \to \text{S} \to \mathcal{P}(\text{S})\big) \\ \times (\text{W} \to \mathcal{P}(\text{S})) \\ \times (\text{W} \times \text{W} \rightharpoonup \text{W}) \\ \times \mathcal{P}((\textsc{Env}\langle\text{L}\rangle \times \text{W}) \times \text{A}) \end{pmatrix} \to \mathcal{P}(\textsc{Triples}\langle\text{A}, \textsc{Op}\langle\text{O}\rangle\rangle)$$

that given a set of triples $T \subseteq \textsc{Triples}\langle\text{A}, \text{O}\rangle$ describing the behaviour of primitive operations in O, an interpretation function $i \in \textsc{Op}\langle\text{O}\rangle \to \text{S} \to \mathcal{P}(\text{S})$, a reification function $r \in \text{W} \to \mathcal{P}(\text{S})$, a composition operator $\oplus : (\text{W} \times \text{W} \rightharpoonup \text{W})$ and a satisfaction relation $S \subseteq (\textsc{Env}\langle\text{L}\rangle \times \text{W}) \times \text{A}$, it produces a set of triples $T' \subseteq \textsc{Triples}\langle\text{A}, \textsc{Op}\langle\text{O}\rangle\rangle$ with the proviso that:

$$\mathsf{safe}(T, i, r, \oplus, S) \implies T \subseteq T' \wedge \mathsf{safe}(T', i, r, \oplus, S)$$

Assume that the set of $\textsc{PLogic}$ *triples* is defined as:

$$\textsc{PLTriples} \triangleq tf(\textsc{PrimPLTriples}, [\![.]\!]_{\text{PL}}, \lfloor.\rfloor_{\text{PL}}, \circ, \models_{\text{PL}})$$

where $\circ$ denotes the $\textsc{PLogic}$ composition operator (Par. 15).

Definition 43 ($\textsc{PLogic}_{\mathbb{A}}$ triples)**.** Given the primitive $\textsc{PLogic}$ triples in $\textsc{PrimPLTriples}$ (Par. 21) and the library axioms $\textsc{Axiom}_{\mathbb{A}}$ (Par. 18), the set of $\textsc{PLogic}_{\mathbb{A}}$ *primitive triples* is:

$$\textsc{PrimPLTriples}_{\mathbb{A}} \triangleq \textsc{PrimPLTriples} \cup \textsc{Axiom}_{\mathbb{A}}$$

Given the $\textsc{PLogic}$ triple making function tf (Par. 21), the $\textsc{PLogic}_{\mathbb{A}}$ composition operator $+$ (Def. 32), the $\textsc{PLogic}_{\mathbb{A}}$ semantics function $[\![.]\!]_{\text{PL}\mathbb{A}}$ (Def. 28), the $\textsc{PLogic}_{\mathbb{A}}$ reification function $\lfloor.\rfloor_{\text{PL}\mathbb{A}}$ (Def. 39) and the $\textsc{PLogic}_{\mathbb{A}}$

satisfiability relation $\models_{\text{PL}\mathbb{A}}$ (Def. 36), the set of PLogic$_{\mathbb{A}}$ *triples*, PLTriples$_{\mathbb{A}}$, is defined as follows:

$$\text{PLTriples}_{\mathbb{A}} \triangleq tf(\text{PrimPLTriples}_{\mathbb{A}}, [\![.]\!]_{\text{PL}\mathbb{A}}, \lfloor . \rfloor_{\text{PL}\mathbb{A}}, +, \models_{\text{PL}\mathbb{A}})$$

Lemma 2 (Primitive soundness)**.** *The primitive triples* PrimPLTriples$_{\mathbb{A}}$ *(Def. 43) are safe with respect to the* PLogic$_{\mathbb{A}}$ *composition + (Def. 32), the* PLogic$_{\mathbb{A}}$ *semantics function* $[\![.]\!]_{\text{PL}\mathbb{A}}$ *(Def. 28), the* PLogic$_{\mathbb{A}}$ *reification function* $\lfloor . \rfloor_{\text{PL}\mathbb{A}}$ *(Def. 39) and the* PLogic$_{\mathbb{A}}$ *satisfaction relation* $\models_{\text{PL}\mathbb{A}}$ *(Def. 36):*

$$\mathsf{safe}(\text{PrimPLTriples}_{\mathbb{A}}, [\![.]\!]_{\text{PL}\mathbb{A}}, \lfloor . \rfloor_{\text{PL}\mathbb{A}}, +, \models_{\text{PL}\mathbb{A}})$$

Proof. follows from the safety of primitive PLogic operations (Par. 21) and the safety of library operations (Par. 20). □

Theorem 1 (PLogic$_{\mathbb{A}}$ soundness)**.** *The* PLogic$_{\mathbb{A}}$ *triples in* PLTriples$_{\mathbb{A}}$ *(Def. 43) are safe with respect to the* PLogic$_{\mathbb{A}}$ *semantics function* $[\![.]\!]_{\text{PL}\mathbb{A}}$ *(Def. 28), the* PLogic$_{\mathbb{A}}$ *reification function* $\lfloor . \rfloor_{\text{PL}\mathbb{A}}$ *(Def. 39), the* PLogic$_{\mathbb{A}}$ *composition operator + (Def. 32) and the* PLogic$_{\mathbb{A}}$ *satisfaction relation* $\models_{\text{PL}\mathbb{A}}$ *(Def. 36):*

$$\mathsf{safe}(\text{PLTriples}_{\mathbb{A}}, [\![.]\!]_{\text{PL}\mathbb{A}}, \lfloor . \rfloor_{\text{PL}\mathbb{A}}, +, \models_{\text{PL}\mathbb{A}})$$

Proof. follows immediately from Lemma 2 and the definition of PLTriples$_{\mathbb{A}}$ (Def. 43). □

WLogic proof rules and soundness The WLogic proof rules include i) the axioms describing the behaviour of primitive operations; ii) the standard rules for composite operations `skip`, sequential and parallel composition, conditional choice and loop; and iii) the standard syntactic rules for assertion manipulation including consequence, frame, disjunction and existential elimination. The proof rules of WLogic are given in Fig. 3.2. The proof rules for primitive while operations are straightforward. The rules for `skip`, sequential and parallel composition, as well as syntactic assertion manipulation rules of consequence, frame, disjunction and existential elimination are standard. Observe that the treatment of variables as resource [5] eliminates the need for a side-conditions in the frame (resp. parallel) rule, requiring that variables mentioned in the frame (resp. pre- and postconditions) and those used in the program be disjoint. Recall that scoped variable declaration (`var x in {C}`) allows us to declare a local variable `x` scoped within `C`. As the variable stack may already contain a variable named `x`, the `var x in {C}` rule allows us to extend the stack with a *fresh* variable `z` for the duration of `C`, provided that all occurrences of `x` in `C` are accordingly substituted by `z`, and that `z` is removed from the stack once `C` has completed its execution. The if statement rule is standard and requires a precondition (P) from which we can derive the precondition of the first (resp. second) branch when `b` evaluates to true (resp. false). The $P \vdash \mathsf{bsafe}(\texttt{b})$ ensures that `b` can be evaluated without the program faulting. Similarly, the while statement rule requires us to prove that P is a loop invariant. That is, when run from P in a state where `b` evaluates to true, the loop body re-establishes P when run from P. As such, if P holds before the loop starts, then it also holds upon termination of the loop at which point `b` evaluates to false. As with the `if` statement rule, the $P \vdash \mathsf{bsafe}(\texttt{b})$ ensures that `b` can be evaluated without the program faulting.

The WLogic proof rules are fault-avoiding and are sound with respect to the WLogic operational semantics and reification function (Def. 40). This is formalised in Theorem 2 below.

Given the parametric WL program states $\text{WPState}\,\langle \text{V} \rangle$ (Def. 25), let $\text{S} \triangleq \text{WPState}\,\langle \text{V} \rangle \times \text{C}$ denote a generic extension of WL program states, comprising parametric WL program states, as well as library program states captured by C.

WLogic Instance (Parameter 21)

Definition 44 (WLogic triples)**.** The set of WLogic *primitive triples*, PrimWTriples, is given by the axioms in the top part of Fig. 3.2.

Let $\text{V} \supseteq \text{WPVal}$ denote an extension of WL program values (Def. 23) and $\text{L} \supseteq \text{WLVal}$ denote an extension of WLogic logical values (Def. 31).

Let $\text{O} \supseteq \text{PrimWOp}$ denote an extension of WL primitive operations

$$\begin{array}{rcl}
\{\mathrm{x} \Rrightarrow \mathrm{X}\} & \texttt{x = alloc()} & \{\exists \mathrm{Y}.\ \mathrm{x} \Rrightarrow \mathrm{Y} * \mathrm{Y} \mapsto -\} \\
\{\mathrm{x} \Rrightarrow \mathrm{X} * \mathrm{X} \mapsto \mathrm{V}\} & \texttt{dealloc(x)} & \{\mathrm{x} \Rrightarrow -\} \\
\{\mathrm{x} \Rrightarrow \mathrm{X} * \mathrm{y} \Rrightarrow \mathrm{Y} * \mathrm{Y} \mapsto \mathrm{V}\} & \texttt{x = [y]} & \{\mathrm{x} \Rrightarrow \mathrm{V} * \mathrm{y} \Rrightarrow \mathrm{Y} * \mathrm{Y} \mapsto \mathrm{V}\} \\
\{\mathrm{x} \Rrightarrow \mathrm{X} * \mathrm{y} \Rrightarrow \mathrm{Y} * \mathrm{X} \mapsto \mathrm{V}\} & \texttt{[x] = y} & \{\mathrm{x} \Rrightarrow \mathrm{X} * \mathrm{y} \Rrightarrow \mathrm{Y} * \mathrm{X} \mapsto \mathrm{Y}\} \\
\{(\mathrm{x} \Rrightarrow \mathrm{X} * P) \wedge \mathsf{e}{\downarrow}\mathrm{V}\} & \texttt{x = e} & \{\mathrm{x} \Rrightarrow \mathrm{V} * P\}
\end{array}$$

$$\frac{\{P * \mathrm{z} \Rrightarrow -\}\ \texttt{C}[\mathrm{z}/\mathrm{x}]\ \{Q * \mathrm{z} \Rrightarrow -\}}{\{P\}\ \texttt{var x in \{C\}}\ \{Q\}}\ \mathsf{fresh}(\mathrm{z})$$

$$\frac{(P, \texttt{C}, Q) \in PT}{\{P\}\ \texttt{C}\ \{Q\}} \qquad \{P\}\ \texttt{skip}\ \{P\}$$

$$\frac{\{P\}\ \texttt{C}_1\ \{R\} \quad \{R\}\ \texttt{C}_2\ \{Q\}}{\{P\}\ \texttt{C}_1;\texttt{C}_2\ \{Q\}} \qquad \frac{P \vdash P' \quad \{P'\}\ \texttt{C}\ \{Q'\} \quad Q' \vdash Q}{\{P\}\ \texttt{C}\ \{Q\}}$$

$$\frac{\{P_1\}\ \texttt{C}_1\ \{Q_1\} \quad \{P_2\}\ \texttt{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \texttt{C}_1 \,||\, \texttt{C}_2\ \{Q_1 * Q_2\}} \qquad \frac{\{P\}\ \texttt{C}\ \{Q\}}{\{P * R\}\ \texttt{C}\ \{Q * R\}}$$

$$\frac{P \vdash \mathsf{bsafe}(\texttt{b}) \quad \begin{array}{c}\{P \wedge \mathsf{btrue}(\texttt{b})\}\ \texttt{C}_1\ \{Q\} \\ \{P \wedge \mathsf{bfalse}(\texttt{b})\}\ \texttt{C}_2\ \{Q\}\end{array}}{\{P\}\ \texttt{if (b) then \{C}_1\texttt{\} else \{C}_2\texttt{\}}\ \{Q\}} \qquad \frac{\{P_1\}\ \texttt{C}\ \{Q_1\} \quad \{P_2\}\ \texttt{C}\ \{Q_2\}}{\{P_1 \vee P_2\}\ \texttt{C}\ \{Q_1 \vee Q_2\}}$$

$$\frac{P \vdash \mathsf{bsafe}(\texttt{b}) \quad \{P \wedge \mathsf{btrue}(\texttt{b})\}\ \texttt{C}\ \{P\}}{\{P\}\ \texttt{while(b)\{C\}}\ \{P \wedge \mathsf{bfalse}(\texttt{b})\}} \qquad \frac{\{P\}\ \texttt{C}\ \{Q\}}{\{\exists \mathrm{x}.\, P\}\ \texttt{C}\ \{\exists \mathrm{x}.\, Q\}}$$

Figure 3.2.: WLogic proof rule with primitive rules (above) and generic composite rules parametric in primitive rules from *PT* (below)

(Def. 27).

Let $\mathrm{S} \triangleq \mathrm{WPS}\textsc{tate}\langle \mathrm{V}\rangle \times \mathrm{C}$ denote an extension of WL program states (Def. 25), comprising WL program states in $\mathrm{WPS}\textsc{tate}\langle \mathrm{V}\rangle$, as well as library program states captured by C.

Let $\mathrm{W} \triangleq \mathrm{WLS}\textsc{tate}\langle \mathrm{V}\rangle \times \mathrm{D}$ denote an extension of WLogic logical states (Def. 33), comprising WLogic logical states in $\mathrm{WLS}\textsc{tate}\langle \mathrm{V}\rangle$, as well as library logical states captured by D.

Let $\mathrm{PE} \supseteq \textsc{PrimWLExp}$ denote an extension of WLogic logical expressions (Def. 35) with $\mathrm{E} \triangleq \mathrm{LE}\textsc{xp}\langle \mathrm{PE}\rangle$.

Let $\mathrm{PA} \supseteq \textsc{PrimWAst}$ denote an extension of primitive WLogic assertions (Def. 37) with $\mathrm{A} \triangleq \textsc{Ast}\langle \mathrm{PA}, \mathrm{E}\rangle$.

The WLogic *parametric triple making function*,

$$
\mathit{tfw}(.) : \begin{pmatrix} \mathcal{P}(\textsc{Triples}\langle \mathrm{A}, \mathrm{O}\rangle) \\ \times \big(\mathrm{WO}\textsc{p}\langle \mathrm{O}\rangle \to \mathrm{S} \to \mathcal{P}(\mathrm{S})\big) \\ \times (\mathrm{W} \to \mathcal{P}(\mathrm{S})) \\ \times (\mathrm{W} \times \mathrm{W} \rightharpoonup \mathrm{W}) \\ \times \mathcal{P}((\mathrm{LE}\textsc{nv}\langle \mathrm{L}\rangle \times \mathrm{W}) \times \mathrm{A}) \end{pmatrix} \to \mathcal{P}(\textsc{Triples}\langle \mathrm{A}, \mathrm{WO}\textsc{p}\langle \mathrm{O}\rangle\rangle)
$$

is defined as follows for a set of primitive triples $PT \subseteq \textsc{Triples}\langle \mathrm{A}, \mathrm{O}\rangle$, an interpretation function $i \in \mathrm{WO}\textsc{p}\langle \mathrm{O}\rangle \to \mathrm{S} \to \mathcal{P}(\mathrm{S})$, a reification function $r \in \mathrm{W} \to \mathcal{P}(\mathrm{S})$, a composition operator $\oplus : \mathrm{W} \times \mathrm{W} \rightharpoonup \mathrm{W}$ and a satisfaction relation $S \subseteq (\mathrm{LE}\textsc{nv}\langle \mathrm{L}\rangle \times \mathrm{W}) \times \mathrm{A}$:

$$
\mathit{tfw}(PT, i, r, \oplus, S) \triangleq \begin{cases} T & \text{if } \mathsf{safe}(PT, i, r, \oplus, S) \\ \emptyset & \text{otherwise} \end{cases}
$$

where T is defined by the rules in the bottom part of Fig. 3.2, and $P \vdash P' \overset{\text{def}}{\iff} \forall \Gamma \in \mathrm{LE}\textsc{nv}\langle \mathrm{L}\rangle, w \in \mathrm{W}.\ (\Gamma, w)\ S\ (P \Rightarrow P')$.

For all $PT \subseteq \textsc{Triples}\langle \mathrm{A}, \mathrm{O}\rangle$, $i \in \mathrm{WO}\textsc{p}\langle \mathrm{O}\rangle \to \mathrm{S} \to \mathcal{P}(\mathrm{S})$, $r \in \mathrm{W} \to \mathcal{P}(\mathrm{S})$, $\oplus : \mathrm{W} \times \mathrm{W} \rightharpoonup \mathrm{W}$ and $S \subseteq (\mathrm{LE}\textsc{nv}\langle \mathrm{L}\rangle \times \mathrm{W}) \times \mathrm{A}$, if $\mathsf{safe}(PT, i, r, \oplus, S)$ holds, then the set of triples $T = \mathit{tfw}(PT, i, r, \oplus, S)$ contains PT and $\mathsf{safe}(T, i, r, \oplus, S)$ holds (Theorem 2).

The set of WLogic *triples*, WTriples, is defined as follows:

$$\text{WTriples} \triangleq \mathit{tfw}(\text{PrimWTriples}, [\![.]\!]_{\text{W}}, \lfloor.\rfloor_{\text{W}}, \circ_{\text{W}}, \models_{\text{W}})$$

Example 10 (WLogic$_{\mathbb{L}}$ triples)**.** Given the primitive triples PrimWTriples (Def. 44) and the list axioms Axiom$_{\mathbb{L}}$ (Def. 38), the set of WLogic$_{\mathbb{L}}$ *primitive triples* is:

$$\text{PrimWTriples}_{\mathbb{L}} \triangleq \text{PrimWTriples} \cup \text{Axiom}_{\mathbb{L}}$$

Given the WLogic triple making function *tfw* (Def. 44), the WLogic$_{\mathbb{L}}$ composition operator $+$ (Example 6), the WLogic$_{\mathbb{L}}$ semantics function $[\![.]\!]_{\text{WL}\mathbb{L}}$ (Example 4), the WLogic$_{\mathbb{L}}$ reification function $\lfloor.\rfloor_{\text{WL}\mathbb{L}}$ (Example 9) and the WLogic$_{\mathbb{L}}$ satisfiability relation $\models_{\text{WL}\mathbb{L}}$ (Example 8), the set of WLogic$_{\mathbb{L}}$ *triples* is:

$$\text{WTriples}_{\mathbb{L}} \triangleq \mathit{tfw}(\text{PrimWTriples}_{\mathbb{L}}, [\![.]\!]_{\text{WL}\mathbb{L}}, \lfloor.\rfloor_{\text{WL}\mathbb{L}}, +, \models_{\text{WL}\mathbb{L}})$$

Theorem 2 (WLogic soundness)**.** *For all* C, D *and for all* $\text{V} \supseteq \text{WPVal}$ *(Def. 23),* $\text{S} \triangleq \text{WPState}\langle\text{V}\rangle \times \text{C}$ *(Def. 25),* $\text{W} \triangleq \text{WLState}\langle\text{V}\rangle \times \text{D}$ *(Def. 33),* $\oplus : \text{W} \times \text{W} \rightharpoonup \text{W}$*,* $\text{L} \supseteq \text{WLVal}$ *(Def. 31),* $\text{PE} \supset \text{PrimWLExp}$ *and* $\text{E} \triangleq \text{LExp}\langle\text{PE}\rangle$ *(Def. 35),* $\text{PA} \supset \text{PrimWAst}$ *and* $\text{A} \triangleq \text{WAst}\langle\text{PA}, \text{E}\rangle$ *(Def. 37),* $\text{O} \supseteq \text{PrimWOp}$ *(Def. 27),* $PT \subseteq \text{Triples}\langle\text{A}, \text{O}\rangle$*,* $i \in \text{WOp}\langle\text{O}\rangle \to \text{S} \to \mathcal{P}(\text{S})$*,* $r \in \text{W} \to \mathcal{P}(\text{S})$ *and* $S \subseteq (\text{LEnv}\langle\text{L}\rangle \times \text{W}) \times \text{A}$*, if* PT *is safe with respect to* i*,* r*,* $\oplus$ *and* S*, then the set of triples* $T{=}\mathit{tfw}(PT, i, r, \oplus, S)$ *contains* PT *and is also safe with respect to* i*,* r*,* $\oplus$ *and* S*:*

$$\mathsf{safe}(PT, i, r, \oplus, S) \implies PT \subseteq T \land \mathsf{safe}(T, i, r, \oplus, S) \tag{3.3}$$

Proof. Pick an arbitrary $t \in T$. From the definitions of T and *tfw* (Def. 44), we then know the following (since otherwise $T{=}\emptyset$ and thus $t \notin T$):

$$\mathsf{safe}(PT, i, r, \oplus, S) \tag{3.4}$$

From the definitions of T and the triple making function *tfw* we know that either i) $t \in PT$; or $t \in \mathit{tfw}(PT, i, r, \oplus, S) \setminus PT$ (i.e. t is one of the inductive cases). When $t \in PT$, then its safety follows immediately from 3.4. Now let us assume $t{=}(P, \mathtt{C}, Q) \in \mathit{tfw}(PT, i, r, \oplus, S) \backslash PT$ (i.e. t is one of the inductive triples). We are then required to show:

$$\mathsf{safe}(\mathit{tfw}(PT, i, r, \oplus, S) \backslash PT, i, r, \oplus, S) \tag{3.5}$$

In his thesis [60], Wright shows that the WLogic triples have a partial, fault avoiding interpretation. That is,

$$\begin{aligned}
&\forall (P, \mathtt{C}, Q) {\in} \text{WTriples}, \Gamma {\in} \text{LEnv}\left\langle \text{WLVal} \right\rangle, s, s' {\in} \text{WPState}, w, r {\in} \text{WLState}. \\
&\quad \Gamma, w \models_{\text{W}} P \land s \in \lfloor w \circ_{\text{W}} r \rfloor_{\text{W}} \land s' \in [\![\mathtt{C}]\!]_{\text{W}}(s) \Rightarrow \\
&\qquad\qquad s' \neq \lightning \land \exists w' \in \text{WLState}.\ \Gamma, w' \models_{\text{W}} Q \land s' \in \lfloor w' \circ_{\text{W}} r \rfloor_{\text{W}}
\end{aligned} \tag{3.6}$$

From the definition of safe triples (Def. 41) and (3.6) we have:

$$\mathsf{safe}(\text{WTriples}, [\![.]\!]_{\text{W}}, \lfloor . \rfloor_{\text{W}}, \circ_{\text{W}}, \models_{\text{W}})$$

From the definition of WTriples and the above we then have:

$$\mathsf{safe}(\mathit{tfw}(\text{PrimWTriples}, [\![.]\!]_{\text{W}}, \lfloor . \rfloor_{\text{W}}, \circ_{\text{W}}, \models_{\text{W}}), [\![.]\!]_{\text{W}}, \lfloor . \rfloor_{\text{W}}, \circ_{\text{W}}, \models_{\text{W}}) \tag{3.7}$$

The proof of (3.6), and consequently that of (3.7), is by induction over the structure of triples in WTriples. It is straightforward to lift the inductive proof of (3.7) to establish the correctness of a generic extension of WLogic as required by (3.5). In particular, as $\mathit{tfw}(PT, i, r, \oplus, S) \backslash PT$ includes inductive cases only, the proof of all cases follows from the (lifted) inductive hypotheses and the safety of primitive triples in PT (3.4).

We now have described all the necessary ingredients for extending an SL-based program logic to enable reasoning about the clients of an abstract library $\mathbb{A}$.

In what follows we present a simple tree library $\mathbb{T}$ and use WLogic$_{\mathbb{T}}$ (the WLogic program logic extended with the tree library $\mathbb{T}$ using the methodology described here) to specify the behaviour of its operations and to reason locally about its client programs. Later in §5 we study the Document Object Model (DOM) library and demonstrate how to specify its operations and reason about its client programs. Although the underlying data structure of the DOM library is rather complex, it shares fundamental similarities with that of the tree library $\mathbb{T}$ in the upcoming chapter. As such, we present the tree library $\mathbb{T}$ in the upcoming chapter as a precursor to the DOM library in §5 in order to familiarise the reader with the necessary reasoning components.

4. A Tree Library: $\mathbb{T}$

We study a simple tree library $\mathbb{T}$ for reading from and updating a simple tree data structure. In §4.1 we instantiate the general theory of SSL presented in §3.1 in order to model the tree library $\mathbb{T}$, and to write assertions describing the underlying tree data structure. We then integrate our tree assertions with those of WLogic in §3.2 in order to specify the behaviour of library $\mathbb{T}$ operations. In §4.2 we use our specification to reason about several client programs of $\mathbb{T}$.

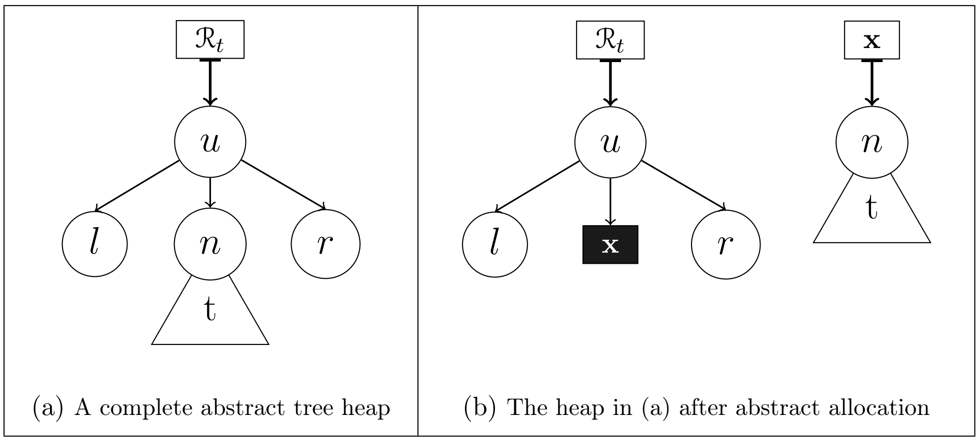
4.1. SSL Model and Assertions: Library $\mathbb{T}$

Recall that the general theory of SSL is parametric and may be instantiated accordingly for a library of structured data. In §3 we presented the general theory of SSL with its parameters delineated in solid boxes labelled "SSL Parameter". In what follows, we revisit the SSL parameters and instantiate them for the tree library $\mathbb{T}$. As before, we present these instances in dashed boxes labelled "SSL $\mathbb{T}$ Instance (Parameter X)", where X is the reference to the corresponding SSL parameter in §3.

Tree root addresses Recall that *program heaps* (e.g. the tree program heap in Fig. 4.1a) are mappings from *root addresses* to complete *program data* with no context holes. For the tree library, we define a designated *root address*, $\mathcal{R}_t$, denoting the location in the tree heap where the tree data is stored.

SSL $\mathbb{T}$ Instance (Parameter 1)

Definition 45 (Tree root addresses)**.** The set of *tree root addresses* is $\textsc{RAdd}_{\mathbb{T}} \triangleq \{\mathcal{R}_t\}$.



$\mathcal{R}_t$
u
l
n
r
t
(a) A complete abstract tree heap
$\mathcal{R}_t$
u
l
x
r
x
n
t
(b) The heap in (a) after abstract allocation

Figure 4.1.: Abstract tree heaps

Tree program data Recall that *program data* is library specific and provides a high-level representation of the underlying data structure that is agnostic to how the data structure may be represented in the machine. The program data for trees describes a *forest* which is an ordered collection of trees. Forests are defined inductively and include empty forests ($\varnothing$), forests with singleton trees (e.g. $n[\mathrm{t}]$ in Fig. 4.1) and composite forests (e.g. $l \otimes n[\mathrm{t}] \otimes r$ in Fig. 4.1a). For brevity, we write n for $n[\varnothing]$. To model forests, we assume a countably infinite set of *node identifiers*, $n \in \textsc{Id}$.

SSL $\mathbb{T}$ Instance (Parameter 2)

Definition 46 (Tree program data)**.** Let $n \in \textsc{Id}$ denote a countably infinite set of *node identifiers*. The set of *program data for trees*, $\mathrm{t} \in \textsc{PData}_{\mathbb{T}}$, is defined by the following grammar, where $n \in \textsc{Id}$:

$$\mathrm{t} ::= \varnothing \mid n[\mathrm{t}] \mid \mathrm{t}_1 \otimes \mathrm{t}_2$$

Tree data does not contain repeated identifiers; the $\otimes$ operation is associative with identity $\varnothing$ and all tree data are equal up to the associativity of $\otimes$.[1]

[1]It is straightforward to formalise these restrictions.

Tree program values Recall from §3.1 that SSL assumes an abstract library to define a set of *library-specific program values* that include root addresses. Library-specific program values denote the set of values that may be observed by the clients of the library. For the tree library $\mathbb{T}$, the program values include the tree root address $\mathcal{R}_t$, as well as the node identifiers in $\textsc{Id}$.

SSL $\mathbb{T}$ Instance (Parameter 3)

Definition 47 (Tree program values)**.** Given the set of tree root addresses $\textsc{RAdd}_{\mathbb{T}}$ (Def. 45) and the set of node identifiers $\textsc{Id}$ (Def. 46), the set of *program values for trees* is $v \in \textsc{PVal}_{\mathbb{T}} \triangleq \textsc{RAdd}_{\mathbb{T}} \cup \textsc{Id}$.

Tree logical data Recall that logical heaps (with context holes) such as the logical tree heap in Fig. 4.1b are mappings from addresses to *logical data*. As with program data, logical data is library specific. Similar to tree program data, logical data for trees describes a *forest* which is an ordered collection of trees. Forests are defined inductively and include empty forests, forests with singleton trees and composite forests. Moreover, logical tree data may be incomplete with context holes. As before, we use the **boldface** font and write $\mathbf{t}$, $\mathbf{t}_1$ and so forth to range over logical tree data. This is to remind the reader that logical tree data may contain context holes.

SSL $\mathbb{T}$ Instance (Parameter 4)

Definition 48 (Tree logical data)**.** The set of *logical data for trees*, $\mathbf{t} \in \textsc{LData}_{\mathbb{T}}$, is defined by the following grammar, where $\mathbf{x} \in \textsc{AAdd}$ (Def. 8) and $n \in \textsc{Id}$ (Def. 46):

$$\mathbf{t} ::= \varnothing \mid \mathbf{x} \mid n[\mathbf{t}] \mid \mathbf{t}_1 \otimes \mathbf{t}_2$$

Tree data does not contain repeated identifiers or context holes; the $\otimes$ operation is associative with identity $\varnothing$ and all tree data are equal upto the associativity of $\otimes$.[2]

The *tree address function*, $\mathrm{addr}_{\mathbb{T}}(.) : \textsc{LData}_{\mathbb{T}} \to \mathcal{P}(\textsc{AAdd})$, is defined inductively over the structure of abstract tree data as follows:

$$\mathrm{addr}_{\mathbb{T}}(\varnothing) \triangleq \emptyset \qquad \mathrm{addr}_{\mathbb{T}}(\mathbf{x}) \triangleq \{\mathbf{x}\} \qquad \mathrm{addr}_{\mathbb{T}}(n[\mathbf{t}]) \triangleq \mathrm{addr}_{\mathbb{T}}(\mathbf{t})$$
$$\mathrm{addr}_{\mathbb{T}}(\mathbf{t}_1 \otimes \mathbf{t}_2) \triangleq \mathrm{addr}_{\mathbb{T}}(\mathbf{t}_1) \uplus \mathrm{addr}_{\mathbb{T}}(\mathbf{t}_2)$$

Tree context application Recall that data splitting in SSL is achieved through abstract allocation (e.g. the transition from Fig. 4.1a to 4.1b). Conversely, data can be combined via abstract deallocation (e.g. the transition from Fig. 4.1b to 4.1a) where the subdata at an abstract address is *compressed* into its counterpart context hole. Data compression is defined in terms of *context application* on logical data, describing the collapsing of abstract data into context holes. For tree data, we define context application $\mathbf{t}_1 \diamond_{\mathbf{x}} \mathbf{t}_2$ in the standard way: it is undefined when $\mathbf{x} \notin \mathrm{addr}_{\mathbb{T}}(\mathbf{t}_1)$; otherwise, it is defined as $\mathbf{t}_1[\mathbf{t}_2/\mathbf{x}]$, denoting the standard substitution of $\mathbf{t}_2$ for $\mathbf{x}$ in $\mathbf{t}_1$, provided that the substitution result is in $\textsc{LData}_{\mathbb{T}}$.

SSL $\mathbb{T}$ Instance (Parameter 5)

Definition 49 (Tree application)**.** Given the set of logical tree data $\textsc{LData}_{\mathbb{T}}$ (Def. 48) and the set of abstract addresses $\textsc{AAdd}$ (Def. 8), the *tree context application* function, $\diamond : \textsc{LData}_{\mathbb{T}} \times \textsc{AAdd} \times \textsc{LData}_{\mathbb{T}} \rightharpoonup \textsc{LData}_{\mathbb{T}}$, is defined inductively over the structure of logical tree data as follows:

$$\varnothing \diamond_{\mathbf{x}} \mathbf{t} \quad \text{undefined} \qquad \mathbf{y} \diamond_{\mathbf{x}} \mathbf{t} \triangleq \begin{cases} \mathbf{t} & \text{if } \mathbf{x} = \mathbf{y} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$n[\mathbf{t}'] \diamond_{\mathbf{x}} \mathbf{t} \triangleq \begin{cases} n[\mathbf{t}''] & \text{if } \mathbf{t}' \diamond_{\mathbf{x}} \mathbf{t} = \mathbf{t}'' \text{ and } n[\mathbf{t}''] \in \textsc{LData}_{\mathbb{T}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

[2]It is straightforward to formalise these restrictions.

$$(\mathbf{t}_1 \otimes \mathbf{t}_2) \diamond_{\mathbf{x}} \mathbf{t} \triangleq \begin{cases} \mathbf{t}' \otimes \mathbf{t}_2 & \text{if } \ \mathbf{t}_1 \diamond_{\mathbf{x}} \mathbf{t}{=}\mathbf{t}' \ \text{ and } \ \mathbf{t}' \otimes \mathbf{t}_2 \in \text{LD}{\small\text{ATA}}_{\mathbb{T}} \\ \mathbf{t}_1 \otimes \mathbf{t}' & \text{if } \ \mathbf{t}_2 \diamond_{\mathbf{x}} \mathbf{t}{=}\mathbf{t}' \ \text{ and } \ \mathbf{t}_1 \otimes \mathbf{t}' \in \text{LD}{\small\text{ATA}}_{\mathbb{T}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Tree operations Recall that abstract libraries allow for the underlying data structure to be manipulated via a set of operations. For the tree library $\mathbb{T}$, we define the operation set to comprise: `r := getFirst(n)`, `r := getRight(n)`, `r := getUp(n)`, `r := newNodeAfter(n)`, `deleteTree(n)`, and `appendChild(n, m)`. We have chosen these operations to demonstrate a wide range of structural manipulations on trees. Each of these operations *faults* if any of the nodes given as parameters are not present in the tree structure. The `r := getFirst(n)` operation returns the identifier of the first child of node `n` in `r` when it exists; it returns `null` if `n` has no children. Similarly, `r := getRight(n)` returns the identifier of the immediate right sibling of node `n` in `r` when it exists; it returns `null` if `n` is the last child of its parent and has no right sibling. The `r := getUp(n)` operation returns the parent identifier for node `n` in `r` when it exists; it returns `null` if `n` is the root node at $\mathcal{R}$. The `r := newNodeAfter(n)` operation creates a new node with a fresh identifier, makes it the right sibling of node `n` and returns this fresh identifier in `r`. The `deleteTree(n)` operation removes the entire subtree identified by `n` from the tree. Finally, the `appendChild(n,m)` operation moves the subtree at `m` to be the last child of node `n`. This operation faults if `n` is a descendant of `m` as it would introduce a cycle and break the tree structure.

SSL $\mathbb{T}$ Instance (Parameter 6)

Definition 50 (Tree operations)**.** The set of *tree operations*, $\mathtt{C}_{\mathbb{T}} \in \text{O}{\small\text{P}}_{\mathbb{T}}$, is defined by the following grammar, for all program variables $\mathtt{r}, \mathtt{n}, \mathtt{m} \in \text{PV}{\small\text{AR}}$ (Def. 1):

$$\begin{aligned} \mathtt{C}_{\mathbb{T}} ::= {} & \mathtt{r := getFirst(n)} \mid \mathtt{r := getRight(n)} \mid \mathtt{r := getUp(n)} \\ & \mid \mathtt{r := newNodeAfter(n)} \mid \mathtt{deleteTree(n)} \mid \mathtt{appendChild(n, m)} \end{aligned}$$

Tree logical values Recall from §3.1 that SSL assumes an abstract library to define a set of *library-specific logical values* that contain program values as well as abstract addresses. Logical values denote the values associated with logical variables. For the tree library $\mathbb{T}$, the logical values are defined as the extension of tree program values (Def. 47) with abstract addresses (Def. 8) and logical tree data (Def. 48). As we demonstrate later, we include logical tree data in the set of logical values to allow for writing expressions that inspect the structure of tree data (see Def. 52).

SSL $\mathbb{T}$ Instance (Parameter 7)

Definition 51 (Tree logical values)**.** Given the set of tree program values $\textsc{PVal}_{\mathbb{T}}$ (Def. 47), the set of abstract addresses $\textsc{AAdd}$ (Def. 8) and the set of tree logical data $\textsc{LData}_{\mathbb{T}}$ (Def. 48), the set of *logical values for trees* is $v \in \textsc{LVal}_{\mathbb{T}} \triangleq \textsc{PVal}_{\mathbb{T}} \cup \textsc{AAdd} \cup \textsc{LData}_{\mathbb{T}}$.

Tree logical expressions Recall that libraries may specify a set of logical expressions in order to assert certain properties about the underlying data. For the tree library $\mathbb{T}$, the logical expressions include logical variables and are defined by a similar grammar to that of logical data (Def. 48).

SSL $\mathbb{T}$ Instance (Parameter 8)

Definition 52 (Tree logical expressions)**.** The set of *logical expressions for trees*, $e \in \textsc{LExp}_{\mathbb{T}}$, is defined by the following grammar, where $\textsc{n}, \textsc{t}, \alpha \in \textsc{LVar}$ (Def. 2):

$$e ::= \varnothing \mid \textsc{t} \mid \alpha \mid \textsc{n}[e] \mid e_1 \otimes e_2$$

Given the set of logical values for trees $\textsc{LVal}_{\mathbb{T}}$ (Def. 51) and the set of logical environments $\textsc{LEnv}\langle\textsc{LVal}_{\mathbb{T}}\rangle$ (Def. 3), the *evaluation function* for tree expressions, $(\!|.|\!)^{(.)}_{\mathbb{T}} : (\textsc{LExp}_{\mathbb{T}} \times \textsc{LEnv}\langle\textsc{LVal}_{\mathbb{T}}\rangle) \rightharpoonup \textsc{LVal}_{\mathbb{T}}$, is defined inductively over the structure of primitive logical expressions as follows, where $\Gamma \in \textsc{LEnv}\langle \textsc{LVal}_{\mathbb{T}} \rangle$:

$$(\!|\varnothing|\!)^{\Gamma}_{\mathbb{T}} = \varnothing \qquad (\!|\textsc{t}|\!)^{\Gamma}_{\mathbb{T}} = \Gamma(\textsc{t}) \qquad (\!|\alpha|\!)^{\Gamma}_{\mathbb{T}} = \begin{cases} \Gamma(\alpha) & \text{if } \Gamma(\alpha) \in \textsc{AAdd} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|\textsc{n}[e]|\!)^{\Gamma}_{\mathbb{T}} = \begin{cases} n[\mathbf{t}] & \text{if } \Gamma(\textsc{n}) = n \wedge (\!|e|\!)^{\Gamma}_{\mathbb{T}} = \mathbf{t} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|e_1 \otimes e_2|\!)^{\Gamma}_{\mathbb{T}} = \begin{cases} \mathbf{t}_1 \otimes \mathbf{t}_2 & \text{if } (\!|e_1|\!)^{\Gamma}_{\mathbb{T}} = \mathbf{t}_1 \wedge (\!|e_2|\!)^{\Gamma}_{\mathbb{T}} = \mathbf{t}_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Tree data assertions Recall that given a library $\mathbb{A}$, the *SSL assertions* for $\mathbb{A}$ comprise heap assertions describing abstract heaps in $\textsc{LHeap}_{\mathbb{A}}$. Heap assertions in turn are defined via *data assertions*, describing the underlying data in $\textsc{LData}_{\mathbb{A}}$. For the tree library $\mathbb{T}$, the tree-specific data assertions comprise assertions to describe empty forests, forest holes (where the associated forest has been split away, leaving behind a hole), singleton forests and composite forests.

SSL $\mathbb{T}$ Instance (Parameter 9)

Definition 53 (Tree data assertions)**.** The set of *tree data assertions*, $\Lambda \in \textsc{LAst}_{\mathbb{T}}$, is defined by the following grammar, where $\alpha, \textsc{n} \in \textsc{LVar}$ (Def. 2) and $\Delta \in \textsc{DAst}_{\mathbb{T}}$ (Def. 19):

$$\Lambda ::= \varnothing \mid \alpha \mid \textsc{n}[\Delta] \mid \Delta_1 \otimes \Delta_2$$

Given the set of logical values for trees $\textsc{LVal}_{\mathbb{T}}$ (Def. 51), the set of logical environments $\textsc{LEnv}\langle \textsc{LVal}_{\mathbb{T}} \rangle$ (Def. 3) and the set of tree logical data $\textsc{LData}_{\mathbb{T}}$ (Def. 48), the *satisfiability relation for tree data assertions*, $|\!|\!\models_{\mathbb{A}}$: $(\textsc{LEnv}\langle \textsc{LVal}_{\mathbb{T}} \rangle \times \textsc{LData}_{\mathbb{T}}) \times \textsc{LAst}_{\mathbb{T}}$, is defined as follows, where $\Gamma \in \textsc{LEnv}\langle \textsc{LVal}_{\mathbb{T}} \rangle$, $\mathbf{t} \in \textsc{LData}_{\mathbb{T}}$, $n \in \textsc{Id}$ (Def. 46 and $\textsc{AAdd}$ denotes the set of abstract addresses (Def. 8):

$$\Gamma, \mathbf{t} \;|\!|\!\models_{\mathbb{A}} \varnothing \qquad \text{iff} \quad \mathbf{t} = \varnothing$$

$$
\begin{array}{lll}
\Gamma, \mathbf{t} \mathrel{||}\models_{\mathbb{A}} \alpha & \text{iff} & \Gamma(\alpha) = \mathbf{t} \wedge \mathbf{t} \in \textsc{AAdd} \\
\Gamma, \mathbf{t} \mathrel{||}\models_{\mathbb{A}} \textsc{n}[\Delta] & \text{iff} & \exists n, \mathbf{t}'.\, \Gamma(\textsc{n}){=}n \wedge \mathbf{t}{=}n[\mathbf{t}'] \wedge \Gamma, \mathbf{t}' \mathrel{|}\models_{\mathbb{A}} \Delta \\
\Gamma, \mathbf{t} \mathrel{||}\models_{\mathbb{A}} \Delta_1 \otimes \Delta_2 & \text{iff} & \exists \mathbf{t}_1, \mathbf{t}_2.\, \mathbf{t}{=}\mathbf{t}_1 \otimes \mathbf{t}_2 \wedge \Gamma, \mathbf{t}_1 \mathrel{|}\models_{\mathbb{A}} \Delta_1 \wedge \Gamma, \mathbf{t}_2 \mathrel{|}\models_{\mathbb{A}} \Delta_2
\end{array}
$$

Library $\mathbb{T}$ specification Recall that in order to specify the behaviour of library operations, we appeal to standard separation logic (SL) assertions such as $P * Q$, as well as an abstract predicate, $\mathsf{vars}(\dots)$, describing the values associated with program variables. As we discussed earlier in §3, rather than extending the SSL assertions with the standard SL assertions and the $\mathsf{vars}(\dots)$ predicate, we view SSL as an *add-on* to SL, and expect the SSL assertions to be incorporated into an SL-based logic that includes the standard connectives such as $P * Q$, as well as assertions for describing the underlying variable store. Here, we combine SSL with the SL-based WLogic presented in §3.2 and extend it to $\textsc{WLogic}_{\mathbb{T}}$. We use $\textsc{WLogic}_{\mathbb{T}}$ to specify the behaviour of library $\mathbb{T}$ operations, and to reason about its client programs. The axiomatic specification of library $\mathbb{T}$ operations is given in Fig. 4.2. Observe that each axiom must preserve the set of abstract addresses present in their footprint. This is evident in the `deleteTree` axiom stipulating that the tree being removed be complete, in that it contains no context holes. This is expressed via the derived assertion $\mathsf{complete}$ defined as follows:

$$\mathsf{complete}(\textsc{t}) \triangleq \textsc{t} \doteq \varnothing \vee (\exists \textsc{c}, \textsc{t}_c, \textsc{t}'.\, \textsc{t} \doteq \textsc{c}[\textsc{t}_c] \otimes \textsc{t}' * \mathsf{complete}(\textsc{t}_c) * \mathsf{complete}(\textsc{t}'))$$

The $\mathsf{complete}(\textsc{t})$ assertion states that the forest $\textsc{t}$ is either empty (the first disjunct), or it is of the form $\textsc{c}[\textsc{t}_c] \otimes \textsc{t}'$, where $\textsc{t}_c$ and $\textsc{t}'$ both satisfy $\mathsf{complete}(\textsc{t}')$. The $\mathsf{complete}(\textsc{t})$ assertion is *pure*, i.e. contains no resource, and merely describes the shape of the forest $\textsc{t}$. Observe that $\mathsf{complete}(\textsc{t})$ ensures that the tree data described by $\textsc{t}$ contains no context holes. Were the tree data in the precondition of `deleteTree(n)` incomplete and contained a context hole, it would be destroyed and its counterpart abstract heap cell could not be connected anywhere. This would in turn yield a malformed heap as the connectivity described by abstract addresses would be broken (i.e. the resulting tree heap will not be well-formed according

$$\{\mathsf{vars}(\mathtt{r}\!:\!-,\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{N}[\mathrm{R}[\beta] \otimes \gamma]\}$$
`r := getFirst(n)`
$$\{\mathsf{vars}(\mathtt{r}\!:\!\mathrm{R},\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{N}[\mathrm{R}[\beta] \otimes \gamma]\}$$

$$\{\mathsf{vars}(\mathtt{r}\!:\!-,\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{N}[\varnothing]\}$$
`r := getFirst(n)`
$$\{\mathsf{vars}(\mathtt{r}\!:\!\mathtt{null},\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{N}[\varnothing]\}$$

$$\{\mathsf{vars}(\mathtt{r}\!:\!-,\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{N}[\beta] \otimes \mathrm{R}[\gamma]\}$$
`r := getRight(n)`
$$\{\mathsf{vars}(\mathtt{r}\!:\!\mathrm{R},\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{N}[\beta] \otimes \mathrm{R}[\gamma]\}$$

$$\{\mathsf{vars}(\mathtt{r}\!:\!-,\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{U}[\beta \otimes \mathrm{N}[\gamma]]\}$$
`r := getRight(n)`
$$\{\mathsf{vars}(\mathtt{r}\!:\!\mathtt{null},\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{U}[\beta \otimes \mathrm{N}[\gamma]]\}$$

$$\{\mathsf{vars}(\mathtt{r}\!:\!-,\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{R}[\beta_1 \otimes \mathrm{N}[\gamma] \otimes \beta_2]\}$$
`r := getUp(n)`
$$\{\mathsf{vars}(\mathtt{r}\!:\!\mathrm{R},\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{R}[\beta_1 \otimes \mathrm{N}[\gamma] \otimes \beta_2]\}$$

$$\{\mathsf{vars}(\mathtt{r}\!:\!-,\mathtt{n}\!:\!\mathrm{N}) * \mathcal{R}_t \mapsto \beta_1 \otimes \mathrm{N}[\gamma] \otimes \beta_2\}$$
`r := getUp(n)`
$$\{\mathsf{vars}(\mathtt{r}\!:\!\mathtt{null},\mathtt{n}\!:\!\mathrm{N}) * \mathcal{R}_t \mapsto \beta_1 \otimes \mathrm{N}[\gamma] \otimes \beta_2\}$$

$$\{\mathsf{vars}(\mathtt{r}\!:\!-,\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{N}[\beta]\}$$
`r := newNodeAfter(n)`
$$\{\exists \mathrm{R}.\ \mathsf{vars}(\mathtt{r}\!:\!\mathrm{R},\mathtt{n}\!:\!\mathrm{N}) * \alpha \mapsto \mathrm{N}[\beta] \otimes \mathrm{R}[\varnothing]\}$$

$$\{\mathsf{vars}(\mathtt{n} : \mathrm{N}) * \alpha \mapsto \mathrm{N}[\mathrm{T}] * \mathsf{complete}(\mathrm{T})\}$$
`deleteTree(n)`
$$\{\mathsf{vars}(\mathtt{n} : \mathrm{N}) * \alpha \mapsto \varnothing\}$$

$$\{\mathsf{vars}(\mathtt{n}\!:\!\mathrm{N},\mathtt{m}\!:\!\mathrm{M}) * \alpha \mapsto \mathrm{N}[\beta] * \gamma \mapsto \mathrm{M}[\mathrm{T}] * \mathsf{complete}(\mathrm{T})\}$$
`appendChild(n, m)`
$$\{\mathsf{vars}(\mathtt{n}\!:\!\mathrm{N},\mathtt{m}\!:\!\mathrm{M}) * \alpha \mapsto \mathrm{N}[\beta \otimes \mathrm{M}[\mathrm{T}]] * \gamma \mapsto \varnothing\}$$

Figure 4.2.: The axiomatic specification of library $\mathbb{T}$ operations

to Def. 14), and would thus render the operation unsafe.

Recall that when $\mathtt{n}{=}\mathrm{N}$ and $\mathtt{m}{=}\mathrm{M}$, the `appendChild(n, m)` operation moves the subtree at M to be the last child of node N, and faults if M is an ancestor of N (otherwise it would introduce a cycle and break the tree structure). To ensure that M is not an ancestor of N, we require the entire subtree at M to be separate from the subtree at N. This is achieved by the $\mathsf{complete}(\mathrm{T})$ assertion and the separating conjunction $*$ in the precondition. That is, the $*$ ensures that the subtree at M is *disjoint* from the subtree at N, and since the forest T underneath M is *complete*, the T does not contain a context hole into which the subtree at N may collapse and render M an ancestor of N.

SSL $\mathbb{T}$ Instance (Parameter 18)

Definition 54 (Tree axioms)**.** The *axioms of tree operation* are given in Fig. 4.2.

4.2. Reasoning about $\mathbb{T}$ Client Programs

We use the WLogic$_{\mathbb{T}}$ program logic to reason about two client programs of $\mathbb{T}$ written in the while language of WLogic$_{\mathbb{T}}$. For brevity, we write `var` $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$ `in {C}` for `var` $\mathtt{x}_1$ `in {var` $\mathtt{x}_2$ `in {...` `var` $\mathtt{x}_n$ `in {C}}}`.

4.2.1. The `getLast(n)` Client Program

The `m := getLast(n)` program in Fig. 4.3 returns the identifier of the last child of node `n` in `m` when it exists, and returns `null` if `n` has no children. To do this, the program traverses the child list of node `n` from the beginning by calling the $\mathbb{T}$ library operation `getFirst(n)`, and repeatedly calling the `getRight(...)` operation to proceed down the list.

The behaviour of the `m := getLast(n)` client program can be specified as follows where (4.1) describes the case when node `n` has a last child, and (4.2) describes the case when `n` has no children.

$$\begin{array}{c}\left\{\mathsf{vars}(\mathtt{n}{:}\textsc{n}, \mathtt{m}{:}-) * \alpha \mapsto \textsc{n}[\textsc{t} \otimes \textsc{m}[\beta]\,] * \mathsf{children}(\textsc{t})\right\} \\ \mathtt{m := getLast(n)} \\ \left\{\mathsf{vars}(\mathtt{n}{:}\textsc{n}, \mathtt{m}{:}\textsc{m}) * \alpha \mapsto \textsc{n}[\textsc{t} \otimes \textsc{m}[\beta]\,] * \mathsf{children}(\textsc{t})\right\}\end{array} \tag{4.1}$$

$$\begin{array}{c}\left\{\mathsf{vars}(\mathtt{n}{:}\textsc{n}, \mathtt{m}{:}-) * \alpha \mapsto \textsc{n}[\varnothing]\right\} \\ \mathtt{m := getLast(n)} \\ \left\{\mathsf{vars}(\mathtt{n}{:}\textsc{n}, \mathtt{m}{:}\mathtt{null}) * \alpha \mapsto \textsc{n}[\varnothing]\right\}\end{array} \tag{4.2}$$

where

$$\mathsf{children}(\textsc{t}) \triangleq \textsc{t} \doteq \varnothing \vee (\exists \textsc{c}, \textsc{t}_c, \textsc{t}'.\ \textsc{t} \doteq \textsc{c}[\textsc{t}_c] \otimes \textsc{t}' * \mathsf{children}(\textsc{t}'))$$

When $\mathtt{n}{=}\textsc{n}$ and $\textsc{n}$ has no children (4.2), the behaviour of the program is simple and its footprint is limited to the (empty) forest underneath $\textsc{n}$. On the other hand, when the forest underneath $\textsc{n}$ is not empty, the footprint comprises the *child list* of $\textsc{n}$. For a given tree of the form $n[c_1[\mathbf{t}_1] \otimes c_2[\mathbf{t}_2] \otimes \ldots \otimes c_m[\mathbf{t}_m]]$, the child list of n comprises the nodes $c_1, c_2, \ldots, c_m$ but not their respective child forests $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_m$. When $\mathtt{n}{=}\textsc{n}$, the `getLast(n)` operation inspects the immediate children of $\textsc{n}$ and its footprint is thus the child list of $\textsc{n}$. As such, we must ensure that

```
 1. m := getLast(n){
 2.   var curr, next in {
 3.     curr := getFirst(n);
 4.     if(curr != null){
 5.       next := getRight(curr);
 6.       while(next != null){
 7.         curr := next;
 8.         next := getRight(curr);
 9.     } }
10.     m := curr;
11.   }
12. }
```

Figure 4.3.: The `getLast(n)` client program

the precondition in (4.1) contains the resources for the child list of N. In particular, we must ensure that child list of N is *complete* and contains no context holes. This is captured by children(T), asserting that the forest T underneath N is either empty (the first disjunct), or it is of the form $C[T_c] \otimes T'$, where T' itself satisfies children(T'). Note that children(T) implies that there are no context holes at the top level of T (i.e. the child list of N contains no context holes) as required. As with the complete(T) assertion, the children(T) is *pure* (i.e. contains no resource).

A proof sketch of specification (4.1) is given in Figs. 4.4-4.5. As before, at each proof point we highlight the effect of the preceding operation when applicable. For instance, after the assignment of line 8 the value of `curr` is updated to C, whereas the while statement of line 17 has no effect. At various proof points we use abstract (de)allocation (Lemma 1) to rewrite the tree resources into the shape required by the axioms of tree operations (Fig. 4.2). For instance, prior to applying the `getFirst(n)` axiom at line 8, we use abstract deallocation at line 7 to split off part of the forest under N, and thus get the subtree at N into the shape required by the `getFirst(n)` axioms.

4.2.2. The `moveChildren(n, m)` Client Program

The `moveChildren(n,m)` program in Fig. 4.6 removes the child forest of node `n` and appends it to the end of the child forest of node `m`. To do this, the program traverses the child list of node `n` from the beginning

```
1.  $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-) * \alpha \mapsto \mathrm{N}[\mathrm{T} \otimes \mathrm{M}[\beta]] * \mathsf{children}(\mathrm{T})\}$
2.  m := getLast(n){
3.  $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-) * \alpha \mapsto \mathrm{N}[\mathrm{T} \otimes \mathrm{M}[\beta]] * \mathsf{children}(\mathrm{T})\}$
4.    var curr, next in {
5.  $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}-, \mathtt{next}{:}-) * \alpha \mapsto \mathrm{N}[\mathrm{T} \otimes \mathrm{M}[\beta]] * \mathsf{children}(\mathrm{T})\}$
    // Unwrap the definition of children
6.  $\left\{\begin{array}{l} \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}-, \mathtt{next}{:}-) * \big((\mathrm{T} \doteq \varnothing * \alpha \mapsto \mathrm{N}[\varnothing \otimes \mathrm{M}[\beta]]) \\ \lor\ (\exists \mathrm{L}, \mathrm{T}_l, \mathrm{T}'.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] \otimes \mathrm{T}' * \mathsf{children}(\mathrm{T}') * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l] \otimes \mathrm{T}' \otimes \mathrm{M}[\beta]])\big) \end{array}\right\}$
    // Abstract allocation at $\gamma$ by Lemma 1
7.  $\left\{\begin{array}{l} \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}-, \mathtt{next}{:}-) * \exists \gamma.\ \big((\mathrm{T} \doteq \varnothing * \alpha \mapsto \mathrm{N}[\mathrm{M}[\beta] \otimes \gamma] * \gamma \mapsto \varnothing) \\ \lor\ (\exists \mathrm{L}, \mathrm{T}_l, \mathrm{T}'.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] \otimes \mathrm{T}' * \mathsf{children}(\mathrm{T}') * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l] \otimes \gamma] * \gamma \mapsto \mathrm{T}' \otimes \mathrm{M}[\beta])\big) \end{array}\right\}$
8.      curr := getFirst(n);
9.  $\left\{\begin{array}{l} \exists \mathrm{C}.\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}-) * \exists \gamma. \big((\mathrm{T} \doteq \varnothing * \mathrm{C} \doteq \mathrm{M} * \alpha \mapsto \mathrm{N}[\mathrm{M}[\beta] \otimes \gamma] * \gamma \mapsto \varnothing) \\ \lor (\exists \mathrm{L}, \mathrm{T}_l, \mathrm{T}'. \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] \otimes \mathrm{T}' * \mathsf{children}(\mathrm{T}') * \mathrm{C} \doteq \mathrm{L} * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l] \otimes \gamma] * \gamma \mapsto \mathrm{T}' \otimes \mathrm{M}[\beta])\big) \end{array}\right\}$
    // Abstract deallocation at $\gamma$ by Lemma 1
10. $\left\{\begin{array}{l} \exists \mathrm{C}.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}-) * \big((\mathrm{C} \doteq \mathrm{M} * \alpha \mapsto \mathrm{N}[\mathrm{T} \otimes \mathrm{M}[\beta]] * \mathsf{children}(\mathrm{T})) \\ \lor\ (\exists \mathrm{L}, \mathrm{T}_l, \mathrm{T}'.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] \otimes \mathrm{T}' * \mathsf{children}(\mathrm{T}') * \mathrm{C} \doteq \mathrm{L} * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l] \otimes \mathrm{T}' \otimes \mathrm{M}[\beta]])\big) \end{array}\right\}$
    // Unwrap the definition of children in the second disjunct
11. $\left\{\begin{array}{l} \exists \mathrm{C}.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}-) * \big((\mathrm{C} \doteq \mathrm{M} * \alpha \mapsto \mathrm{N}[\mathrm{T} \otimes \mathrm{M}[\beta]] * \mathsf{children}(\mathrm{T})) \\ \quad \lor (\exists \mathrm{L}, \mathrm{T}_l.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l]) * \mathrm{C} \doteq \mathrm{L} * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l] \otimes \mathrm{M}[\beta]]) \\ \quad \lor (\exists \mathrm{L}, \mathrm{T}_l, \mathrm{O}, \mathrm{T}_o, \mathrm{T}'.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] \otimes \mathrm{O}[\mathrm{T}_o] \otimes \mathrm{T}' * \mathsf{children}(\mathrm{T}') * \mathrm{C} \doteq \mathrm{L} \\ \qquad * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l] \otimes \mathrm{O}[\mathrm{T}_o] \otimes \mathrm{T}' \otimes \mathrm{M}[\beta]])\big) \end{array}\right\}$
    // Abstract allocation at $\gamma, \delta, \epsilon$ by Lemma 1
12. $\left\{\begin{array}{l} \exists \mathrm{C}.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}-) * \big((\mathrm{C} \doteq \mathrm{M} * \alpha \mapsto \mathrm{N}[\mathrm{T} \otimes \mathrm{M}[\beta]] * \mathsf{children}(\mathrm{T})) \\ \quad \lor (\exists \mathrm{L}, \mathrm{T}_l, \gamma, \delta.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] * \mathrm{C} \doteq \mathrm{L} * \alpha \mapsto \mathrm{N}[\gamma] * \gamma \mapsto \mathrm{L}[\delta] \otimes \mathrm{M}[\beta] * \delta \mapsto \mathrm{T}_l) \\ \quad \lor (\exists \mathrm{L}, \mathrm{T}_l, \mathrm{O}, \mathrm{T}_o, \mathrm{T}', \gamma, \delta, \epsilon.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] \otimes \mathrm{O}[\mathrm{T}_o] \otimes \mathrm{T}' * \mathsf{children}(\mathrm{T}') * \mathrm{C} \doteq \mathrm{L} \\ \qquad * \alpha \mapsto \mathrm{N}[\gamma \otimes \mathrm{T}' \otimes \mathrm{M}[\beta]] * \gamma \mapsto \mathrm{L}[\delta] \otimes \mathrm{O}[\epsilon] * \delta \mapsto \mathrm{T}_l * \epsilon \mapsto \mathrm{T}_o)\big) \end{array}\right\}$
13.     if(curr != null){
14.       next := getRight(curr)
15. $\left\{\begin{array}{l} \exists \mathrm{C}, \mathrm{R}.\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}\mathrm{R}) \\ * \big((\mathrm{C} \doteq \mathrm{M} * \mathrm{R} \doteq \mathtt{null} * \alpha \mapsto \mathrm{N}[\mathrm{T} \otimes \mathrm{M}[\beta]] * \mathsf{children}(\mathrm{T})) \\ \quad \lor (\exists \mathrm{L}, \mathrm{T}_l, \gamma, \delta.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] * \mathrm{C} \doteq \mathrm{L} * \mathrm{R} \doteq \mathrm{M} * \alpha \mapsto \mathrm{N}[\gamma] * \gamma \mapsto \mathrm{L}[\delta] \otimes \mathrm{M}[\beta] * \delta \mapsto \mathrm{T}_l) \\ \quad \lor (\exists \mathrm{L}, \mathrm{T}_l, \mathrm{O}, \mathrm{T}_o, \mathrm{T}', \gamma, \delta, \epsilon.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] \otimes \mathrm{O}[\mathrm{T}_o] \otimes \mathrm{T}' * \mathsf{children}(\mathrm{T}') * \mathrm{C} \doteq \mathrm{L} * \mathrm{R} \doteq \mathrm{O} \\ \qquad * \alpha \mapsto \mathrm{N}[\gamma \otimes \mathrm{T}' \otimes \mathrm{M}[\beta]] * \gamma \mapsto \mathrm{L}[\delta] \otimes \mathrm{O}[\epsilon] * \delta \mapsto \mathrm{T}_l * \epsilon \mapsto \mathrm{T}_o)\big) \end{array}\right\}$
    // Abstract deallocation at $\gamma, \delta, \epsilon$ by Lemma 1
16. $\left\{\begin{array}{l} \exists \mathrm{C}, \mathrm{R}.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}\mathrm{R}) \\ * \big((\mathrm{C} \doteq \mathrm{M} * \mathrm{R} \doteq \mathtt{null} * \alpha \mapsto \mathrm{N}[\mathrm{T} \otimes \mathrm{M}[\beta]] * \mathsf{children}(\mathrm{T})) \\ \quad \lor (\exists \mathrm{L}, \mathrm{T}_l.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] * \mathrm{C} \doteq \mathrm{L} * \mathrm{R} \doteq \mathrm{M} * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l] \otimes \mathrm{M}[\beta]]) \\ \quad \lor (\exists \mathrm{L}, \mathrm{T}_l, \mathrm{O}, \mathrm{T}_o, \mathrm{T}'.\ \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] \otimes \mathrm{O}[\mathrm{T}_o] \otimes \mathrm{T}' * \mathsf{children}(\mathrm{T}') * \mathrm{C} \doteq \mathrm{L} * \mathrm{R} \doteq \mathrm{O} \\ \qquad\qquad * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l] \otimes \mathrm{O}[\mathrm{T}_o] \otimes \mathrm{T}' \otimes \mathrm{M}[\beta]])\big) \end{array}\right\}$
```

Figure 4.4.: A proof sketch of `getLast(n)` (continued in Fig. 4.5)

15. $$\left\{\begin{array}{l} \exists \mathrm{C},\mathrm{R}.\, \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}\mathrm{R}) \\ *\Big((\mathrm{C}\doteq\mathrm{M} * \mathrm{R}\doteq\mathtt{null} * \alpha \mapsto \mathrm{N}[\mathrm{T}\otimes\mathrm{M}[\beta]\,] * \mathsf{children}(\mathrm{T})) \\ \quad \vee(\exists \mathrm{L},\mathrm{T}_l.\ \mathrm{T}\doteq\mathrm{L}[\mathrm{T}_l] * \mathrm{C}\doteq\mathrm{L} * \mathrm{R}\doteq\mathrm{M} * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{M}[\beta]\,]) \\ \quad \vee(\exists \mathrm{L},\mathrm{T}_l,\mathrm{O},\mathrm{T}_o,\mathrm{T}'.\ \mathrm{T}\doteq\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}' * \mathsf{children}(\mathrm{T}') * \mathrm{C}\doteq\mathrm{L} * \mathrm{R}\doteq\mathrm{O} \\ \qquad\qquad\qquad * \ \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}'\otimes\mathrm{M}[\beta]\,])\Big) \end{array}\right\}$$

// Generalise the definition of T

16. $$\left\{\begin{array}{l} \exists \mathrm{C},\mathrm{R}.\, \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}\mathrm{R}) \\ *\Big((\mathrm{C}\doteq\mathrm{M} * \mathrm{R}\doteq\mathtt{null} * \alpha \mapsto \mathrm{N}[\mathrm{T}\otimes\mathrm{M}[\beta]\,] * \mathsf{children}(\mathrm{T})) \\ \quad \vee(\exists \mathrm{L},\mathrm{T}_l,\mathrm{T}'.\,\mathrm{T}\doteq\mathrm{T}'\otimes\mathrm{L}[\mathrm{T}_l] * \mathsf{children}(\mathrm{T}')*\mathrm{C}\doteq\mathrm{L}*\mathrm{R}\doteq\mathrm{M}*\alpha \mapsto \mathrm{N}[\mathrm{T}'\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{M}[\beta]\,]) \\ \quad \vee(\exists \mathrm{L},\mathrm{T}_l,\mathrm{O},\mathrm{T}_o,\mathrm{T}_1,\mathrm{T}_2.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2 * \mathrm{C}\doteq\mathrm{L} * \mathrm{R}\doteq\mathrm{O} \\ \qquad * \mathsf{children}(\mathrm{T}_1) * \mathsf{children}(\mathrm{T}_2) * \alpha \mapsto \mathrm{N}[\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2\otimes\mathrm{M}[\beta]\,])\Big) \end{array}\right\}$$

17. `        while(next != null){`

18. $$\left\{\begin{array}{l} \exists \mathrm{C},\mathrm{R}.\, \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}\mathrm{R}) \\ *\Big((\exists \mathrm{L},\mathrm{T}_l,\mathrm{T}'.\,\mathrm{T}\doteq\mathrm{T}'\otimes\mathrm{L}[\mathrm{T}_l] * \mathsf{children}(\mathrm{T}')*\mathrm{C}\doteq\mathrm{L}*\mathrm{R}\doteq\mathrm{M}*\alpha \mapsto \mathrm{N}[\mathrm{T}'\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{M}[\beta]\,]) \\ \quad \vee(\exists \mathrm{L},\mathrm{T}_l,\mathrm{O},\mathrm{T}_o,\mathrm{T}_1,\mathrm{T}_2.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2 * \mathrm{C}\doteq\mathrm{L} * \mathrm{R}\doteq\mathrm{O} \\ \qquad * \mathsf{children}(\mathrm{T}_1) * \mathsf{children}(\mathrm{T}_2) * \alpha \mapsto \mathrm{N}[\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2\otimes\mathrm{M}[\beta]\,])\Big) \end{array}\right\}$$

19. `            curr := next;`

20. $$\left\{\begin{array}{l} \exists \mathrm{C}.\, \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}\mathrm{C}) \\ *\Big((\exists \mathrm{L},\mathrm{T}_l,\mathrm{T}'.\,\mathrm{T}\doteq\mathrm{T}'\otimes\mathrm{L}[\mathrm{T}_l] * \mathsf{children}(\mathrm{T}')*\mathrm{C}\doteq\mathrm{M}*\alpha \mapsto \mathrm{N}[\mathrm{T}'\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{M}[\beta]\,]) \\ \ \vee(\exists \mathrm{L},\mathrm{T}_l,\mathrm{O},\mathrm{T}_o,\mathrm{T}_1,\mathrm{T}_2.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2 * \mathrm{C}\doteq\mathrm{O} \\ \qquad * \mathsf{children}(\mathrm{T}_1) * \mathsf{children}(\mathrm{T}_2) * \alpha \mapsto \mathrm{N}[\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2\otimes\mathrm{M}[\beta]\,])\Big) \end{array}\right\}$$

// Wrap the definition of children

21. $$\left\{\begin{array}{l} \exists \mathrm{C}.\, \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}\mathrm{C}) * \Big((\mathrm{C}\doteq\mathrm{M}*\alpha \mapsto \mathrm{N}[\mathrm{T}\otimes\mathrm{M}[\beta]\,] * \mathsf{children}(\mathrm{T})) \\ \quad \vee(\exists \mathrm{L},\mathrm{T}_l,\mathrm{T}_1,\mathrm{T}_2.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{T}_2 * \mathrm{C}\doteq\mathrm{L} \\ \qquad * \ \mathsf{children}(\mathrm{T}_1) * \mathsf{children}(\mathrm{T}_2) * \alpha \mapsto \mathrm{N}[\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{T}_2\otimes\mathrm{M}[\beta]\,])\Big) \end{array}\right\}$$

// Take similar steps as those in 10-12

22. $$\left\{\begin{array}{l} \exists \mathrm{C}.\, \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}\mathrm{C}) * \Big((\mathrm{C}\doteq\mathrm{M} * \alpha \mapsto \mathrm{N}[\mathrm{T}\otimes\mathrm{M}[\beta]\,] * \mathsf{children}(\mathrm{T})) \\ \quad \vee(\exists \mathrm{L},\mathrm{T}_l,\mathrm{T}',\gamma,\delta.\ \mathrm{T}\doteq\mathrm{T}'\otimes\mathrm{L}[\mathrm{T}_l] * \mathrm{C}\doteq\mathrm{L} * \alpha \mapsto \mathrm{N}[\mathrm{T}'\otimes\gamma] * \mathsf{children}(\mathrm{T}') \\ \qquad\qquad\qquad * \ \gamma \mapsto \mathrm{L}[\delta]\otimes\mathrm{M}[\beta] * \delta \mapsto \mathrm{T}_l) \\ \quad \vee(\exists \mathrm{L},\mathrm{T}_l,\mathrm{O},\mathrm{T}_o,\mathrm{T}_1,\mathrm{T}_2,\gamma,\delta,\epsilon.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2 * \mathrm{C}\doteq\mathrm{L} \\ \qquad\quad * \ \mathsf{children}(\mathrm{T}_1) * \mathsf{children}(\mathrm{T}_2) * \alpha \mapsto \mathrm{N}[\mathrm{T}_1\otimes\gamma\otimes\mathrm{T}_2\otimes\mathrm{M}[\beta]\,] \\ \qquad\qquad\qquad * \ \gamma \mapsto \mathrm{L}[\delta]\otimes\mathrm{O}[\epsilon] * \delta \mapsto \mathrm{T}_l * \epsilon \mapsto \mathrm{T}_o)\Big) \end{array}\right\}$$

23. `            next := getRight(curr);`

// Abstractly deallocate γ, δ, ϵ by Lemma 1 as in line 15

24. $$\left\{\begin{array}{l} \exists \mathrm{C},\mathrm{R}.\, \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}-, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}\mathrm{R}) \\ *\Big((\mathrm{C}\doteq\mathrm{M} * \mathrm{R}\doteq\mathtt{null} * \alpha \mapsto \mathrm{N}[\mathrm{T}\otimes\mathrm{M}[\beta]\,] * \mathsf{children}(\mathrm{T})) \\ \quad \vee(\exists \mathrm{L},\mathrm{T}_l,\mathrm{T}'.\,\mathrm{T}\doteq\mathrm{T}'\otimes\mathrm{L}[\mathrm{T}_l] * \mathrm{C}\doteq\mathrm{L}*\mathrm{R}\doteq\mathrm{M}*\alpha \mapsto \mathrm{N}[\mathrm{T}'\otimes\mathrm{L}[\delta]\otimes\mathrm{M}[\beta]\,]*\mathsf{children}(\mathrm{T}')) \\ \quad \vee(\exists \mathrm{L},\mathrm{T}_l,\mathrm{O},\mathrm{T}_o,\mathrm{T}_1,\mathrm{T}_2.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2 * \mathrm{C}\doteq\mathrm{L} * \mathrm{R}\doteq\mathrm{O} \\ \qquad\quad * \mathsf{children}(\mathrm{T}_1) * \mathsf{children}(\mathrm{T}_2) * \alpha \mapsto \mathrm{N}[\mathrm{T}_1\otimes\mathrm{L}[\mathrm{T}_l]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2\otimes\mathrm{M}[\beta]\,])\Big) \end{array}\right\}$$

25. `      } }`

26. $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}-,\mathtt{curr}{:}\mathrm{M},\mathtt{next}{:}\mathtt{null}) * \alpha \mapsto \mathrm{N}[\mathrm{T}\otimes\mathrm{M}[\beta]\,] * \mathsf{children}(\mathrm{T})\}$

27. `      m := curr;`

28. `    }` $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M},\mathtt{curr}{:}\mathrm{M},\mathtt{next}{:}\mathtt{null}) * \alpha \mapsto \mathrm{N}[\mathrm{T}\otimes\mathrm{M}[\beta]\,] * \mathsf{children}(\mathrm{T})\}$

29. `}` $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M}) * \alpha \mapsto \mathrm{N}[\mathrm{T}\otimes\mathrm{M}[\beta]\,] * \mathsf{children}(\mathrm{T})\}$

Figure 4.5.: A proof sketch of `getLast(n)` (continued from Fig. 4.4)

```
moveChildren(n, m){
  var curr, next in {
    curr := getFirst(n);
    while(curr != null){
      next := getRight(curr);
      appendChild(m, curr);
      curr := next;
    }
  }
}
```

Figure 4.6.: The `moveChildren(n, m)` client program

by calling the $\mathbb{T}$ library operation `getFirst(n)`. At each iteration, the program records the child node to the right of the current (`curr`) node in `next` by calling the `getRight(curr)` operation, and subsequently appends the current child (`curr`) to the end of the child list of `m` by calling the `appendChild(m, curr)` library operation.

In order to preserve the tree structure and avoid cycles, we must ensure that `m` is not a descendant of `n` (or its children). That is, the program must fault if moving the child forest of `n` breaks the tree structure by introducing a cycle. However, since the program manipulates the tree structure via library operations only, and the library operations in turn preserve the tree structure, the program need not enforce this structure-preservation explicitly. More concretely, since `appendChild(m, curr)` faults if `m` is a descendant of `curr`, the well-formedness of the tree structure is ensured at each iteration when moving the current node by calling `appendChild(m, curr)`. We thus specify the behaviour of `moveChildren(n, m)` as follows:

$$\left\{\mathsf{vars}(\texttt{n}{:}\textsc{n}, \texttt{m}{:}\textsc{m}) * \alpha \mapsto \textsc{n}[\textsc{t}] * \beta \mapsto \textsc{m}[\gamma] * \mathsf{complete}(\textsc{t})\right\}$$
$$\texttt{moveChildren(n, m)}$$
$$\left\{\mathsf{vars}(\texttt{n}{:}\textsc{n}, \texttt{m}{:}\textsc{m}) * \alpha \mapsto \textsc{n}[\varnothing] * \beta \mapsto \textsc{m}[\gamma \otimes \textsc{t}] * \mathsf{complete}(\textsc{t})\right\}$$

When `n`=N and `m`=M, to ensure that M is not a descendant of N, we require the entire subtree at N to be separate from the subtree at M. As with the specification of `appendChild` operation in Fig. 4.2, this is achieved by the separating conjunction $*$ and the $\mathsf{complete}(\textsc{t})$ assertion in the precondition. That is, the $*$ ensures that the entire subtree at N is *disjoint* from the subtree at M, and since the forest T underneath N is *complete*, the T does not contain a context hole into which the subtree at M may collapse and render M a descendant of N.

A proof sketch of `moveChildren(n, m)` is given in Figs. 4.7-4.8. As before, we use abstract (de)allocation (Lemma 1) at various proof points to rewrite the tree resources into the shape required by the axioms of tree operations (Fig. 4.2).

```
1.  $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}) * \alpha \mapsto \mathrm{N}[\mathrm{T}] * \beta \mapsto \mathrm{M}[\gamma] * \mathsf{complete}(\mathrm{T})\}$
2.  moveChildren(n, m){
3.  $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}) * \alpha \mapsto \mathrm{N}[\mathrm{T}] * \beta \mapsto \mathrm{M}[\gamma] * \mathsf{complete}(\mathrm{T})\}$
4.    var curr, next in {
5.  $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}, \mathtt{curr}{:}-, \mathtt{next}{:}-) * \alpha \mapsto \mathrm{N}[\mathrm{T}] * \beta \mapsto \mathrm{M}[\gamma] * \mathsf{complete}(\mathrm{T})\}$
    // Unfold the definition of $\mathsf{complete}(\mathrm{T})$.
6.  $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}, \mathtt{curr}{:}-, \mathtt{next}{:}-) * \big((\mathrm{T} \doteq \varnothing * \alpha \mapsto \mathrm{N}[\varnothing]) \lor$
    $(\exists \mathrm{N}, \mathrm{T}_n, \mathrm{T}'. \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] \otimes \mathrm{T}' * \mathsf{complete}(\mathrm{T}_l) * \mathsf{complete}(\mathrm{T}') * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l] \otimes \mathrm{T}'])\big) * \beta \mapsto \mathrm{M}[\gamma]\}$
    // Abstract allocation at $\delta$ by Lemma 1
7.  $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}, \mathtt{curr}{:}-, \mathtt{next}{:}-) * \exists \delta. \big((\mathrm{T} \doteq \varnothing * \alpha \mapsto \mathrm{N}[\varnothing]) \lor$
    $(\exists \mathrm{L}, \mathrm{T}_l, \mathrm{T}'. \mathrm{T} \doteq \mathrm{L}[\mathrm{T}_l] \otimes \mathrm{T}' * \mathsf{complete}(\mathrm{T}_l) * \mathsf{complete}(\mathrm{T}') * \alpha \mapsto \mathrm{N}[\mathrm{L}[\mathrm{T}_l] \otimes \delta] * \delta \mapsto \mathrm{T}')\big)$
    $* \beta \mapsto \mathrm{M}[\gamma]\}$
8.      curr := getFirst(n);
9.  $\{\exists \mathrm{C}. \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}-) * \big((\mathrm{T} \doteq \varnothing * \mathrm{C} = \mathtt{null} * \alpha \mapsto \mathrm{N}[\varnothing]) \lor$
    $(\exists \mathrm{T}_c, \mathrm{T}', \delta. \mathrm{T} \doteq \mathrm{C}[\mathrm{T}_c] \otimes \mathrm{T}' * \mathsf{complete}(\mathrm{T}_l) * \mathsf{complete}(\mathrm{T}') * \alpha \mapsto \mathrm{N}[\mathrm{C}[\mathrm{T}_c] \otimes \delta] * \delta \mapsto \mathrm{T}')\big)$
    $* \beta \mapsto \mathrm{M}[\gamma]\}$
    // Abstract deallocation at $\delta$ by Lemma 1
10. $\{\exists \mathrm{C}. \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}-) * \big((\mathrm{T} \doteq \varnothing * \mathrm{C} = \mathtt{null} * \alpha \mapsto \mathrm{N}[\varnothing]) \lor$
    $(\exists \mathrm{C}, \mathrm{T}_c, \mathrm{T}'. \mathrm{T} \doteq \mathrm{C}[\mathrm{T}_c] \otimes \mathrm{T}' * \mathsf{complete}(\mathrm{T}_c) * \mathsf{complete}(\mathrm{T}') * \alpha \mapsto \mathrm{N}[\mathrm{C}[\mathrm{T}_c] \otimes \mathrm{T}'])\big) * \beta \mapsto \mathrm{M}[\gamma]\}$
11. $\{\exists \mathrm{C}. \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}-)$
    $* \big((\mathrm{C} = \mathtt{null} * \alpha \mapsto \mathrm{N}[\varnothing] * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}])$
      $\lor (\exists \mathrm{T}_c, \mathrm{T}_1, \mathrm{T}_2. \mathrm{T} \doteq \mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] \otimes \mathrm{T}_2 * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}_1]$
        $* \mathsf{complete}(\mathrm{T}_c) * \mathsf{complete}(\mathrm{T}_2) * \alpha \mapsto \mathrm{N}[\mathrm{C}[\mathrm{T}_c] \otimes \mathrm{T}_2])\big)\}$
12.     while(curr != null){
13. $\{\exists \mathrm{C}. \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}-) * \exists \mathrm{T}_c, \mathrm{T}_1, \mathrm{T}_2. \mathrm{T} \doteq \mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] \otimes \mathrm{T}_2$
    $* \mathsf{complete}(\mathrm{T}_c) * \mathsf{complete}(\mathrm{T}_2) * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}_1] * \alpha \mapsto \mathrm{N}[\mathrm{C}[\mathrm{T}_c] \otimes \mathrm{T}_2]\}$
    // Unfold the definition of $\mathsf{complete}(\mathrm{T}_2)$.
14. $\{\exists \mathrm{C}, \mathrm{T}_c, \mathrm{T}_1. \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}-) * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}_1]$
    $* \big((\mathrm{T} \doteq \mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] * \mathsf{complete}(\mathrm{T}_c) * \alpha \mapsto \mathrm{N}[\mathrm{C}[\mathrm{T}_c]])$
      $\lor (\exists \mathrm{O}, \mathrm{T}_o, \mathrm{T}_2. \mathrm{T} \doteq \mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] \otimes \mathrm{O}[\mathrm{T}_o] \otimes \mathrm{T}_2 * \mathsf{complete}(\mathrm{T}_c)$
        $* \mathsf{complete}(\mathrm{T}_o) * \mathsf{complete}(\mathrm{T}_2) * \alpha \mapsto \mathrm{N}[\mathrm{C}[\mathrm{T}_c] \otimes \mathrm{O}[\mathrm{T}_o] \otimes \mathrm{T}_2])\big)\}$
    // Abstract allocation at $\gamma, \delta, \epsilon$ by Lemma 1
15. $\{\exists \mathrm{C}, \mathrm{T}_c, \mathrm{T}_1, \delta, \epsilon. \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}-) * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}_1]$
    $* \big((\mathrm{T} \doteq \mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] * \mathsf{complete}(\mathrm{T}_c) * \alpha \mapsto \mathrm{N}[\delta \otimes \mathrm{C}[\epsilon]] * \delta \mapsto \varnothing * \epsilon \mapsto \mathrm{T}_c)$
      $\lor (\exists \mathrm{O}, \mathrm{T}_o, \mathrm{T}_2, \zeta. \mathrm{T} \doteq \mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] \otimes \mathrm{O}[\mathrm{T}_o] \otimes \mathrm{T}_2 * \mathsf{complete}(\mathrm{T}_c) * \alpha \mapsto \mathrm{N}[\delta \otimes \mathrm{T}_2]$
        $* \mathsf{complete}(\mathrm{T}_o) * \mathsf{complete}(\mathrm{T}_2) * \delta \mapsto \mathrm{C}[\epsilon] \otimes \mathrm{O}[\zeta] * \epsilon \mapsto \mathrm{T}_c * \zeta \mapsto \mathrm{T}_o)\big)\}$
16.       next := getRight(curr)
17. $\{\exists \mathrm{C}, \mathrm{T}_c, \mathrm{T}_1, \delta, \epsilon, \mathrm{R}. \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{m}{:}\mathrm{M}, \mathtt{curr}{:}\mathrm{C}, \mathtt{next}{:}\mathrm{R}) * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}_1]$
    $* \big((\mathrm{T} \doteq \mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] * \mathsf{complete}(\mathrm{T}_c) * \mathrm{R} \doteq \mathtt{null} * \alpha \mapsto \mathrm{N}[\delta \otimes \mathrm{C}[\epsilon]] * \delta \mapsto \varnothing * \epsilon \mapsto \mathrm{T}_c)$
    $\lor (\exists \mathrm{O}, \mathrm{T}_o, \mathrm{T}_2, \zeta. \mathrm{T} \doteq \mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] \otimes \mathrm{O}[\mathrm{T}_o] \otimes \mathrm{T}_2 * \mathsf{complete}(\mathrm{T}_c) * \mathrm{R} \doteq \mathrm{O} * \alpha \mapsto \mathrm{N}[\delta \otimes \mathrm{T}_2]$
          $* \mathsf{complete}(\mathrm{T}_o) * \mathsf{complete}(\mathrm{T}_2) * \delta \mapsto \mathrm{C}[\epsilon] \otimes \mathrm{O}[\zeta] * \epsilon \mapsto \mathrm{T}_c * \zeta \mapsto \mathrm{T}_o)\big)\}$
```

Figure 4.7.: Proof sketch of `moveChildren(n,m)` (continued in Fig. 4.8)

17. $$\left\{\begin{array}{l} \exists \mathrm{C},\mathrm{T}_c,\mathrm{T}_1,\delta,\epsilon,\mathrm{R}.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M},\mathtt{curr}{:}\mathrm{C},\mathtt{next}{:}\mathrm{R}) * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}_1] \\ *\Big((\mathrm{T}\doteq\mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] * \mathsf{complete}(\mathrm{T}_c) * \mathrm{R}\doteq\mathtt{null} * \alpha \mapsto \mathrm{N}[\delta \otimes \mathrm{C}[\epsilon]\,] * \delta\mapsto\varnothing * \epsilon\mapsto\mathrm{T}_c) \\ \quad \vee(\exists \mathrm{O},\mathrm{T}_o,\mathrm{T}_2,\zeta.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{C}[\mathrm{T}_c]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2*\mathsf{complete}(\mathrm{T}_c)*\mathrm{R}\doteq\mathrm{O}*\alpha\mapsto\mathrm{N}[\delta\otimes\mathrm{T}_2] \\ \qquad\qquad *\mathsf{complete}(\mathrm{T}_o)*\mathsf{complete}(\mathrm{T}_2) * \ \delta\mapsto\mathrm{C}[\epsilon]\otimes\mathrm{O}[\zeta] * \epsilon\mapsto\mathrm{T}_c * \zeta\mapsto\mathrm{T}_o)\Big) \end{array}\right\}$$

// Abstract deallocation at δ, ϵ, ζ by Lemma 1

18. $$\left\{\begin{array}{l} \exists \mathrm{C},\mathrm{T}_c,\mathrm{T}_1,\mathrm{R}.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M},\mathtt{curr}{:}\mathrm{C},\mathtt{next}{:}\mathrm{R}) * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}_1] \\ *\Big((\mathrm{T}\doteq\mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] * \mathsf{complete}(\mathrm{T}_c) * \mathrm{R}\doteq\mathtt{null} * \alpha \mapsto \mathrm{N}[\mathrm{C}[\mathrm{T}_c]\,]) \\ \quad \vee(\exists \mathrm{O},\mathrm{T}_o,\mathrm{T}_2.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{C}[\mathrm{T}_c]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2*\mathsf{complete}(\mathrm{T}_c)*\mathrm{R}\doteq\mathrm{O} \\ \qquad *\mathsf{complete}(\mathrm{T}_o)*\mathsf{complete}(\mathrm{T}_2) * \alpha\mapsto\mathrm{N}[\mathrm{C}[\mathrm{T}_c]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2\,])\Big) \end{array}\right\}$$

// Abstract allocation at θ

19. $$\left\{\begin{array}{l} \exists \mathrm{C},\mathrm{T}_c,\mathrm{T}_1,\mathrm{R},\theta.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M},\mathtt{curr}{:}\mathrm{C},\mathtt{next}{:}\mathrm{R}) * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}_1] \\ \Big((\mathrm{T}\doteq\mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] * \mathsf{complete}(\mathrm{T}_c) * \mathrm{R}\doteq\mathtt{null} * \alpha \mapsto \mathrm{N}[\theta] * \theta\mapsto\mathrm{C}[\mathrm{T}_c]) \\ \quad \vee(\exists \mathrm{O},\mathrm{T}_o,\mathrm{T}_2.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{C}[\mathrm{T}_c]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2*\mathsf{complete}(\mathrm{T}_c)*\mathrm{R}\doteq\mathrm{O} \\ \qquad *\mathsf{complete}(\mathrm{T}_o)*\mathsf{complete}(\mathrm{T}_2) * \alpha\mapsto\mathrm{N}[\theta\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2\,]*\theta\mapsto\mathrm{C}[\mathrm{T}_c])\Big) \end{array}\right\}$$

```
20.        appendChild(m, curr);
```

21. $$\left\{\begin{array}{l} \exists \mathrm{C},\mathrm{T}_c,\mathrm{T}_1,\mathrm{R},\theta.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M},\mathtt{curr}{:}\mathrm{C},\mathtt{next}{:}\mathrm{R}) * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}_1\otimes\mathrm{C}[\mathrm{T}_c]\,] \\ \Big((\mathrm{T}\doteq\mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] * \mathrm{R}\doteq\mathtt{null} * \alpha \mapsto \mathrm{N}[\theta] * \theta\mapsto\varnothing) \\ \quad \vee(\exists \mathrm{O},\mathrm{T}_o,\mathrm{T}_2.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{C}[\mathrm{T}_c]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2*\mathrm{R}\doteq\mathrm{O} \\ \qquad *\mathsf{complete}(\mathrm{T}_o)*\mathsf{complete}(\mathrm{T}_2) * \alpha\mapsto\mathrm{N}[\theta\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2\,]*\theta\mapsto\varnothing)\Big) \end{array}\right\}$$

```
22.        curr := next;
```

23. $$\left\{\begin{array}{l} \exists \mathrm{C},\mathrm{T}_c,\mathrm{T}_1,\mathrm{R},\theta.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M},\mathtt{curr}{:}\mathrm{R},\mathtt{next}{:}\mathrm{R}) * \beta \mapsto \mathrm{M}[\gamma \otimes \mathrm{T}_1\otimes\mathrm{C}[\mathrm{T}_c]\,] \\ \Big((\mathrm{T}\doteq\mathrm{T}_1 \otimes \mathrm{C}[\mathrm{T}_c] * \mathrm{R}\doteq\mathtt{null} * \alpha \mapsto \mathrm{N}[\theta] * \theta\mapsto\varnothing) \\ \quad \vee(\exists \mathrm{O},\mathrm{T}_o,\mathrm{T}_2.\ \mathrm{T}\doteq\mathrm{T}_1\otimes\mathrm{C}[\mathrm{T}_c]\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2*\mathrm{R}\doteq\mathrm{O} \\ \qquad *\mathsf{complete}(\mathrm{T}_o)*\mathsf{complete}(\mathrm{T}_2) * \alpha\mapsto\mathrm{N}[\theta\otimes\mathrm{O}[\mathrm{T}_o]\otimes\mathrm{T}_2\,]*\theta\mapsto\varnothing)\Big) \end{array}\right\}$$

// Abstract deallocation at θ and renaming

24. $$\left\{\begin{array}{l} \exists \mathrm{C}.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M},\mathtt{curr}{:}\mathrm{C},\mathtt{next}{:}-) \\ \Big((\mathrm{C}\doteq\mathtt{null} * \alpha \mapsto \mathrm{N}[\varnothing] * \beta\mapsto\mathrm{M}[\gamma\otimes\mathrm{T}]) \\ \vee(\exists \mathrm{T}_1,\mathrm{T}_c,\mathrm{T}_2.\ \mathrm{T}\doteq\mathrm{T}_1\otimes[(\mathrm{C},\mathrm{T}_c)]\otimes\mathrm{T}_2 \\ \quad *\mathsf{complete}(\mathrm{T}_c)*\mathsf{complete}(\mathrm{T}_2) * \alpha\mapsto\mathrm{N}[\mathrm{C}[\mathrm{T}_c]\otimes\mathrm{T}_2\,]*\beta\mapsto\mathrm{M}[\gamma\otimes\mathrm{T}_1])\Big) \end{array}\right\}$$

```
25.     }
```

26. $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M},\mathtt{curr}{:}\mathtt{null},\mathtt{next}{:}-) * \alpha\mapsto\mathrm{N}[\varnothing] * \beta\mapsto\mathrm{M}[\gamma\otimes\mathrm{T}]\}$

```
27.   }
```

28. $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M}) * \alpha\mapsto\mathrm{N}[\varnothing] * \beta\mapsto\mathrm{M}[\gamma\otimes\mathrm{T}]\}$

```
29. }
```

30. $\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{m}{:}\mathrm{M}) * \alpha\mapsto\mathrm{N}[\varnothing] * \beta\mapsto\mathrm{M}[\gamma\otimes\mathrm{T}]\}$

Figure 4.8.: Proof sketch of `moveChildren(n,m)` (continued from Fig. 4.7)

5. The DOM Library: $\mathbb{DOM}$

The Document Object Model (DOM) describes an XML update library and is maintained by the World Wide Web Consortium (W3C) [2]. This English standard is written in an axiomatic style that lends itself well to formalisation. The standard provides an abstract representation of the DOM tree as well as a wide range of operations for manipulating this tree. The most common use of the DOM is manipulating the content of web pages. The DOM operations are similar to those of our tree library studied in §4, but are much more extensive. DOM is an ideal example for SSL formalisation as it demonstrates the scalability of SSL reasoning. In particular, DOM follows the essence of our tree library in §4, scaling it to a real-world problem. DOM has previously been studied in the context of local reasoning. The first formal axiomatic DOM specification is given in [24, 52], using context logic (CL) [7, 6]. However, this work has several shortcomings.

First, it is not simple to integrate separation logic (SL) reasoning about e.g. C [49], Java [42] and JavaScript [21] with the DOM specifications in CL. The work in [24, 52] explores the verification of simple client programs manipulating a variable store and calling the DOM. It does not verify clients manipulating a standard program heap. More concretely, in contrast to the commutative separating conjunction ($*$) of SSL, the CL assertion language contains the non-commutative separating application ($\diamond$), that splits the DOM tree into a tree context with a hole applied to a partial DOM tree. These two operators are not compatible with each other. In particular, the integration of the CL-based DOM specification with an SL-based program logic requires altering the underlying model and extending the program logic to include a customised frame rule for the separating application. By contrast, as we demonstrated in §3.2, we can combine our SSL specification of DOM with an SL-based program logic in order to reason about DOM client programs written in e.g. C, Java and JavaScript.

We illustrate this by verifying several realistic ad-blocker client programs written in JavaScript, using the program logic of [21].

Second, the specification in [24, 52] does not always allow *local and compositional* client-side reasoning. In [24], the authors note that their DOM axioms are not always local (that is, do not always have small enough footprints), but fail to identify that, for the same reason, their client reasoning is not always compositional. We demonstrate this by presenting a simple client program which can be specified using a *single* SSL specification that locally captures its intuitive footprint, compared to *six* CL specifications that substantially over-approximate the footprint.

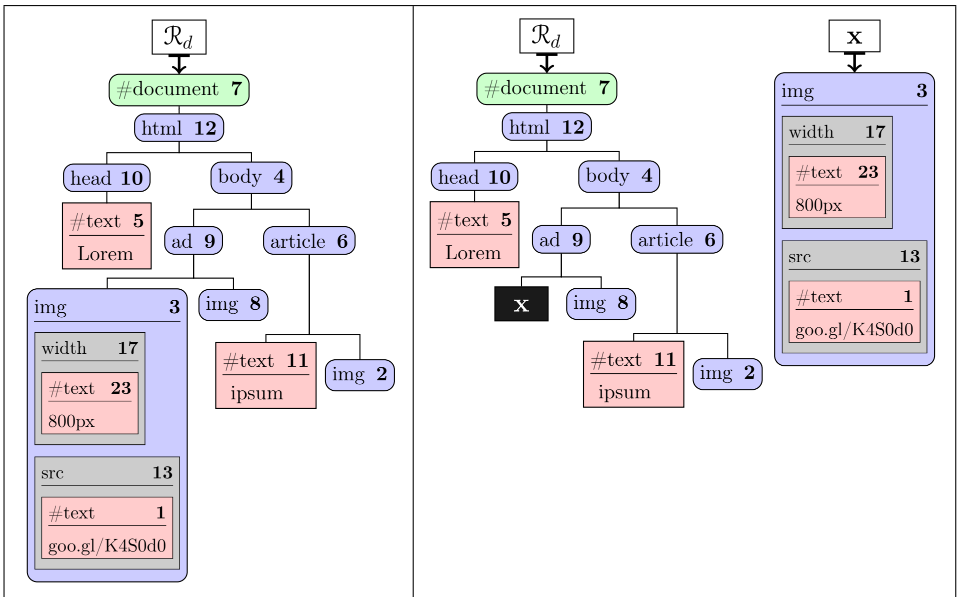
Finally, while the specification in [24] does not model live collections, the specification in [52] makes simplifying choices with respect to live collections and does not always remain faithful to the standard.

In §5.1, we present an intuitive overview of our DOM specifications in SSL and its advantages over the existing approaches. In §5.2, we instantiate the general theory of SSL presented in §3.1 in order to model the DOM library $\mathbb{DOM}$ and to write assertions describing the DOM tree structure. In §5.3, we instantiate the general methodology described in §3.2 to extend the JavaScript program logic of [21], JSLogic, to JSLogic$_{\mathbb{DOM}}$, in order to specify the behaviour of $\mathbb{DOM}$ library operations. In §5.4, we use JSLogic$_{\mathbb{DOM}}$ in order to verify several realistic ad-blocker client programs written in JavaScript.

5.1. Overview

There are currently four revisions of the DOM standard [2]. As with [24], we focus on DOM Core Level 1 which defines the general shape of the DOM tree and provides a set of operations for manipulating the tree structure [1]. Later revisions are mostly concerned with event handling and minor updates to the tree shape.

The W3C DOM Core Level 1 standard [1] is presented in an object-oriented (OO) and language-independent fashion using IDL (Interface Definition Language). It consists of a set of interfaces that describe the fields and methods exposed by each DOM datatype. A DOM object is a tree comprising a collection of *node* objects. DOM defines twelve specialised node types. We focus on an expressive fragment of DOM Core Level 1



(a) A complete abstract DOM heap

(b) The heap in (a) after abstract allocation

Figure 5.1.: Abstract DOM heaps in SSL

that allows us to create, update, and traverse DOM documents. As such, we model the four most commonly used node types: *document*, *element*, *text* and *attribute* nodes. Additionally, we model *live collections of nodes* such as the *NodeList* interface in DOM Core Levels 1-4. Our fragment underpins DOM Core Levels 1-4. By focusing on this fragment, we can direct our attention on the core difficulties of the reasoning without handling the verbosity of the entire document. As demonstrated in [52], it is straightforward to extend this fragment to the full DOM Core Level 1 without adding to the complexity of the underlying model. While it will be necessary to expand the model as we consider additional features in the higher levels of the standard (e.g. DOM events), the fragment specified here will remain largely unaffected, as these additional features are independent of our fragment. We proceed with an overview of our DOM fragment. In the remainder of this thesis we refer to our DOM fragment simply as DOM.

DOM nodes A DOM object (e.g. the tree in Fig. 5.1a) is defined as a tree comprising a collection of *node* objects. DOM nodes are the building blocks of DOM data. Each node in DOM is associated with a *type*, a *name*, an optional *value*, and information about its surroundings (e.g. its parent, siblings, etc.). As well as this generic node type, DOM defines twelve specialised node types, of which we model the four most commonly used: *text*, *element*, *attribute* and *document* nodes. Given the OO nature of the standard, each node object is uniquely identified by its reference. To capture this more abstractly (and admit non-OO implementations), we associate each node with a unique *node identifier*. We thus assume a countably infinite set of node identifiers $\textsc{Id}$, as well as a designated *document identifier* associated with the document object, $d \in \textsc{Id}$. *Document nodes* represent entire DOM objects. Each DOM object contains exactly one document node, named #document, with no value and at most a single child, referred to as the *document element*. The document node is the root of the DOM tree and provides primary access to the DOM data. In Fig. 5.1a, the document node is the node with identifier 7. *Element nodes* structure the content of a DOM object. They have arbitrary names (not containing the '#' character), no values and an arbitrary number of text and element nodes as their children. In Fig. 5.1a, the node named "html" with identifier 12 is an element node which has two children with identifiers 10 and 4. *Text nodes* represent the textual content of the document. Each text node has name "#text", an arbitrary string value and no children. In Fig. 5.1a, the node with identifier 5 is a text node with string data "Lorem". *Attribute nodes* store information about the element nodes to which they are attached. They have arbitrary names (not containing the '#' character), arbitrary string values and an arbitrary number of text nodes as their children. The attributes of an element must have unique names. In Fig. 5.1a, the element node with identifier 3 has two attributes: one with name "src", value "goo.gl/K4S0d0", and identifier 13; and another with name "width", value "800px", and identifier 17.

DOM operations We present the complete set of DOM operations and their axioms later in §5.2. Here, we describe the `n.getAttribute(s)` and `n.setAttribute(s,v)` operations and their axioms to give an intuitive account of SSL for DOM. The `n.getAttribute(s)` operation inspects the attributes of element node `n`. It returns the value of the attribute named `s` if it exists, or the empty string otherwise. For instance, given the DOM tree of Fig. 5.1a, when variable `n` holds value 3 (the element node named “img”, placed as the left child of node “ad”), and `s` holds “src”, then `r := n.getAttribute(s)` yields `r`=“goo.gl/K4S0d0”.

Intuitively, the footprint of `n.getAttribute(s)` is limited to the element node `n` and its “src” attribute. To describe this footprint minimally, we need to split the element node at `n` away from the larger surrounding DOM tree. To do this, we introduce *abstract DOM heaps* that store abstract tree fragments. For instance, Fig. 5.1a contains an abstract DOM heap with one cell at address $\mathcal{R}_d$ and a complete abstract DOM tree as its value. It is abstract in that it hides the details of how a DOM tree might be concretely represented in a machine. As before, abstract heaps allow for their data to be split by imposing additional instrumentation using abstract addresses. Such splitting is illustrated by the transition from Fig. 5.1a to Fig. 5.1b. The heap in Fig. 5.1a contains a complete tree at address $\mathcal{R}_d$. This tree can be split using *abstract allocation* to obtain the heap in Fig. 5.1b with the subtree at node 3 at a fresh, fictional *abstract cell* $\mathbf{x}$, and an incomplete tree at $\mathcal{R}_d$ with a *context hole* $\mathbf{x}$ indicating the position to which the subtree will return. Since we are only interested in the attribute named “src”, we can use abstract allocation again to split away the other unwanted attribute (“width”) and place it at a fresh abstract cell $\mathbf{y}$. After this second abstract allocation at $\mathbf{y}$, the subtree at node 3 and its “src” attribute correspond to the intuitive footprint of `n.getAttribute(s)`. Once the `getAttribute` operation is complete, we can join the tree back together through *abstract deallocation*, as in the transition from Fig. 5.1b to 5.1a.

Using SSL, we develop *local* specifications of DOM operations that only touch the intuitive footprints of the operations. The assertion language comprises *DOM assertions* that describe abstract DOM heaps. For instance, the DOM assertion below describes the abstract heap cell at $\mathbf{x}$ in Fig. 5.1b, where α denotes a logical variable corresponding to the abstract address $\mathbf{x}$:

$$\alpha \mapsto \mathrm{img}_3[\mathrm{width}_{17}[\#\mathrm{text}_{23}[800\mathrm{px}]] \odot \mathrm{src}_{13}[\#\mathrm{text}_1[\mathrm{goo.gl/K4S0d0}]], \varnothing]$$

This assertion states that the heap cell at abstract logical address α holds an “img” element with identifier 3, no children ($\varnothing$) and a set of attributes described by $\mathrm{width}_{17}[\#\mathrm{text}_{23}[800\mathrm{px}]] \odot \mathrm{src}_{13}[\#\mathrm{text}_1[\mathrm{goo.gl/K4S0d0}]]$, which contains a “src” attribute (with identifier 13 and value “goo.gl/K4S0d0”) and a “width” attribute (with identifier 17 and value “800px”). The attributes of a node are grouped by the commutative $\odot$ operator.

Once the “width” attribute of element node 3 has been abstractly allocated away at **y**, the abstract heap cell at **x** can be described by the DOM assertion $\alpha \mapsto \mathrm{img}_3[\beta \odot \mathrm{src}_{13}[\#\mathrm{text}_1[\mathrm{goo.gl/K4S0d0}]], \varnothing]$ where α and β denote logical variables corresponding to abstract addresses **x** and **y**, respectively.. When we are only interested in the value of an attribute, we can write an assertion that is agnostic to the shape of the text content under the attribute. For instance, we can write:

$$\alpha \mapsto \mathrm{img}_3[\beta \odot \mathrm{src}_{13}[\mathrm{T}], \varnothing] * \mathsf{val}(\mathrm{T}, \mathrm{goo.gl/K4S0d0})$$

to state that attribute 13 contains some text content described by logical variable T, and that the value of T (i.e. the value of the attribute) is “goo.gl/K4S0d0”. The $\mathsf{val}(\mathrm{T}, \mathrm{goo.gl/K4S0d0})$ assertion is *pure* in that it contains no resources and merely describes the string value of T.

Using SSL, we can now locally specify `r := n.getAttribute(s)` as[1]:

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{n}:\mathrm{N}, \mathtt{s}:\mathrm{S}, \mathtt{r}:-)\\ *\,\alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\beta \odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}], \gamma]\\ *\,\mathsf{val}(\mathrm{T}, \mathrm{S}'')\end{array}\right\} \mathtt{r := n.getAttribute(s)} \left\{\begin{array}{l}\mathsf{vars}(\mathtt{n}:\mathrm{N}, \mathtt{s}:\mathrm{S}, \mathtt{r}:\mathrm{S}'')\\ *\,\alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\beta \odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}], \gamma]\\ *\,\mathsf{val}(\mathrm{T}, \mathrm{S}'')\end{array}\right\} \quad (5.1)$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{n}:\mathrm{N}, \mathtt{s}:\mathrm{S}, \mathtt{r}:-)\\ *\,\alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A},\gamma] * \mathsf{out}_{\mathsf{n}}(\mathrm{A},\mathrm{S})\end{array}\right\} \mathtt{r := n.getAttribute(s)} \left\{\begin{array}{l}\mathsf{vars}(\mathtt{n}:\mathrm{N}, \mathtt{s}:\mathrm{S}, \mathtt{r}:\text{“”})\\ *\,\alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A},\gamma] * \mathsf{out}_{\mathsf{n}}(\mathrm{A},\mathrm{S})\end{array}\right\} \quad (5.2)$$

Axiom (5.1) captures the case when `n` contains an attribute named `s`; axiom (5.2) when `n` has no such attribute. The precondition of (5.1) contains three assertions. As before, the $\mathsf{vars}(\mathtt{n}:\mathrm{N}, \mathtt{s}:\mathrm{S}, \mathtt{r}:-)$ assertion describes a variable store where program variables `n`, `s` and `r` have logical values N, S and an arbitrary value $(-)$, respectively. The $\alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\beta \odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}], \gamma]$ assertion describes an abstract DOM heap cell at the abstract address α containing the subtree described by assertion $\mathrm{S}'_{\mathrm{N}}[\beta \odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}], \gamma]$. This assertion describes a subtree with a single element node with identifier N and name S'. Its children have been framed off, leaving behind the context hole γ (using abstract allocation as in the transition from Fig. 5.1a to 5.1b, then framing off the cell at γ). It has an attribute named S with

[1]It is possible to combine multiple cases into one by rewriting the pre- and postconditions as a disjunction of the cases and using logical variables to track each case. For clarity, we opt to write each case separately.

identifier $\textsc{m}$ and text content $\textsc{t}$, plus (potentially) other attributes that have been framed off, leaving behind the context hole β. This framing off of the children and attributes other than `s` captures the intuition that the footprint of `n.getAttribute(s)` is limited to element `n` and attribute `s`. Lastly, the $\mathsf{val}(\textsc{t}, \textsc{s}'')$ assertion states that the value of text content $\textsc{t}$ is $\textsc{s}''$. The postcondition of (5.1) declares that the subtree remains the same and that the value of `r` in the variable store is updated to $\textsc{s}''$, i.e. the value of the attribute named $\textsc{s}$.

The precondition of (5.2) contains the assertion $\alpha \mapsto \textsc{s}'_{\textsc{n}}[\textsc{a}, \gamma]$ where, this time, the attributes of the element node identified by $\textsc{n}$ are described by the logical variable $\textsc{a}$. With the precondition of (5.1), all other attributes can be framed off leaving context hole β. With the precondition of (5.2) however, the attributes are part of the intuitive footprint since we must check the absence of an attribute named $\textsc{s}$. This is captured by the $\mathsf{out_n}(\textsc{a}, \textsc{s})$ assertion. The postcondition of (5.2) declares that the subtree remains the same and the value of `r` in the variable store is updated to the empty string “ ”, as mandated by the English specification.

The `n.setAttribute(s,v)` operation inspects the attributes of element node `n`. It then sets the value of the attribute named `s` to `v` if such an attribute exists (5.3). Otherwise, it creates a new attribute named `s` with value `v` and attaches it to node `n` (5.4). We can specify this English description as[1]:

$$\left\{\begin{array}{l}\mathsf{vars}(\mathbf{n} : \textsc{n}, \mathbf{s} : \textsc{s}, \mathbf{v} : \textsc{s}'') \\ * \, \alpha \mapsto \textsc{s}'_{\textsc{n}}[\beta \odot \textsc{s}_{\textsc{m}}[\textsc{t}], \gamma] \\ * \, \delta \mapsto \varnothing_g * \mathsf{grove}(\textsc{t}, \textsc{g})\end{array}\right\} \texttt{n.setAttribute(s,v)} \left\{\begin{array}{l}\exists \textsc{r}.\ \mathsf{vars}(\mathbf{n} : \textsc{n}, \mathbf{s} : \textsc{s}, \mathbf{v} : \textsc{s}'') \\ * \, \alpha \mapsto \textsc{s}'_{\textsc{n}}[\beta \odot \textsc{s}_{\textsc{m}}[\#\text{text}_{\textsc{r}}[\textsc{s}'']], \gamma] \\ * \, \delta \mapsto \textsc{g}\end{array}\right\} \quad (5.3)$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathbf{n} : \textsc{n}, \mathbf{s} : \textsc{s}, \mathbf{v} : \textsc{s}'') \\ * \, \alpha \mapsto \textsc{s}'_{\textsc{n}}[\textsc{a}, \gamma] * \mathsf{out_n}(\textsc{a}, \textsc{s})\end{array}\right\} \texttt{n.setAttribute(s,v)} \left\{\begin{array}{l}\exists \textsc{m}, \textsc{r}.\ \mathsf{vars}(\mathbf{n} : \textsc{n}, \mathbf{s} : \textsc{s}, \mathbf{v} : \textsc{s}'') \\ * \, \alpha \mapsto \textsc{s}'_{\textsc{n}}[\textsc{a} \odot \textsc{s}_{\textsc{m}}[\#\text{text}_{\textsc{r}}[\textsc{s}'']], \gamma]\end{array}\right\} \quad (5.4)$$

Recall that attribute nodes may have an arbitrary number of text nodes as their children where the concatenated values of the text nodes denotes the value of the attribute. As such, when `n` contains an attribute named `s`, its value is set to `v` by removing the existing children (text nodes) of `s`, creating a new text node with value `v` and attaching it to `s` (axiom 5.3). What is then to happen to the removed children of `s`? In DOM, nodes are not disposed of: whenever a node is removed, it is no longer a part of the DOM tree but still exists in memory. To model this, we associate the document object with a *grove* designating an *unordered* space for the removed nodes. The $\delta \mapsto \varnothing_g$ assertion in the precondition of (5.3) simply reserves an empty spot ($\varnothing_g$) in the grove. The $\mathsf{grove}(\mathrm{T}, \mathrm{G})$ predicate converts the children of `s` into an unordered set of nodes G. In the postcondition the converted children of `s` (i.e. G) are moved to the empty grove at δ. Similarly, when `n` does not contain an attribute named `s`, a new attribute named `s` is created and attached to `n`. The value of `s` is set to `v` by creating a new text node with value `v` and attaching it to `s` (axiom 5.4).

Comparison to existing work (locality and compositionality) In contrast to the *commutative* separating conjunction $*$ in SSL, context logic (CL) and multi-holed context logic (MCL) use a *non-commutative separating application* $\bullet$ to split the DOM tree structure [7, 6]. For instance, the $C \bullet_\alpha P$ formula describes a tree that can be split into a context tree C with hole α and a subtree P to be applied to the context hole. The application operator $\bullet$ is not commutative in that a context cannot be applied to a tree. In [24, 52], the authors noted that the CL axioms for DOM operations such as `appendChild` were not *local*, as they required more than the intuitive footprint of the operations. Later in §5.2 we demonstrate that the SSL specifications of these operations are local and capture their intuitive footprint accurately (see p. 159). What the authors in [24, 52] did not observe was that their CL DOM specification does not provide a *compositional* way of generating local specifications for client programs. Consider the following client program that copies the value of the "src" attribute of node `p` to that of `q`:

$$\mathbb{C} \triangleq \texttt{s:=p.getAttribute("src"); q.setAttribute("src", s)}$$

Let us assume that `p` contains a "src" attribute while `q` does not. Using our SSL specifications, we can specify the behaviour of $\mathbb{C}$ locally as:

$$\left\{\alpha \mapsto P * \beta \mapsto Q * S\right\} \mathbb{C} \left\{\exists \mathrm{M}, \mathrm{O}, \mathrm{F}', \mathrm{F}''.\, \alpha \mapsto P * \beta \mapsto Q' * S'\right\} \tag{5.5}$$

with $\qquad P \triangleq \mathrm{S}^{1}_{\mathrm{P}}[\mathrm{A}_1 \odot \mathrm{src}_{\mathrm{N}}[\mathrm{T}_2]_{\mathrm{F}_2}, \mathrm{T}_1]^{\mathrm{E}_1}_{\mathrm{F}_1}$

$$\begin{aligned}
Q &\triangleq \mathrm{s}'_{\mathrm{Q}}[\mathrm{A}, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}} \\
S &\triangleq \mathsf{vars}(\mathtt{p} : \mathrm{P}, \mathtt{q} : \mathrm{Q}, \mathtt{s} : -) * \mathsf{val}(\mathrm{T}_2, \mathrm{S}) * \mathsf{out}_{\mathsf{n}}(\mathrm{A}, \text{“src”}) \\
Q' &\triangleq \mathrm{s}'_{\mathrm{Q}}[\mathrm{A} \odot \mathrm{src}_{\mathrm{M}}[\#\mathrm{text}_{\mathrm{O}}[\mathrm{S}]_{\mathrm{F}'}]_{\mathrm{F}''}, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}} \\
S' &\triangleq \mathsf{vars}(\mathtt{p} : \mathrm{P}, \mathtt{q} : \mathrm{Q}, \mathtt{s} : \mathrm{S})
\end{aligned}$$

Observe that the element nodes identified by `p` and `q` may be in one of three orientations with respect to one another: i) `p` and `q` are not related and describe disjoint subtrees; ii) `q` is an ancestor of `p`; iii) `p` is an ancestor of `q`. All three orientations are captured by the SSL specification in (5.5).

Using the specifications of [24, 52], we can specify the behaviour of $\mathbb{C}$ as follows (in multi-holed context logic (MCL) adapted to our notation), with (5.6)-(5.8) respectively describing orientations (i)-(iii) above:

$$\left\{((C \bullet_\alpha P) \bullet_\beta Q) * S\right\} \ \mathbb{C} \ \left\{\exists \mathrm{M}, \mathrm{O}, \mathrm{F}', \mathrm{F}''.\ ((C \bullet_\alpha P) \bullet_\beta Q') * S'\right\} \tag{5.6}$$

$$\left\{(Q \bullet_\alpha P) * S\right\} \ \mathbb{C} \ \left\{\exists \mathrm{M}, \mathrm{O}, \mathrm{F}', \mathrm{F}''.\ (Q' \bullet_\alpha P) * S'\right\} \tag{5.7}$$

$$\left\{(P \bullet_\alpha Q) * S\right\} \ \mathbb{C} \ \left\{\exists \mathrm{M}, \mathrm{O}, \mathrm{F}', \mathrm{F}''.\ (P \bullet_\alpha Q') * S'\right\} \tag{5.8}$$

When the subtrees at `p` and `q` are not related (i), the precondition of (5.6) states that the DOM tree can be split into a subtree with top node Q, and a tree context with hole variable β satisfying the $C \bullet_\alpha P$ formula. This context itself can be split into a subcontext with top node P and a context C with hole α. The postcondition of (5.6) states that Q is extended with a “src" attribute, and the context $C \bullet_\alpha P$ remains unchanged. This specification is not *local* in that it is larger than the intuitive footprint of $\mathbb{C}$. The only parts of the tree required by $\mathbb{C}$ are the two elements P and Q. However, the precondition in (5.6) also requires the surrounding *linking* context C: to assert that P and Q are not related (P is not an ancestor of Q and vice versa), we must appeal to a linking context C that is an ancestor of both P and Q. This results in a significant overapproximation of the footprint. As either C or P, but not both, may contain a context hole named β, specification (5.6) includes the behaviour of (5.8), which can thus be omitted. We have included it as it is more local.

More significantly however, we need to specify the orientations in (ii) and (iii) separately. This is due to the *non-commutativity* of the composition operator $\bullet$ in context logic. That is, $P \bullet_\alpha Q \neq Q \bullet_\alpha P$, in contrast to

(5.5) where $\alpha \mapsto P * \beta \mapsto Q = \beta \mapsto Q * \alpha \mapsto P$. Therefore, the number of specifications of a client program rapidly increases as its footprint grows. For instance, consider the following program with a slightly larger footprint, copying the concatenated values of "src" attributes in nodes `p` and `r` to that of `q`:

$\mathbb{C}' \triangleq$ `s:=p.getAttribute("src"); s':=r.getAttribute("src");`
`q.setAttribute("src",s+s');`

Let us assume that `p`, `r` both contain an attribute named "src" while `q` does not. We can then specify $\mathbb{C}'$ locally in SSL with *one* specification similar to (5.5), as follows:

$$\left\{\alpha \mapsto P * \beta \mapsto Q * \gamma \mapsto R * S\right\} \mathbb{C}' \left\{\exists \mathrm{M}, \mathrm{O}, \mathrm{F}', \mathrm{F}'', \mathrm{S}.\ \alpha \mapsto P * \beta \mapsto Q' * S'\right\}$$

with P, Q and Q' as defined above, and R, S and S' defined below:

$$R \triangleq \mathrm{S}^3_\mathrm{R}[\mathrm{A}_3 \odot \mathrm{src}_\mathrm{L}[\mathrm{T}_4]_{\mathrm{F}_4}, \mathrm{T}_3]^{\mathrm{E}_3}_{\mathrm{F}_3}$$
$$S \triangleq \mathsf{vars}(\mathtt{p} : \mathrm{P}, \mathtt{q} : \mathrm{Q}, \mathtt{r} : \mathrm{R}, \mathtt{s} : -) * \mathsf{val}(\mathrm{T}_2, \mathrm{S}_2) * \mathsf{val}(\mathrm{T}_4, \mathrm{S}_4) * \mathsf{out}_\mathsf{n}(\mathrm{A}, \text{“src”})$$
$$S' \triangleq \mathsf{vars}(\mathtt{p} : \mathrm{P}, \mathtt{q} : \mathrm{Q}, \mathtt{r} : \mathrm{R}, \mathtt{s} : \mathrm{S}) * \mathrm{S} \doteq \mathrm{S}_2 {+\!\!+} \mathrm{S}_4$$

By contrast, when specifying $\mathbb{C}'$ in MCL, not only is locality compromised in cases analogous to (5.6) due to the linking context, we need to provide *eight* separate triples to specify the behaviour of $\mathbb{C}'$. Forgoing locality for some cases as discussed above, we can describe $\mathbb{C}'$ by no fewer than *six* specifications!

Note that as an attempt at rectifying the non-compositionality of the CL specifications for $\mathbb{C}$ above (respectively $\mathbb{C}'$), one may instead turn to *first-order logic*. For instance, we can specify the behaviour of $\mathbb{C}$ as follows:

$$\begin{array}{c}\left\{\left(\mathsf{reach}(\mathcal{R}_d, P) \land \mathsf{reach}(\mathcal{R}_d, Q)\right) * S\right\} \\ \mathbb{C} \\ \left\{\exists \mathrm{M}, \mathrm{O}, \mathrm{F}', \mathrm{F}''.\ \left(\mathsf{reach}(\mathcal{R}_d, P) \land \mathsf{reach}(\mathcal{R}_d, Q')\right) * S'\right\}\end{array} \tag{5.9}$$

where $\mathsf{reach}(\mathcal{R}_d, P)$ simply states that the element node described by P is *reachable* from the root of the DOM tree $\mathcal{R}_d$. However, the specification in (5.9) is inaccurate. More concretely, the pre- and postcondition state that the element nodes $\textsc{p}$ and $\textsc{q}$ are reachable from the DOM tree root $\mathcal{R}_d$. What this specification does *not* state is that with the exception of the element node $\textsc{q}$, the remainder of the DOM tree remains unchanged. That is, the *context* surrounding the element nodes $\textsc{p}$ and $\textsc{q}$ is not altered by program $\mathbb{C}$. This is because the assertions in the pre- and postcondition are too weak in that they merely stipulate that the element nodes $\textsc{p}$ and $\textsc{q}$ be reachable from the root $\mathcal{R}_d$. As such, it is possible for $\mathbb{C}$ to alter the DOM tree freely, so long as the reachability of $\textsc{p}$ and $\textsc{q}$ is preserved.

One may attempt to strengthen the specification in (5.9) by adding an additional predicate, $\mathsf{DOM}(\mathcal{R}_d, \textsc{t}_{\textsc{dom}})$, tracking the DOM tree as follows:

$$\left\{(\mathsf{DOM}(\mathcal{R}_d, \textsc{t}_{\textsc{dom}}) \wedge \mathsf{reach}(\mathcal{R}_d, P) \wedge \mathsf{reach}(\mathcal{R}_d, Q)) * S\right\}$$
$$\mathbb{C}$$
$$\left\{\exists \textsc{m}, \textsc{o}, \textsc{f}', \textsc{f}''.\ (\mathsf{DOM}(\mathcal{R}_d, \textsc{t}_{\textsc{dom}}) \wedge \mathsf{reach}(\mathcal{R}_d, P) \wedge \mathsf{reach}(\mathcal{R}_d, Q')) * S'\right\}$$

where $\mathsf{DOM}(\mathcal{R}_d, \textsc{t}_{\textsc{dom}})$ describes the *entire* DOM tree at $\mathcal{R}_d$. However, not only is this specification *not local* in that it encompasses the entire DOM tree at $\mathcal{R}_d$ (via the $\mathsf{DOM}(\mathcal{R}_d, \textsc{t}_{\textsc{dom}})$ predicate), it is in fact *incorrect*. This is because the above specification states that the DOM tree remains completely unchanged by $\mathbb{C}$: the contents of the tree are captured by the same logical variable $\textsc{t}_{\textsc{dom}}$ in both the pre- and the postcondition. To remedy this, one must *split* the DOM tree into elements $\textsc{p}$ and $\textsc{q}$ and the context C surrounding them, so that one can assert that C is not altered by $\mathbb{C}$. This splitting of data into contexts and their sub-data is indeed the key idea behind context logic which was later adopted by SSL.

This concludes the overview of our DOM specification in SSL and its advantages over the existing approaches. In what follows, we instantiate the general theory of SSL presented in §3.1 in order to model the DOM library formally and to write assertions describing the DOM tree structure.

5.2. SSL Model and Assertions: Library $\mathbb{DOM}$

Recall that the general theory of SSL is parametric and may be instantiated for a particular library of structured data. In §3 we presented the general theory of SSL with its parameters delineated in solid boxes labelled "SSL Parameter". In what follows, we revisit the SSL parameters and instantiate them for the DOM library $\mathbb{DOM}$. As before, we present these instances in dashed boxes labelled "SSL $\mathbb{DOM}$ Instance (Parameter X)", where X is the reference to the corresponding SSL parameter in §3.

DOM root addresses Recall that program heaps (e.g. the DOM program heap in Fig. 5.1a) are mappings from root addresses to complete program data with no context holes. For the DOM library, we define a designated *root address*, $\mathcal{R}_d$, denoting the location in the DOM heap where the DOM object is stored.

SSL $\mathbb{DOM}$ Instance (Parameter 1)

Definition 55 (DOM root addresses)**.** The set of *DOM root addresses* is $\textsc{RAdd}_{\mathbb{DOM}} \triangleq \{\mathcal{R}_d\}$.

DOM program data Recall that *program data* is library specific and provides a high-level representation of the underlying data structure without exposing how the data structure may be represented in the machine. The program data for DOM describes the DOM object as a complete DOM document node (e.g. the document node with identifier 7 in Fig. 5.1a). A DOM document node in turn may contain text, element and attribute nodes. We associate each node with a set of *forest listeners*, *fs*; we further associate element and document nodes with a set of *tag listeners*, *ts*. As we describe shortly, we use these listeners to model *live collections*. Ignoring *fs* and *ts* for now, we write: i) $\#\text{text}_n[\text{s}]_{fs}$ for the text node with identifier n and text data s; ii) $\text{s}_n[\text{as}, \text{f}]^{ts}_{fs}$ for the element node with identifier n, tag name s, *attribute set* as, and *child forest* f; iii) $\text{s}_n[\text{tf}]_{ts}$ for the attribute node with identifier n, name s, and *text forest* tf; and iv) $\#\text{doc}_d[\text{e}]^{ts}_{fs}$ & g for the document object with the designated identifier d, document element e (or $\varnothing_e$ for no document element) and *grove* g.

DOM nodes may be grouped into *forests*, *attribute sets*, *text forests* and *groves*, ranged over by f, as, tf and g, respectively. A *forest* represents the child list of an element node, modelled as an *ordered*, possibly empty collection of element and text nodes. An *attribute set* represents the attribute nodes associated with an element node and is modelled as an *unordered*, possibly empty collection of attribute nodes. A *text forest* describes the child list of an attribute node, modelled as an ordered, possibly empty collection of text nodes. A *grove* is where the orphaned nodes are stored. In DOM, nodes are never discarded and whenever a node is removed from the document, it is moved to the grove. The grove is also where newly created nodes are placed. The document object is thus associated with a grove, modelled as an unordered, possibly empty collection of forests, text forest and attribute sets.

Live collections and tag/forest listeners The DOM API provides several interfaces for traversing DOM trees based on *live collections* of nodes, such as the *NodeList* interface in DOM Core Levels 1-4. DOM Core Level 4 also introduces the *HTMLCollection* interface for live collections of element nodes. We describe our model of live collections in terms of NodeLists. However, our model is abstract and captures the behaviour of both NodeLists and HTMLCollections.

The NodeList interface describes an ordered collection of nodes. NodeLists are *live* in that they dynamically reflect the changes to the document. Several DOM operations return NodeLists. For instance, the `n.getElementsByTagName(s)` operation returns a NodeList (using depth-first, left-to-right search) containing the identifiers of those element nodes under the tree rooted at `n` whose tag names match `s`. Given the DOM tree of Fig. 5.1a, when `n`=4 and `s`="img", then `r:=n.getElementsByTagName(s)` yields `r`=$[3, 8, 2]$. However, since NodeLists are live, if node 8 is later removed from the document, then `r`=$[3, 2]$. This operation may be called on both document and element nodes. We thus associate each such node with a *set of tag listeners*, *ts*. Each tag listener is of the form $(\mathrm{s}, \mathit{fid})$ where s denotes the search string (e.g. "img" in the example above) and $\mathit{fid} \in \textsc{Id}$ denotes the identifier of the resulting NodeList.

The `n.childNodes` operation also returns a NodeList, containing the identifiers of the immediate children of `n`. For instance, with the DOM tree of Fig. 5.1a, when `n`=4, then `r := n.childNodes` returns `r`=$[9, 6]$. As before, the value of `r` is live and dynamically reflects the changes to the child forest of `n`. The `n.childNodes` operation may be called on *any* DOM node (the call on text nodes always yields an empty list as they have no children). We thus associate each DOM node with a *set of forest listeners*, *fs*. Each forest listener, $\mathit{fid} \in \textsc{Id}$, denotes the identifier of a NodeList.

SSL $\mathbb{DOM}$ Instance (Parameter 2)

Definition 56 (DOM program data)**.** Let $\textsc{Id}$ denote a countably infinite set of *DOM identifiers* and $\textsc{Char}$ denote a set of *characters*. The sets of *DOM strings* $\mathrm{s} \in \mathrm{S}$, *text nodes* $\mathrm{t} \in \mathrm{T}$, *element nodes* $\mathrm{e} \in \mathrm{E}$, *attribute nodes* $\mathrm{a} \in \mathrm{A}$, *document nodes* $\mathrm{doc} \in \mathrm{D}$, *forests* $\mathrm{f} \in \mathrm{F}$, *attribute sets* $\mathrm{as} \in \mathrm{AS}$, *text forests* $\mathrm{tf} \in \mathrm{TF}$ and *groves* $\mathrm{g} \in \mathrm{G}$, are defined by the following grammars where $n \in \textsc{Id}$, $c \in \textsc{Char}$, $fs \in \mathcal{P}(\textsc{Id})$ and $ts \in \mathcal{P}(\mathrm{S}{\times}\textsc{Id})$:

$$\mathrm{s} ::= \varnothing_s \mid \mathrm{c} \mid \mathrm{s}_1.\mathrm{s}_2 \qquad \mathrm{t} ::= \#\mathrm{text}_n[\mathrm{s}]_{fs} \qquad \mathrm{e} ::= \mathrm{s}_n[\mathrm{as}, \mathrm{f}]^{ts}_{fs} \qquad \mathrm{a} ::= \mathrm{s}_n[\mathrm{tf}]_{fs}$$
$$\mathrm{doc} ::= \#\mathrm{doc}_d[\varnothing_e]^{ts}_{fs} \;\&\; \mathrm{g} \;\mid\; \#\mathrm{doc}_d[\mathrm{e}]^{ts}_{fs} \;\&\; \mathrm{g}$$
$$\mathrm{f} ::= \varnothing_f \mid \mathrm{t} \mid \mathrm{e} \mid \mathrm{f}_1 \otimes \mathrm{f}_2 \qquad \mathrm{as} ::= \varnothing_a \mid \mathrm{a} \mid \mathrm{as}_1 \odot \mathrm{as}_2$$
$$\mathrm{tf} ::= \varnothing_{tf} \mid \mathrm{t} \mid \mathrm{tf}_1 \oslash \mathrm{tf}_2 \qquad \mathrm{g} ::= \varnothing_g \mid \mathrm{e} \mid \mathrm{t} \mid \mathrm{a} \mid \mathrm{g}_1 \oplus \mathrm{g}_2$$

The $.$, $\otimes$, $\odot$, $\oslash$ and $\oplus$ operations are associative with identities $\varnothing_s$, $\varnothing_f$, $\varnothing_a$, $\varnothing_{tf}$ and $\varnothing_g$, respectively; the $\odot$ and $\oplus$ operations are commutative. All data are equal up to the properties of $.$, $\oslash$, $\otimes$, $\odot$ and $\oplus$. DOM Data do not contain repeated identifiers; element nodes contain attributes with distinct names.[2]
The set of *DOM program data* is $\mathrm{d} \in \textsc{PData}_{\mathbb{DOM}} \triangleq \mathrm{D}$.

When the type of data is clear from the context, we drop the subscripts and write e.g. $\varnothing$ for $\varnothing_f$. For clarity, we drop the forest and tag listeners when not relevant to the discussion and write e.g. $\mathrm{s}_n[\mathrm{as}, \mathrm{f}]$ instead of $\mathrm{s}_n[\mathrm{as}, \mathrm{f}]^{ts}_{fs}$.

DOM program values Recall that SSL assumes an abstract library to define a set of *library-specific program values* that include root addresses. Library-specific program values denote the set of values that may be observed by the clients of the library (via program variables). For the DOM library $\mathbb{DOM}$, the program values include the DOM root address $\mathcal{R}_d$, DOM identifiers in $\textsc{Id}$ and DOM strings in S (Def. 56).

[2]It is straightforward to formalise these restrictions.

SSL $\mathbb{DOM}$ Instance (Parameter 3)

Definition 57 (DOM program values)**.** Given the set of DOM root addresses $\textsc{RAdd}_{\mathbb{DOM}}$ (Def. 55) and the sets of DOM identifiers $\textsc{Id}$ and DOM strings S (Def. 56), the set of *program values for DOM* is $v \in \textsc{PVal}_{\mathbb{DOM}} \triangleq \textsc{RAdd}_{\mathbb{DOM}} \cup \textsc{Id} \cup \mathrm{S}$.

DOM logical data Recall that logical heaps (with context holes) such as the logical DOM heap in Fig. 5.1b are mappings from addresses to *logical data.* As with program data, logical data is library specific. However, unlike DOM program data describing *complete* DOM objects (i.e. complete document nodes only), logical DOM data describes incomplete data with context holes, as well as DOM data fragments. More concretely, DOM logical data comprises text, element, attribute and document nodes, as well as DOM forests, attribute sets, text forests and groves. Moreover, each of these DOM data may be incomplete with context holes.

SSL $\mathbb{DOM}$ Instance (Parameter 4)

Definition 58 (DOM logical data)**.** The sets of *logical text nodes* $\mathbf{t} \in \mathbf{T}$, *logical element nodes* $\mathbf{e} \in \mathbf{E}$, *logical attribute nodes* $\mathbf{a} \in \mathbf{A}$, *logical document nodes* $\mathbf{doc} \in \mathbf{D}$, *logical forests* $\mathbf{f} \in \mathbf{F}$, *logical attribute sets* $\mathbf{as} \in \mathbf{AS}$, *logical text forests* $\mathbf{tf} \in \mathbf{TF}$ and *logical groves* $\mathbf{g} \in \mathbf{G}$, are defined by the following grammars where $n \in \textsc{Id}$, $\mathrm{s} \in \mathrm{S}$ (Def. 56), $\mathbf{x} \in \textsc{AAdd}$ (Def. 8), $fs \in \mathcal{P}(\textsc{Id})$ and $ts \in \mathcal{P}(\mathrm{S}{\times}\textsc{Id})$:

$$\mathbf{t} ::= \#\mathrm{text}_n[\mathrm{s}]_{fs} \qquad \mathbf{e} ::= \mathrm{s}_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs} \qquad \mathbf{a} ::= \mathrm{s}_n[\mathbf{tf}]_{fs}$$

$$\mathbf{doc} ::= \#\mathrm{doc}_d[\varnothing_e]^{ts}_{fs} \mathbin{\&} \mathbf{g} \;\mid\; \#\mathrm{doc}_d[\mathbf{e}]^{ts}_{fs} \mathbin{\&} \mathbf{g} \;\mid\; \#\mathrm{doc}_d[\mathbf{x}]^{ts}_{fs} \mathbin{\&} \mathbf{g}$$

$$\mathbf{f} ::= \varnothing_f \mid \mathbf{x} \mid \mathbf{t} \mid \mathbf{e} \mid \mathbf{f}_1 \otimes \mathbf{f}_2 \qquad \mathbf{as} ::= \varnothing_a \mid \mathbf{x} \mid \mathbf{a} \mid \mathbf{as}_1 \odot \mathbf{as}_2$$

$$\mathbf{tf} ::= \varnothing_{tf} \mid \mathbf{x} \mid \mathbf{t} \mid \mathbf{tf}_1 \oslash \mathbf{tf}_2 \qquad \mathbf{g} ::= \varnothing_g \mid \mathbf{x} \mid \mathbf{t} \mid \mathbf{e} \mid \mathbf{a} \mid \mathbf{g}_1 \oplus \mathbf{g}_2$$

The $.$, $\otimes$, $\odot$, $\oslash$ and $\oplus$ operations are associative with identities $\varnothing_s$, $\varnothing_f$, $\varnothing_a$, $\varnothing_{tf}$ and $\varnothing_g$, respectively; the $\odot$ and $\oplus$ operations are commutative. All logical data are equal up to the properties of $.$, $\oslash$, $\otimes$, $\odot$ and $\oplus$. DOM logical data does not contain repeated identifiers and abstract addresses; element nodes contain attributes with distinct names.[3]

The set of *DOM logical data*, $\mathbf{d} \in \textsc{LData}_{\mathbb{DOM}}$, is defined as follows:

$$\textsc{LData}_{\mathbb{DOM}} \triangleq \mathbf{T} \cup \mathbf{E} \cup \mathbf{A} \cup \mathbf{D} \cup \mathbf{F} \cup \mathbf{AS} \cup \mathbf{TF} \cup \mathbf{G}$$

The *DOM address function*, $\text{addr}(.) : \textsc{LData}_{\mathbb{DOM}} \to \mathcal{P}(\textsc{AAdd})$, is defined inductively over the structure of DOM logical data as follows, where $\varnothing_\dagger \in \{\varnothing_f, \varnothing_a, \varnothing_{tf}, \varnothing_g\}$ and $\ddagger \in \{\otimes, \odot, \oslash, \oplus\}$:

$$\begin{aligned}
\text{addr}(\#\text{text}_n[\text{s}]_{fs}) &\triangleq \emptyset \\
\text{addr}(\text{s}_n[\mathbf{as}, \mathbf{f}]_{fs}^{ts}) &\triangleq \text{addr}(\mathbf{as}) \uplus \text{addr}(\mathbf{f}) \\
\text{addr}(\text{s}_n[\mathbf{tf}]_{fs}) &\triangleq \text{addr}(\mathbf{tf}) \\
\text{addr}(\#\text{doc}_d[\varnothing_e]_{fs}^{ts} \ \&\ \mathbf{g}) &\triangleq \text{addr}(\mathbf{g}) \\
\text{addr}(\#\text{doc}_d[\mathbf{e}]_{fs}^{ts} \ \&\ \mathbf{g}) &\triangleq \text{addr}(\mathbf{e}) \uplus \text{addr}(\mathbf{g}) \\
\text{addr}(\#\text{doc}_d[\mathbf{x}]_{fs}^{ts} \ \&\ \mathbf{g}) &\triangleq \{\mathbf{x}\} \uplus \text{addr}(\mathbf{g}) \\
\text{addr}(\mathbf{x}) &\triangleq \{\mathbf{x}\} \\
\text{addr}(\varnothing_\dagger) &\triangleq \emptyset \\
\text{addr}(\mathbf{d}_1 \ddagger \mathbf{d}_2) &\triangleq \text{addr}(\mathbf{d}_1) \uplus \text{addr}(\mathbf{d}_2)
\end{aligned}$$

Definition 59 (DOM logical heaps)**.** Given DOM logical data $\textsc{LData}_{\mathbb{DOM}}$ (Def. 58), the *separation algebra of DOM logical heaps* is $(\textsc{LHeap}_{\mathbb{DOM}}, \bullet_{\mathbb{DOM}}, \mathbf{0}_{\mathbb{DOM}})$, defined as the instantiation of the parametric separation algebra $(\textsc{LHeap}_{\langle . \rangle}, \bullet_{\langle . \rangle}, \mathbf{0}_{\langle . \rangle})$ in Def. 14 with $\textsc{LData}_{\mathbb{DOM}}$.

As before, when the type of data is clear from the context, we drop the subscripts and write $\bullet$ for $\bullet_{\mathbb{DOM}}$ and write $\mathbf{0}$ for $\mathbf{0}_{\mathbb{DOM}}$.

DOM context application Recall that abstract data in SSL may be combined via abstract deallocation (e.g. the transition from Fig. 5.1b to 5.1a) where the subdata at an abstract address is *collapsed* into its counterpart context hole. Data collapsing is defined in terms of *context application.* As with the tree data in §4.1, we define context application $\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2$ for DOM data in the standard way: it is undefined when $\mathbf{x} \notin \text{addr}(\mathbf{d}_1)$;

[3]It is straightforward to formalise these restrictions.

otherwise, it is defined as $\mathbf{d}_1[\mathbf{d}_2/\mathbf{x}]$, denoting the standard substitution of $\mathbf{d}_2$ for $\mathbf{x}$ in $\mathbf{d}_1$, provided that the result is in $\textsc{LData}_{\mathbb{DOM}}$.

SSL $\mathbb{DOM}$ Instance (Parameter 5)

Definition 60 (DOM application)**.** The *DOM context application* function, $\diamond : \textsc{LData}_{\mathbb{DOM}} \times \textsc{AAdd} \times \textsc{LData}_{\mathbb{DOM}} \rightharpoonup \textsc{LData}_{\mathbb{DOM}}$, is defined inductively over the structure of DOM logical data as follows, where $\varnothing_\dagger \in \{\varnothing_f, \varnothing_a, \varnothing_{tf}, \varnothing_g, \varnothing_e\}$ and $\ddagger \in \{\otimes, \odot, \oslash, \oplus\}$:

$$\#\text{text}_n[\text{s}]_{fs} \diamond_{\mathbf{x}} \mathbf{d} \qquad \text{undefined}$$

$$\text{s}_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs} \diamond_{\mathbf{x}} \mathbf{d} \triangleq \begin{cases} \text{s}_n[\mathbf{as}', \mathbf{f}]^{ts}_{fs} & \text{if } \ \mathbf{as} \diamond_{\mathbf{x}} \mathbf{d} {=} \mathbf{as}' \\ & \text{and } \ \text{s}_n[\mathbf{as}', \mathbf{f}]^{ts}_{fs} \in \textsc{LData}_{\mathbb{DOM}} \\ \text{s}_n[\mathbf{as}, \mathbf{f}']^{ts}_{fs} & \text{if } \ \mathbf{f} \diamond_{\mathbf{x}} \mathbf{d} {=} \mathbf{f}' \\ & \text{and } \ \text{s}_n[\mathbf{as}, \mathbf{f}']^{ts}_{fs} \in \textsc{LData}_{\mathbb{DOM}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{s}_n[\mathbf{tf}]_{fs} \diamond_{\mathbf{x}} \mathbf{d} \triangleq \begin{cases} \text{s}_n[\mathbf{tf}']_{fs} & \text{if } \ \mathbf{tf} \diamond_{\mathbf{x}} \mathbf{d} {=} \mathbf{tf}' \text{ and } \text{s}_n[\mathbf{tf}']_{fs} \in \textsc{LData}_{\mathbb{DOM}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\#\text{doc}_d[\varnothing_e]^{ts}_{fs} \,\&\, \mathbf{g}) \diamond_{\mathbf{x}} \mathbf{d} \triangleq \begin{cases} \#\text{doc}_d[\varnothing_e]^{ts}_{fs} \,\&\, \mathbf{g}' & \text{if } \ \mathbf{g} \diamond_{\mathbf{x}} \mathbf{d} = \mathbf{g}' \ \text{ and} \\ & \#\text{doc}_d[\varnothing_e]^{ts}_{fs} \,\&\, \mathbf{g}' {\in} \textsc{LData}_{\mathbb{DOM}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\#\text{doc}_d[\mathbf{e}]^{ts}_{fs} \,\&\, \mathbf{g}) \diamond_{\mathbf{x}} \mathbf{d} \triangleq \begin{cases} \#\text{doc}_d[\mathbf{e}']^{ts}_{fs} \,\&\, \mathbf{g} & \text{if } \ \mathbf{e} \diamond_{\mathbf{x}} \mathbf{d} = \mathbf{e}' \ \text{ and} \\ & \#\text{doc}_d[\mathbf{e}']^{ts}_{fs} \,\&\, \mathbf{g} \in \textsc{LData}_{\mathbb{DOM}} \\ \#\text{doc}_d[\mathbf{e}]^{ts}_{fs} \,\&\, \mathbf{g}' & \text{if } \ \mathbf{g} \diamond_{\mathbf{x}} \mathbf{d} = \mathbf{g}' \ \text{ and} \\ & \#\text{doc}_d[\mathbf{e}]^{ts}_{fs} \,\&\, \mathbf{g}' \in \textsc{LData}_{\mathbb{DOM}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\#\mathrm{doc}_d[\mathbf{y}]^{ts}_{fs} \mathbin{\&} \mathbf{g}) \diamond_{\mathbf{x}} \mathbf{d} \triangleq \begin{cases} \#\mathrm{doc}_d[\mathbf{d}']^{ts}_{fs} \mathbin{\&} \mathbf{g} & \text{if } \ \mathbf{y} \diamond_{\mathbf{x}} \mathbf{d} = \mathbf{d}' \ \text{ and} \\ & \#\mathrm{doc}_d[\mathbf{d}']^{ts}_{fs} \mathbin{\&} \mathbf{g} \in \textsc{LData}_{\mathbb{DOM}} \\ \#\mathrm{doc}_d[\mathbf{e}]^{ts}_{fs} \mathbin{\&} \mathbf{g}' & \text{if } \ \mathbf{g} \diamond_{\mathbf{x}} \mathbf{d} = \mathbf{g}' \ \text{ and} \\ & \#\mathrm{doc}_d[\mathbf{e}]^{ts}_{fs} \mathbin{\&} \mathbf{g}' \in \textsc{LData}_{\mathbb{DOM}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\varnothing_{\dagger} \diamond_{\mathbf{x}} \mathbf{d} \quad \text{undefined} \qquad\qquad \mathbf{y} \diamond_{\mathbf{x}} \mathbf{d} \triangleq \begin{cases} \mathbf{d} & \text{if } \ \mathbf{x} = \mathbf{y} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\mathbf{d}_1 \ddagger \mathbf{d}_2) \diamond_{\mathbf{x}} \mathbf{d} \triangleq \begin{cases} \mathbf{d}' \ddagger \mathbf{d}_2 & \text{if } \ \mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d} = \mathbf{d}' \\ & \text{and } \ \mathbf{d}' \ddagger \mathbf{d}_2 \in \textsc{LData}_{\mathbb{DOM}} \\ \mathbf{d}_1 \ddagger \mathbf{d}' & \text{if } \ \mathbf{d}_2 \diamond_{\mathbf{x}} \mathbf{d} = \mathbf{d}' \\ & \text{and } \ \mathbf{d}' \ddagger \mathbf{d}_2 \in \textsc{LData}_{\mathbb{DOM}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

DOM operations For the DOM library $\mathbb{DOM}$, the operations comprise the DOM Core Level 1 operations associated with i) the generic Node interface; ii) the text, element, attribute and document node interfaces; and iii) the NodeList interface. Similar to the tree library studied in §4, the DOM operations comprise "lookup" operations for inspecting the DOM tree, as well as "update" operations manipulating the structure of the DOM tree.

SSL $\mathbb{DOM}$ Instance (Parameter 6)

Definition 61 (DOM operations)**.** The set of *DOM operations*, $\mathtt{C}_{\mathbb{DOM}} \in \textsc{Op}_{\mathbb{DOM}}$, is defined as follows:

$$\textsc{Op}_{\mathbb{DOM}} = \textsc{NOp} \cup \textsc{TOp} \cup \textsc{EOp} \cup \textsc{AOp} \cup \textsc{DOp} \cup \textsc{NLOp}$$

with the *node operations* NOp, *text node operations* TOp, *element* *node operations* $\text{EO}\textsc{p}$, *attribute node operations* $\text{AO}\textsc{p}$, *document node operations* $\text{DO}\textsc{p}$ and *NodeList operations* $\text{NLO}\textsc{p}$ defined by the following grammars, where $\mathtt{a}, \mathtt{c}, \mathtt{i}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{r}, \mathtt{s}, \mathtt{u} \in \text{PV}\textsc{ar}$ (Def. 1):

$$
\begin{aligned}
\text{NO}\textsc{p} \ni \mathtt{C} ::= {} & \mathtt{r:=n.nodeName} \mid \mathtt{r:=n.nodeValue} \mid \mathtt{r:=n.nodeType} \\
& \mid \mathtt{r:=n.parentNode} \mid \mathtt{r:=n.childNodes} \\
& \mid \mathtt{r:=n.firstChild} \mid \mathtt{r:=n.lastChild} \\
& \mid \mathtt{r:=n.previousSibling} \mid \mathtt{r:=n.nextSibling} \\
& \mid \mathtt{r:=n.ownerDocument} \mid \mathtt{r:=u.insertBefore(m,n)} \\
& \mid \mathtt{r:=u.replaceChild(n,o)} \mid \mathtt{r:=u.removeChild(o)} \\
& \mid \mathtt{r:=u.appendChild(n)} \mid \mathtt{r:=n.hasChildNodes()} \\
\text{TO}\textsc{p} \ni \mathtt{C} ::= {} & \mathtt{r:=n.data} \mid \mathtt{r:=n.length} \\
& \mid \mathtt{r:=n.substringData(o,c)} \mid \mathtt{n.appendData(a)} \\
& \mid \mathtt{n.insertData(o,s)} \mid \mathtt{n.deleteData(o,c)} \\
& \mid \mathtt{n.replaceData(o,c,s)} \mid \mathtt{r:=n.splitText(o)} \\
\text{EO}\textsc{p} \ni \mathtt{C} ::= {} & \mathtt{r:=n.tagName} \mid \mathtt{r:=n.getAttribute(s)} \\
& \mid \mathtt{n.setAttribute(s,v)} \mid \mathtt{n.removeAttribute(s)} \\
& \mid \mathtt{r:=n.getAttributeNode(s)} \\
& \mid \mathtt{r:=n.setAttributeNode(a)} \\
& \mid \mathtt{r:=n.removeAttributeNode(a)} \\
& \mid \mathtt{r:=n.getElementsByTagName(s)} \\
\text{AO}\textsc{p} \ni \mathtt{C} ::= {} & \mathtt{r:=n.name} \mid \mathtt{r:=n.value} \\
\text{DO}\textsc{p} \ni \mathtt{C} ::= {} & \mathtt{r:=n.documentElement} \mid \mathtt{r:=n.createElement(s)} \\
& \mid \mathtt{r:=n.createTextNode(s)} \mid \mathtt{r:=n.createAttribute(s)} \\
& \mid \mathtt{r:=n.getElementsByTagName(s)} \\
\text{NLO}\textsc{p} \ni \mathtt{C} ::= {} & \mathtt{r:=f.length()} \mid \mathtt{r:=f.item(i)}
\end{aligned}
$$

A full description of the behaviour of DOM operations is given in §A. We proceed with a description of a select number of DOM operations. DOM operations can be categorised into three groups as follows:

1. Operations for *reading* the data associated with DOM *nodes* such as:

- `r:=n.parentNode`: when `n` identifies a DOM node, the identifier of the parent node of `n` is returned in `r` when it exists; `null` is returned if `n` is a document or attribute node (a document node is the top-most node and has no parent; an attribute node is associated with an element node, but is *not* the child of an element node), or if `n` resides in the grove.
- `r:=n.length`: when `n` identifies a text node, the length of the value (text contents) of `n` is returned in `r` (i.e. returns a non-negative value corresponding to the number of characters in the value of `n`).
- `r:=n.getAttribute(s)`: when `n` identifies an element node and `s` holds a string, the attributes of `n` are inspected and the value of the attribute named `s` is returned in `r` if such an attribute exists. If `n` has no attribute named `s` then the empty string is returned.
- `r :=n.name`: when `n` identifies an attribute node, the name of `n` is returned in `r`.
- `r:=n.documentElement`: when `n` identifies a document node, the identifier of the document element is returned in `r` when it exists; otherwise `null` is returned.

2. Operations for *modifying* the data associated with DOM *nodes* such as:

   - `u.appendChild(n)`: when `u` and `n` identify DOM nodes, this operation appends `n` to the end of `u`'s child list and returns `n`. It fails if i) the result of appending does not correspond to a well-typed DOM node (e.g. when `n` is a document node); or ii) `n` is an ancestor of `u` (otherwise it would introduce a cycle and break the DOM structure).
   - `n.insertData(o,s)`: when `n` identifies a text node, `o` holds an integer value and `s` holds a string, then `s` is inserted into the text contents (value) of `n` at offset `o` (indexed from 0). This operation fails if `o` is an invalid offset (i.e. negative or greater than the length of the value of `n`). For instance, when the value

of `n` is “lorem”, `o`=1 and `s`=“ipsum”, then `n.insertData(o,s)` updates the value of `n` to “lipsumorem”.

- `n.setAttribute(s,v)`: when `n` identifies an element node and `s` and `v` hold strings, the attributes of `n` are inspected and the value of the attribute named `s` is set to `v` if such an attribute exists. If `n` has no attribute named `s` then the attribute set of `n` is extended with a new attribute with name `s` and value `v`.
- `r:=n.createElement(s)` :when `n` identifies a document node and `s` holds a *safe* DOM string, the DOM grove is extended with a new element node named `s`. The identifier of the new element node is returned in `r`. A DOM string is safe if it does not contain the invalid ‘#’ character. The new element has no attributes and no children. This operation fails if `s` holds an unsafe string (one containing the ‘#’ character).

3. Operations for creating or reading from *NodeLists* such as:
   - `r:=n.childNodes`: when `n` identifies a DOM node, this operation compiles a NodeList containing the identifiers of the children of node `n` and returns its identifier in `r`.
   - `r:=n.getElementsByTagName(s)`: when `n` identifies an element or document node and `s` holds a string, this operation searches the child list of `n` (using depth-first, left-to-right search), compiles a NodeList containing the identifiers of those element nodes whose names match `s` and returns the identifier of this NodeList in `r`.
   - `r:=f.length()`: when `f` identifies a NodeList, the length of `f` is returned in `r`.
   - `r:=f.item(i)`: when `f` identifies a NodeList and `i` holds an integer, the $\mathtt{i}^{\text{th}}$ item of `f` (indexed from 0) is returned in `r`. If `i` holds an out-of-bounds value (i.e. negative or greater than or equal to the length of `f`) then `null` is returned.

DOM logical values Recall that SSL assumes a library to define a set of library-specific logical values denoting the values associated with logical variables. For the DOM library, the set of logical values is defined as the extension of DOM program values (Def. 57) with abstract addresses (Def. 8), DOM logical data (Def. 58) and lists of DOM identifiers (Def. 56). We include the DOM logical data in the set of logical values to allow for writing expressions that inspect the DOM tree structure. As we demonstrate later, we appeal to lists of identifiers to specify the behaviour of NodeList operations.

SSL $\mathbb{DOM}$ Instance (Parameter 7)

Definition 62 (DOM logical values)**.** Given the DOM identifiers $\textsc{Id}$ (Def. 56), DOM program values $\textsc{PVal}_{\mathbb{DOM}}$ (Def. 57), abstract addresses $\textsc{AAdd}$ (Def. 8) and the DOM logical data $\textsc{LData}_{\mathbb{DOM}}$ (Def. 58), the set of *logical values for DOM* is $v \in \textsc{LVal}_{\mathbb{DOM}} \triangleq \textsc{PVal}_{\mathbb{DOM}} \cup \textsc{AAdd} \cup \textsc{LData}_{\mathbb{DOM}} \cup \textsc{List}\langle\textsc{Id}\rangle$.

DOM logical expressions Recall that libraries may specify a set of logical expressions in order to assert certain properties about the underlying data. As with the tree library $\mathbb{T}$ studied in §4, the logical expressions for the DOM library include logical variables and are defined by a similar grammar to that of DOM logical data (Def. 58). We further extend the set of DOM logical expressions with $|\textsc{l}|$ and $|\textsc{l}|^{\textsc{i}}$ expressions describing the length of list $\textsc{l}$ and the $\textsc{i}^{\text{th}}$ element of list $\textsc{l}$, respectively. When defining the evaluation function for DOM logical expressions, given a mathematical list L we write $|L|$ for the length of L, and write $|L|^i$ for the i^{th} element of L. It is straightforward to give a formal inductive definition for these constructs and we have omitted them here.

SSL $\mathbb{T}$ Instance (Parameter 8)

Definition 63 (DOM logical expressions)**.** The set of *DOM logical expressions*, $e \in \textsc{LExp}_{\mathbb{DOM}}$, is defined by the following grammar where $\textsc{x}, \alpha, \textsc{l}, \textsc{n}, \textsc{e}, \textsc{f}, \textsc{s} \in \textsc{LVar}$ (Def. 2):

$e ::= \textsc{x}$ Logical variables

$\quad\;\; \mid \alpha$ Context holes

$\mid \lvert \mathrm{L} \rvert$	List length
$\mid \lvert \mathrm{L} \rvert^{\mathrm{I}}$	List element
$\mid \#\mathrm{text}_{\mathrm{N}}[e]_{\mathrm{F}} \mid \mathrm{S}_{\mathrm{N}}[e_1, e_2]^{\mathrm{E}}_{\mathrm{F}}$	Text and element nodes
$\mid \mathrm{S}_{\mathrm{N}}[e]_{\mathrm{F}} \mid \#\mathrm{doc}_{\mathrm{N}}[e_1]^{\mathrm{E}}_{\mathrm{F}}\ \&\ e_2$	Attribute and document nodes
$\mid \varnothing_e$	Empty document element
$\mid \varnothing_s \mid e_1.e_2$	Strings
$\mid \varnothing_f \mid e_1 \otimes e_2 \mid \varnothing_a \mid e_1 \odot e_2$	Forests and attribute sets
$\mid \varnothing_{tf} \mid e_1 \oslash e_2 \mid \varnothing_g \mid e_1 \oplus e_2$	Text forests and groves

The *evaluation function* for DOM logical expressions, $\llparenthesis . \rrparenthesis^{(.)}_{\mathbb{DOM}} : (\mathrm{LExp}_{\mathbb{DOM}} \times \mathrm{LEnv}) \rightharpoonup \mathrm{LVal}_{\mathbb{DOM}}$, is defined inductively over the structure of logical expressions as follows, where $\Gamma \in \mathrm{LEnv}$, $\varnothing_{\dagger} \in \{\varnothing_e, \varnothing_s, \varnothing_f, \varnothing_a, \varnothing_{tf}, \varnothing_g\}$ and $\ddagger \in \{., \otimes, \odot, \oslash, \oplus\}$:

$$\llparenthesis \mathrm{X} \rrparenthesis^{\Gamma}_{\mathbb{DOM}} = \Gamma(\mathrm{X}) \qquad \llparenthesis \alpha \rrparenthesis^{\Gamma}_{\mathbb{DOM}} = \begin{cases} \Gamma(\alpha) & \text{if } \Gamma(\alpha) \in \mathrm{AAdd} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llparenthesis \lvert \mathrm{L} \rvert \rrparenthesis^{\Gamma}_{\mathbb{DOM}} = \begin{cases} n & \text{if } \exists L \in \mathrm{List}\langle \mathrm{Id} \rangle .\ \Gamma(\mathrm{L}) \in L \wedge \lvert L \rvert = n \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llparenthesis \lvert \mathrm{L} \rvert^{\mathrm{I}} \rrparenthesis^{\Gamma}_{\mathbb{DOM}} = \begin{cases} n & \text{if } \exists L \in \mathrm{List}\langle \mathrm{Id} \rangle, i.\ \Gamma(\mathrm{L}) = L \\ & \quad \wedge \Gamma(\mathrm{I}) = i \wedge 0 \leq i < \lvert L \rvert \wedge \lvert L \rvert^{i} = n \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llparenthesis \#\mathrm{text}_{\mathrm{N}}[e]_{\mathrm{F}} \rrparenthesis^{\Gamma}_{\mathbb{DOM}} = \begin{cases} \#\mathrm{text}_n[\mathrm{s}]_{fs} & \text{if } \exists n, fs, \mathrm{s}. \\ & \quad \Gamma(\mathrm{N}) = n \text{ and } \Gamma(\mathrm{F}) = fs \\ & \quad \text{and } \llparenthesis e \rrparenthesis^{\Gamma}_{\mathbb{DOM}} = \mathrm{s} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llparenthesis \mathrm{S}_{\mathrm{N}}[e_1, e_2]^{\mathrm{E}}_{\mathrm{F}} \rrparenthesis^{\Gamma}_{\mathbb{DOM}} = \begin{cases} \mathrm{s}_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs} & \text{if } \exists \mathrm{s}, n, fs, ts, \mathbf{as}, \mathbf{f}. \\ & \quad \Gamma(\mathrm{S}) = \mathrm{s} \text{ and } \Gamma(\mathrm{N}) = n \\ & \quad \text{and } \Gamma(\mathrm{F}) = fs \text{ and } \Gamma(\mathrm{E}) = ts \\ & \quad \text{and } \llparenthesis e_1 \rrparenthesis^{\Gamma}_{\mathbb{DOM}} = \mathbf{as} \\ & \quad \text{and } \llparenthesis e_2 \rrparenthesis^{\Gamma}_{\mathbb{DOM}} = \mathbf{f} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|\mathrm{S}_{\mathrm{N}}[e]_{\mathrm{F}}|\!)^{\Gamma}_{\mathbb{DOM}} = \begin{cases} \mathrm{s}_n[\mathbf{tf}]_{fs} & \text{if } \exists \mathrm{s}, n, fs, \mathbf{tf}. \\ & \quad \Gamma(\mathrm{S}){=}\mathrm{s} \text{ and } \Gamma(\mathrm{N}){=}n \\ & \quad \text{and } \Gamma(\mathrm{F}){=}fs \text{ and } (\!|e|\!)^{\Gamma}_{\mathbb{DOM}}{=}\mathbf{tf} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|\#\mathrm{doc}_{\mathrm{N}}[e_1]^{\mathrm{E}}_{\mathrm{F}} \,\&\, e_2|\!)^{\Gamma}_{\mathbb{DOM}} = \begin{cases} \#\mathrm{doc}_n[\mathbf{d}]^{ts}_{fs} \,\&\, \mathbf{g} & \text{if } \exists n, fs, ts, \mathbf{d}, \mathbf{g}. \Gamma(\mathrm{N}){=}n \\ & \text{and } \Gamma(\mathrm{F}){=}fs \\ & \text{and } \Gamma(\mathrm{E}){=}ts \\ & \text{and } (\!|e_1|\!)^{\Gamma}_{\mathbb{DOM}}{=}\mathbf{d} \\ & \text{and } (\!|e_2|\!)^{\Gamma}_{\mathbb{DOM}}{=}\mathbf{g} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|\varnothing_{\dagger}|\!)^{\Gamma}_{\mathbb{DOM}}{=}\varnothing_{\dagger}$$

$$(\!|e_1 \ddagger e_2|\!)^{\Gamma}_{\mathbb{DOM}} = \begin{cases} \mathbf{d}_1 \ddagger \mathbf{d}_2 & \text{if } (\!|e_1|\!)^{\Gamma}_{\mathbb{DOM}}{=}\mathbf{d}_1 \text{ and } (\!|e_2|\!)^{\Gamma}_{\mathbb{DOM}}{=}\mathbf{d}_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

DOM data assertions Recall that given a library $\mathbb{A}$, the *SSL assertions* for $\mathbb{A}$ comprise heap assertions describing abstract heaps in $\mathrm{LHeap}_{\mathbb{A}}$. Heap assertions in turn are defined via *data assertions*, describing the underlying data in $\mathrm{LData}_{\mathbb{A}}$. Data assertions include generic (library-independent) data assertions that include the standard connectives such as conjunction, as well as library-specific assertions in $\mathrm{LAst}_{\mathbb{A}}$. For the DOM library $\mathbb{DOM}$, the library-specific data assertions comprise assertions to describe DOM nodes, forests, attribute sets, text forests and groves.

Definition 64 (DOM data assertions)**.** Given the DOM-specific data assertions $\mathrm{LAst}_{\mathbb{DOM}}$ (Def. 65), the set of *DOM data assertions* is $\Delta \in \mathrm{DAst}_{\mathbb{DOM}}$, defined as the instantiation of the parametric data assertions $\mathrm{DAst}_{\langle . \rangle}$ (Def. 19) with $\mathrm{LAst}_{\mathbb{DOM}}$.

Given the DOM logical values $\mathrm{LVal}_{\mathbb{DOM}}$ (Def. 62), the logical environments $\mathrm{LEnv}\langle \mathrm{LVal}_{\mathbb{DOM}} \rangle$ (Def. 2) and the DOM logical data $\mathrm{LData}_{\mathbb{DOM}}$ (Def. 58), the *satisfiability relation for DOM data assertions*, $\models_{\mathbb{DOM}}$: $(\mathrm{LEnv}\langle \mathrm{LVal}_{\mathbb{DOM}} \rangle \times \mathrm{LData}_{\mathbb{DOM}}) \times \mathrm{DAst}_{\mathbb{DOM}}$, is defined as the instantiation of the parametric satisfiability relation $|\models_{\langle . \rangle}$ (Def. 20) with $\textsc{LVal}_{\mathbb{DOM}}$, $\textsc{LData}_{\mathbb{DOM}}$ and $\textsc{DAst}_{\mathbb{DOM}}$.

SSL $\mathbb{T}$ Instance (Parameter 9)

Definition 65 (DOM-specific data assertions)**.** The set of *DOM-specific data assertions*, $\Lambda \in \textsc{LAst}_{\mathbb{DOM}}$, is defined by the following grammar, where $\alpha, \textsc{n}, \textsc{s}, \textsc{e}, \textsc{f} \in \textsc{LVar}$ (Def. 2) and $\Delta, \Delta_1, \Delta_2 \in \textsc{DAst}_{\mathbb{DOM}}$ (Def. 64):

$\Lambda ::= \alpha$	Context hole
$\mid \#\text{text}_{\textsc{n}}[\Delta]_{\textsc{f}} \mid \textsc{s}_{\textsc{n}}[\Delta_1, \Delta_2]^{\textsc{e}}_{\textsc{f}}$	Text and element nodes
$\mid \textsc{s}_{\textsc{n}}[\Delta]_{\textsc{f}} \mid \#\text{doc}_{\textsc{n}}[\Delta_1]^{\textsc{e}}_{\textsc{f}} \ \& \ \Delta_2$	Attribute and document nodes
$\mid \varnothing_e$	Empty document element
$\mid \varnothing_s \mid \textsc{s} \mid \Delta_1.\Delta_2$	Strings
$\mid \varnothing_f \mid \Delta_1 \otimes \Delta_2 \mid \varnothing_a \mid \Delta_1 \odot \Delta_2$	Forests and attribute sets
$\mid \varnothing_{tf} \mid \Delta_1 \oslash \Delta_2 \mid \varnothing_g \mid \Delta_1 \oplus \Delta_2$	Text forests and groves

Given the DOM logical values $\textsc{LVal}_{\mathbb{DOM}}$ (Def. 62), the logical environments $\textsc{LEnv}\langle \textsc{LVal}_{\mathbb{DOM}} \rangle$ (Def. 2), the DOM logical data $\textsc{LData}_{\mathbb{DOM}}$ (Def. 58) and the DOM data assertion satisfiability relation $|\models_{\mathbb{DOM}}$ (Def. 64), the *satisfiability relation for DOM-specific data assertions*, $||\models_{\mathbb{DOM}}$: $(\textsc{LEnv}\langle \textsc{LVal}_{\mathbb{DOM}} \rangle \times \textsc{LData}_{\mathbb{DOM}}) \times \textsc{LAst}_{\mathbb{DOM}}$, is defined as follows, for all $\Gamma \in \textsc{LEnv}\langle \textsc{LVal}_{\mathbb{DOM}} \rangle$ and $\mathbf{d} \in \textsc{LData}_{\mathbb{DOM}}$, where $\varnothing_\dagger \in \{\varnothing_f, \varnothing_a, \varnothing_{tf}, \varnothing_g\}$ and $\ddagger \in \{\otimes, \odot, \oslash, \oplus\}$:

$\Gamma, \mathbf{d} \ ||\models_{\mathbb{DOM}} \alpha$ iff $\Gamma(\alpha) = \mathbf{d} \wedge \mathbf{d} \in \textsc{AAdd}$

$\Gamma, \mathbf{d} \ ||\models_{\mathbb{DOM}} \#\text{text}_{\textsc{n}}[\Delta]_{\textsc{f}}$ iff $\exists \text{s}, n, fs.\ \Gamma(\textsc{n}){=}n \wedge \Gamma(\textsc{f}){=}fs$
$\wedge\, \mathbf{d}{=}\#\text{text}_n[\text{s}]_{fs} \wedge \Gamma, s \ |\models_{\mathbb{DOM}} \Delta$

$\Gamma, \mathbf{d} \ ||\models_{\mathbb{DOM}} \textsc{s}_{\textsc{n}}[\Delta_1, \Delta_2]^{\textsc{e}}_{\textsc{f}}$ iff $\exists \text{s}, n, \mathbf{as}, \mathbf{f}, fs, ts.\ \Gamma(\textsc{s}){=}\text{s} \wedge \Gamma(\textsc{n}){=}n$
$\wedge\, \Gamma(\textsc{f}){=}fs \wedge \Gamma(\textsc{e}){=}ts$
$\wedge\, \mathbf{d}{=}\text{s}_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs}$
$\wedge\, \Gamma, \mathbf{as} \ |\models_{\mathbb{DOM}} \Delta_1$
$\wedge\, \Gamma, \mathbf{f} \ |\models_{\mathbb{DOM}} \Delta_2$

$$\begin{array}{lll}
\Gamma, \mathbf{d} \Vert\!\models_{\mathbb{DOM}} \textsc{s}_{\textsc{n}}[\Delta]_{\textsc{f}} & \text{iff} & \exists \mathrm{s}, n, \mathbf{tf}, fs.\ \Gamma(\textsc{s}){=}\mathrm{s} \wedge \Gamma(\textsc{n}){=}n \\
 & & \wedge\, \Gamma(\textsc{f}){=}fs \wedge \mathbf{d}{=}\mathrm{s}_n[\mathbf{tf}]_{fs} \\
 & & \wedge\, \Gamma, \mathbf{tf} \Vert\!\models_{\mathbb{DOM}} \Delta \\
\Gamma, \mathbf{d} \Vert\!\models_{\mathbb{DOM}} \#\mathrm{doc}_{\textsc{n}}[\Delta_1]^{\textsc{e}}_{\textsc{f}} \,\&\, \Delta_2 & \text{iff} & \exists n, \mathbf{d}', \mathbf{g}, fs, ts.\ \Gamma(\textsc{n}){=}n \wedge \Gamma(\textsc{f}){=}fs \\
 & & \wedge\, \Gamma(\textsc{e}){=}ts \\
 & & \wedge\, \mathbf{d}{=}\#\mathrm{doc}_n[\mathbf{d}']^{ts}_{fs} \,\&\, \mathbf{g} \\
 & & \wedge\, \Gamma, \mathbf{d}' \Vert\!\models_{\mathbb{DOM}} \Delta_1 \\
 & & \wedge\, \Gamma, \mathbf{g} \Vert\!\models_{\mathbb{DOM}} \Delta_2 \\
\Gamma, \mathbf{d} \Vert\!\models_{\mathbb{DOM}} \varnothing_{\dagger} & \text{iff} & \mathbf{d} = \varnothing_{\dagger} \\
\Gamma, \mathbf{d} \Vert\!\models_{\mathbb{DOM}} \Delta_1 \ddagger \Delta_2 & \text{iff} & \exists \mathbf{d}_1, \mathbf{d}_2.\ \mathbf{d}{=}\mathbf{d}_1 \ddagger \mathbf{d}_2 \\
 & & \wedge\, \Gamma, \mathbf{d}_1 \Vert\!\models_{\mathbb{DOM}} \Delta_1 \\
 & & \wedge\, \Gamma, \mathbf{d}_2 \Vert\!\models_{\mathbb{DOM}} \Delta_2
\end{array}$$

We drop the tag and forest listeners when not relevant to the discussion and write e.g. $\textsc{s}_{\textsc{n}}[\Delta_1, \Delta_2]$ for $\exists \textsc{f}, \textsc{e}.\ \textsc{s}_{\textsc{n}}[\Delta_1, \Delta_2]^{\textsc{e}}_{\textsc{f}}$.

Definition 66 (DOM heap assertions)**.** Given the DOM data assertions $\textsc{DAst}_{\mathbb{DOM}}$ (Def. 64), the set of *DOM heap assertions* is $\textsc{HAst}_{\mathbb{DOM}}$, defined as the instantiation of the parametric heap assertions $\textsc{HAst}_{\langle\cdot\rangle}$ (Def. 19) with $\textsc{DAst}_{\mathbb{DOM}}$.

As before, when the type of data is clear from the context, we drop the subscripts and write $\bullet$ for $\bullet_{\mathbb{DOM}}$ and write $\mathbf{0}$ for $\mathbf{0}_{\mathbb{DOM}}$.

$\mathbb{DOM}$ library specification We give a select number of DOM axioms in Figs. 5.2-5.3, including those of the operations used in the client programs in §5.4. The behaviour of some of the operations is captured by several axioms. We omit the analogous cases here and give the full DOM axiomatisation of our fragment in §A.

As before, the assertions in the pre- and postconditions of axioms make use of the $*$ connective as well as the $\mathsf{vars}(\dots) * \Theta$ where the vars predicate describes the values associated with program variables, and Θ is an SSL assertion that describes the operation footprint. However, recall that the SSL assertion language does not include the $*$ connective or the vars predicate for describing a variable store. As we aim to incorporate our DOM SSL specification into JSLogic as an add-on, The $*$ connective and the vars predicate used in the specifications of Figs. 5.2-5.3 are those of JSLogic presented in the following section. We proceed with a description of the axioms.

`r := n.nodeValue`: when `n`$=$N identifies a DOM node, the value of node N is returned in `r`. The axiom in Fig. 5.2 shows the case for when N identifies a text node with value S.

`r := n.childNodes`: when `n`$=$N, this operation returns the identifier of a forest listener NodeList F associated with node N. Fig. 5.2 shows the axiom for when N identifies an element node. When asked for a forest listener NodeList, a node may either return an existing one, or generate a fresh one and extend its set with it. This flexibility is due to an under-specification in the standard. As such, in the postcondition the original set F_1 is extended to F_2 ($F_1 \subseteq F_2$) with the return value $F \in F_2$. Since the operation footprint is limited to the child forest of N, the attributes of N have been framed off leaving behind the context hole β. The pure assertion $\mathsf{TIDs}(T, L)$ states that the top-level node identifiers (from left to right) of the forest denoted by T correspond to the list L. For instance, $\mathsf{TIDs}(T, [9, 6])$ holds in Fig. 5.1a when T denotes the child forest of node 4 (named "body"). Note that $\mathsf{TIDs}(T, L)$ implies that there are no context holes at the top level of T. As such, the $\mathsf{TIDs}(T, L)$ stipulates that the forest T contain enough resource for compiling a list L of the immediate children of N. The $\mathsf{TIDs}(T, L)$ is a derived assertion defined in Fig. 5.4. It is defined inductively, following the structure of DOM forests (or text forests).

`r := n.firstChild`: when `n`$=$N identifies a DOM node, the identifier of the first child of N is returned in `r`. The axioms in Fig. 5.2 show the cases for when N identifies a document node. In the first axiom the document node has a child (i.e. the document element) with identifier M. In the second axiom the document has no children (i.e. the document element is empty) and thus `null` is returned. The operation footprint is limited to node N and its child M. As such, the grove of N, as well as the child forest and attributes of M have been framed off leaving behind the context holes δ, β and γ, respectively.

`r := n.nextSibling`: when `n`$=$N identifies a DOM node, the identifier of its right sibling is returned. Fig. 5.2 shows some of the cases for when N identifies a text node. In the first axiom, N has a right sibling (R); in the

$$\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{r}{:}\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\}\ \mathtt{r := n.nodeValue}\ \{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{r}{:}\mathrm{S}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}_1} * \mathsf{TIDs}(\mathrm{T}, \mathrm{L})\}\\ \mathtt{r := n.childNodes}\\ \{\exists \mathrm{F}, \mathrm{F}_2.\ \mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{F}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}_2} * \mathrm{F} \dot{\in} \mathrm{F}_2 * \mathrm{F}_1 \dot{\subseteq} \mathrm{F}_2\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S}_{\mathrm{M}}[\beta,\gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}}\ \&\ \delta\}\\ \mathtt{r := n.firstChild}\\ \{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}}\ \&\ \delta\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}}\ \&\ \delta\}\\ \mathtt{r := n.firstChild}\\ \{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}}\ \&\ \delta\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}} \otimes \mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}'}\}\\ \mathtt{r := n.nextSibling}\\ \{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}} \otimes \mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}'}\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}}]^{\mathrm{E}}_{\mathrm{F}'}\}\\ \mathtt{r := n.nextSibling}\\ \{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}}]^{\mathrm{E}}_{\mathrm{F}'}\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \varnothing_g\}\\ \mathtt{r := u.removeChild(o)}\\ \{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{O}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'}\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta_1 \oslash \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'} \oslash \beta_2]_{\mathrm{F}} * \gamma \mapsto \varnothing_g\}\\ \mathtt{r := u.removeChild(o)}\\ \{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{O}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta_1 \oslash \beta_2]_{\mathrm{F}} * \gamma \mapsto \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'}\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma]^{\mathrm{E}1}_{\mathrm{F}_1} * \delta \mapsto \mathrm{S}'_{\mathrm{N}}[\epsilon, \mathrm{T}]^{\mathrm{E}2}_{\mathrm{F}_2} * \mathsf{complete}(\mathrm{T})\}\\ \mathtt{r := u.appendChild(n)}\\ \{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma \otimes \mathrm{S}'_{\mathrm{N}}[\epsilon, \mathrm{T}]^{\mathrm{E}2}_{\mathrm{F}_2}]^{\mathrm{E}1}_{\mathrm{F}_1} * \delta \mapsto (\varnothing_f \vee \varnothing_g)\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathsf{TIDs}(\mathrm{T}, \mathrm{L}) * \mathrm{F} \dot{\in} \mathrm{F}'\}\\ \mathtt{r := f.length()}\\ \{\exists \mathrm{R}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathrm{R} \dot{=} |\mathrm{L}|\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathsf{srch}(\mathrm{T}, \mathrm{S}', \mathrm{L}) * (\mathrm{S}', \mathrm{F}) \dot{\in} \mathrm{E}\}\\ \mathtt{r := f.length()}\\ \{\exists \mathrm{R}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathrm{R} \dot{=} |\mathrm{L}|\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T} \oslash \gamma]_{\mathrm{F}'} * \mathsf{TIDs}(\mathrm{T}, \mathrm{L}) * 0 \dot{\leq} \mathrm{I} \dot{<} |\mathrm{L}| * \mathrm{F} \dot{\in} \mathrm{F}'\}\\ \mathtt{r := f.item(i)}\\ \{\exists \mathrm{R}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T} \oslash \gamma]_{\mathrm{F}'} * \mathrm{R} \dot{=} |\mathrm{L}|^{\mathrm{I}}\}\end{gathered}$$

$$\begin{gathered}\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'}\ \&\ \beta * \mathsf{srch}(\mathrm{T}, \mathrm{S}, \mathrm{L}) * (\mathrm{S}, \mathrm{F}) \dot{\in} \mathrm{E} * 0 \dot{\leq} \mathrm{I} \dot{<} |\mathrm{L}|\}\\ \mathtt{r := f.item(i)}\\ \{\exists \mathrm{R}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'}\ \&\ \beta * \mathrm{R} \dot{=} |\mathrm{L}|^{\mathrm{I}}\}\end{gathered}$$

Figure 5.2.: $\mathbb{DOM}$ Node and NodeList axioms (excerpt)

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{c}:\mathrm{C},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1.\mathrm{S}_2.\mathrm{S}_3]_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{S}_1| * \mathrm{C} \doteq |\mathrm{S}_2|\right\}\\
\texttt{r := n.substringData(o, c)}\\
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{c}:\mathrm{C},\mathtt{r}:\mathrm{S}_2) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1.\mathrm{S}_2.\mathrm{S}_3]_{\mathrm{F}}\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{c}:\mathrm{C},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1.\mathrm{S}_2]_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{S}_1| * \mathrm{C} \dot{\geq} |\mathrm{S}_2|\right\}\\
\texttt{r := n.substringData(o, c)}\\
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{c}:\mathrm{C},\mathtt{r}:\mathrm{S}_2) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1.\mathrm{S}_2]_{\mathrm{F}}\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}.\mathrm{S}']_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{S}|\right\}\\
\texttt{r := n.splitText(o)}\\
\left\{\exists \mathrm{R},\mathrm{F}'.\ \mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \otimes \#\mathrm{text}_{\mathrm{R}}[\mathrm{S}']_{\mathrm{F}'}\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{S}'_{\mathrm{M}}[\mathrm{T}]_{\mathrm{F}'},\gamma]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{val}(\mathrm{T},\mathrm{S}'')\right\}\\
\texttt{r := n.getAttribute(s)}\\
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{r}:\mathrm{S}'') * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{S}'_{\mathrm{M}}[\mathrm{T}]_{\mathrm{F}'},\gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A},\gamma]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{out}_{\mathsf{n}}(\mathrm{A},\mathrm{S}')\right\}\\
\texttt{r := n.getAttribute(s)}\\
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{r}:\text{``''}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A},\gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{v}:\mathrm{S}'') * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{S}'_{\mathrm{M}}[\mathrm{T}]_{\mathrm{F}'},\gamma]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \varnothing_g * \mathsf{grove}(\mathrm{T},\mathrm{G})\right\}\\
\texttt{n.setAttribute(s, v)}\\
\left\{\exists \mathrm{R},\mathrm{F}''.\ \mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{v}:\mathrm{S}'') * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{S}'_{\mathrm{M}}[\#\mathrm{text}_{\mathrm{R}}[\mathrm{S}'']_{\mathrm{F}''}]_{\mathrm{F}'},\gamma]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \mathrm{G}\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{v}:\mathrm{S}'') * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A},\gamma]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{out}_{\mathsf{n}}(\mathrm{A},\mathrm{S}')\right\}\\
\texttt{n.setAttribute(s, v)}\\
\left\{\exists \mathrm{R},\mathrm{M},\mathrm{F}',\mathrm{F}''.\ \mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{v}:\mathrm{S}'') * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A} \odot \mathrm{S}'_{\mathrm{M}}[\#\mathrm{text}_{\mathrm{R}}[\mathrm{S}'']_{\mathrm{F}''}]_{\mathrm{F}'},\gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{a}:\mathrm{A},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{S}'_{\mathrm{A}}[\mathrm{S}'']_{\mathrm{F}'},\gamma]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \varnothing_g\right\}\\
\texttt{r := n.removeAttributeNode(a)}\\
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{a}:\mathrm{A},\mathtt{r}:\mathrm{A}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta,\gamma]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \mathrm{S}'_{\mathrm{A}}[\mathrm{S}'']_{\mathrm{F}'}\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\beta,\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{srch}(\mathrm{T},\mathrm{S},\mathrm{L})\right\}\\
\texttt{r := n.getElementsByTagName(s)}\\
\left\{\exists \mathrm{R},\mathrm{E}'.\ \mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\beta,\mathrm{T}]^{\mathrm{E}'}_{\mathrm{F}} * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\mathrm{S},\mathrm{R}) \in \mathrm{E}'\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}} * \mathsf{val}(\mathrm{T},\mathrm{S}')\right\}\\
\texttt{r := n.value}\\
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{S}') * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S}_{\mathrm{M}}[\beta,\gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \ \&\ \delta\right\}\\
\texttt{r := n.documentElement}\\
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{M}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S}_{\mathrm{M}}[\beta,\gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \ \&\ \delta\right\}
\end{gathered}$$

$$\begin{gathered}
\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \ \&\ \gamma * \mathsf{safe}(\mathrm{S})\right\}\\
\texttt{r := n.createAttribute(s)}\\
\left\{\exists \mathrm{R},\mathrm{F}'.\ \mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \ \&\ \gamma \oplus \mathrm{S}_{\mathrm{R}}[\varnothing_{tf}]_{\mathrm{F}'}\right\}
\end{gathered}$$

Figure 5.3.: $\mathbb{DOM}$ text, element, attribute and document axioms (excerpt)

second, N has no right sibling and is the last child of its parent (U).

`r := u.removeChild(o)` : when `u`=U and `o`=O both identify DOM nodes, this operation removes O from the child list of `u`, moves O to the document grove (DOM nodes are never deleted; orphaned nodes are simply added to the grove) and then returns O in `r`. Fig. 5.2 depicts the axiom for removing a text node (O) from an element node (U). The $\delta \mapsto \varnothing_g$ assertion in the precondition simply reserves an empty spot in the grove (denoted by $\varnothing_g$). In the postcondition, node O is removed from U and added to the grove.

`r := u.appendChild(n)`: when `u`=U and `n`=N both identify DOM nodes, this operation appends N to the end of U's child list and returns N in `r`. Recall that this operation fails if N is an ancestor of U (otherwise it would introduce a cycle and break the DOM structure). Fig. 5.2 shows the axiom for when N identifies an element node. As with the `appendChild` operation of the tree library $\mathbb{T}$ studied in §4, to ensure that N is not an ancestor of U, we require the entire subtree at O to be *separate* from the subtree at N. This is achieved by the $\mathsf{complete}(\text{T})$ assertion and the separating conjunction $*$. The $\mathsf{complete}(\text{T})$ is a derived assertion defined in Fig. 5.4 and describes DOM data with no context holes. The postcondition leaves $\varnothing_f \vee \varnothing_g$ in place of N once moved since we do not know if O has come from a forest or grove position. The disjunction leaves the choice to the frame.

`r := f.length`: when `f`=F identifies a NodeList, its length is returned in `r`. The first axiom in Fig. 5.2 describes the case when F identifies a forest listener NodeList on element node N; the return value is the number of N's immediate children. This is captured by the $\mathsf{TIDs}(\text{T}, \text{L})$ assertion (defined in Fig. 5.4) stating that list L contains the identifiers of the immediate children of N. The return value is thus the length of L ($|\text{L}|$). Analogously, the second axiom describes the case when F identifies a tag listener NodeList on N, listening for tag name S'. The return value in this case is the length of list L corresponding to a pre-order (depth-first, left-to-right) search for string S' on the child forest of N. The pure assertion $\mathsf{srch}(\text{T}, \text{S}', \text{L})$ states that child forest T (underneath N) contains enough resource for compiling a pre-order list L of elements named S. The $\mathsf{srch}(\text{T}, \text{S}', \text{L})$ assertion holds when either i) S'=“*” denoting a wildcard and L contains the identifiers of *all* element nodes in forest T; or ii) $\text{S}' \neq$“*” and L contains the identifiers of those element nodes in forest T whose

$$\begin{aligned}
\mathsf{complete}(\mathrm{T}) &\triangleq (\mathrm{T}\doteq\varnothing_f) \vee \big(\exists \mathrm{N}, \mathrm{T}'.\ \mathrm{T}\doteq\#\mathrm{text}_{\mathrm{N}}[-] \otimes \mathrm{T}' * \mathsf{complete}(\mathrm{T}')\big)\\
&\quad \vee \big(\exists \mathrm{S}',\mathrm{N},\mathrm{F},\mathrm{T}',\mathrm{L}'.\ \mathrm{T}\doteq\mathrm{S}'_{\mathrm{N}}[-, \mathrm{F}] \otimes \mathrm{T}' * \mathsf{complete}(\mathrm{F}) * \mathsf{complete}(\mathrm{T}')\big)\\
&\quad \vee (\mathrm{T}\doteq\varnothing_{tf}) \vee (\exists \mathrm{N}, \mathrm{T}'.\, \mathrm{T}\doteq\#\mathrm{text}_{\mathrm{N}}[-] \oslash \mathrm{T}' * \mathsf{complete}(\mathrm{T}')\big)\\[4pt]
\mathsf{TIDs}(\mathrm{F}, \mathrm{L}) &\triangleq (\mathrm{F}\doteq\varnothing_f * \mathrm{L}\doteq[\,]) \vee (\exists \mathrm{N}, \mathrm{F}', \mathrm{L}'.\ \mathrm{F}\doteq\#\mathrm{text}_{\mathrm{N}}[-]\otimes\mathrm{F}' * \mathsf{TIDs}(\mathrm{F}', \mathrm{L}') * \mathrm{L}\doteq\mathrm{N}{:}\mathrm{L}')\\
&\quad \vee \big(\exists \mathrm{S}',\mathrm{N},\mathrm{F}',\mathrm{L}'.\ \mathrm{F}\doteq\mathrm{S}'_{\mathrm{N}}[-, -]\otimes\mathrm{F}' * \mathsf{TIDs}(\mathrm{F}', \mathrm{L}') * \mathrm{L}\doteq\mathrm{N}{:}\mathrm{L}')\\
&\quad \vee (\mathrm{F}\doteq\varnothing_{tf} * \mathrm{L}\doteq[\,]) \vee (\exists \mathrm{N}, \mathrm{F}',\mathrm{L}'.\, \mathrm{F}\doteq\#\mathrm{text}_{\mathrm{N}}[-]\oslash\mathrm{F}' * \mathsf{TIDs}(\mathrm{F}', \mathrm{L}') * \mathrm{L}\doteq\mathrm{N}{:}\mathrm{L}')\\[4pt]
\mathsf{srch}(\mathrm{F}, \mathrm{S}, \mathrm{L}) &\triangleq (\mathrm{F}\doteq\varnothing_f * \mathrm{L}\doteq[\,]) \vee (\exists \mathrm{N}, \mathrm{F}'.\ \mathrm{F}\doteq\#\mathrm{text}_{\mathrm{N}}[-]\otimes\mathrm{F}' * \mathsf{srch}(\mathrm{F}', \mathrm{S}, \mathrm{L}))\\
&\quad \vee \big(\exists \mathrm{S}',\mathrm{N},\mathrm{F}_1,\mathrm{F}_2,\mathrm{L}_1,\mathrm{L}_2.\, \mathrm{F}\doteq\mathrm{S}'_{\mathrm{N}}[-, \mathrm{F}_1]\otimes\mathrm{F}_2 * \mathsf{srch}(\mathrm{F}_1, \mathrm{S}, \mathrm{L}_1) * \mathsf{srch}(\mathrm{F}_2, \mathrm{S}, \mathrm{L}_2)\\
&\qquad * (\mathrm{S}\doteq\mathrm{S}' \vee \mathrm{S}\doteq\text{“ * ”} \Rightarrow \mathrm{L}\doteq\mathrm{N}{:}(\mathrm{L}_1 +\!\!+ \mathrm{L}_2)) * (\mathrm{S}\,\dot{\neq}\,\mathrm{S}' \wedge \mathrm{S}\,\dot{\neq}\,\text{“ * ”} \Rightarrow \mathrm{L}\doteq\mathrm{L}_1 +\!\!+ \mathrm{L}_2)\big)\\[4pt]
\mathsf{val}(\mathrm{T}, \mathrm{S}) &\triangleq (\mathrm{T}\doteq\varnothing_{tf} * \mathrm{S}\doteq\text{“”})\\
&\quad \vee (\exists \mathrm{N}, \mathrm{S}_1, \mathrm{S}_2, \mathrm{T}'.\ \mathrm{T}\doteq\#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1] \oslash \mathrm{T}' * \mathsf{val}(\mathrm{T}', \mathrm{S}_2) * \mathrm{S}\doteq\mathrm{S}_1.\mathrm{S}_2)\\[4pt]
\mathsf{out}_{\mathsf{n}}(\mathrm{A}, \mathrm{S}) &\triangleq (\mathrm{A}\doteq\varnothing_a) \vee (\exists \mathrm{S}', \mathrm{N}, \mathrm{A}'.\ \mathrm{A}\doteq\mathrm{S}'_{\mathrm{N}}[-] \odot \mathrm{A}' * \mathrm{S}\,\dot{\neq}\,\mathrm{S}' * \mathsf{out}_{\mathsf{n}}(\mathrm{A}', \mathrm{S}))\\[4pt]
\mathsf{out}_{\mathsf{id}}(\mathrm{A}, \mathrm{N}) &\triangleq (\mathrm{A}\doteq\varnothing_a) \vee (\exists \mathrm{S}, \mathrm{N}', \mathrm{A}'.\ \mathrm{A}\doteq\mathrm{S}_{\mathrm{N}'}[-] \odot \mathrm{A}' * \mathrm{N}\,\dot{\neq}\,\mathrm{N}' * \mathsf{out}_{\mathsf{id}}(\mathrm{A}', \mathrm{N}))\\[4pt]
\mathsf{grove}(\mathrm{TF},\mathrm{G}) &\triangleq (\mathrm{TF}\doteq\varnothing_{tf} \wedge \mathrm{G}\doteq\varnothing_g) \vee\\
&\quad \big(\exists \mathrm{N},\mathrm{S},\mathrm{F},\mathrm{TF}',\mathrm{G}'.\ \mathrm{TF}\doteq\#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \oslash \mathrm{TF}' * \mathsf{grove}(\mathrm{T}',\mathrm{G}') * \mathrm{G}\doteq\#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \oplus \mathrm{G}'\big)\\[4pt]
\mathsf{safe}(\mathrm{S}) &\triangleq \neg\exists \mathrm{S}_1, \mathrm{S}_2.\ \mathrm{S}{=}\mathrm{S}_1.\text{“\#”}.\mathrm{S}_2
\end{aligned}$$

Figure 5.4.: Derived assertions for $\mathbb{DOM}$

names are equal to S. For instance, when T denotes the child forest of node 9 (named “ad”) in Fig. 5.1a, then $\mathsf{srch}(\mathrm{T}, \text{img”}, [3, 8])$ holds. The $\mathsf{srch}(\mathrm{T}, \mathrm{S}', \mathrm{L})$ is a derived assertion defined in Fig. 5.4. This predicate is defined inductively, following the structure of DOM forests.

`r := f.item(i)`: when `f`=F identifies a NodeList, the i^th^ element of F is returned in `r`. The axioms in Fig. 5.2 are analogous to those of `r := f.length` with $|\mathrm{L}|^{\mathrm{I}}$ denoting the Ith item of list L.

`r := n.substringData(o,c)`: when `n`=N identifies a text node with value S and both `o`=O and `c`=C denote integer values, the substring of S between index O (inclusive) and index O+C (exclusive) is returned in `r`. If O+C exceeds the length of S, then the substring from index O (inclusive) to the end of S is returned. The first axiom in Fig. 5.3 describes the case when O+C does not exceed the length of S. The second axioms describes the case when O+C does exceed the length of S.

`r := n.splitText(o)`: when `n`=N identifies a text node and `o`=O denotes an integer, the data of N is split into two text nodes at offset O (indexed from 0), keeping both nodes in the tree as siblings and returning the identifier of the new text node (i.e. $\textsc{r}$) in `r`.

`r := n.getAttribute(s)`: when `n`=$\textsc{n}$ identifies an element node and `s`=$\textsc{s}'$ denotes a string value, if the attribute set of $\textsc{n}$ contains an attribute named $\textsc{s}'$, then its value is returned in `r`; otherwise `null` is returned. The first axiom in Fig. 5.3 shows the case when the attribute set of $\textsc{n}$ contains an attribute named $\textsc{s}$' with its value captured by the text forest $\textsc{t}$. The pure assertion $\mathsf{val}(\textsc{t}, \textsc{s}'')$ in the precondition states that the value denoted by text forest $\textsc{t}$ (i.e. the concatenated values of the text nodes in $\textsc{t}$) is $\textsc{s}$". The $\mathsf{val}(\textsc{t}, \textsc{s}'')$ is a derived assertion defined in Fig. 5.4. This predicate is defined inductively, following the structure of DOM text forests. Analogously, the second axiom in Fig. 5.3 shows the case when the attribute set of $\textsc{n}$ (namely $\textsc{a}$) does not contain an attribute named $\textsc{s}$', as described by the pure assertion $\mathsf{out_n}(\textsc{a}, \textsc{s})$. The $\mathsf{out_n}(\textsc{a}, \textsc{s})$ is also a derived assertion defined inductively in Fig. 5.4, following the structure of DOM text forests. The analogous pure assertion $\mathsf{out_{id}}(\textsc{a}, \textsc{n})$ in Fig. 5.4 states that the attribute set $\textsc{a}$ does not contain an attribute with identifier $\textsc{n}$. This assertion is used to specify the behaviour of the analogous `r := getAttributeNode` operation (see §A).

`n.setAttribute(s, v)`: when `n`=$\textsc{n}$ identifies an element node and `s`=$\textsc{s}'$ and `v`=$\textsc{s}''$ denote string values, if the attribute set of $\textsc{n}$ contains an attribute named $\textsc{s}'$, then its value is set to $\textsc{s}''$; otherwise a new attribute with name $\textsc{s}'$ and value $\textsc{s}''$ is added to the attribute set of $\textsc{n}$. Recall that attribute nodes may have an arbitrary number of text nodes as their children with the concatenated values of the text nodes denoting the value of the attribute. As such, when $\textsc{n}$ contains an attribute named $\textsc{s}'$ (first axiom), its value is set to $\textsc{s}''$ by removing its existing children, creating a new text node with value $\textsc{s}''$ and attaching it to $\textsc{s}'$. The $\delta \mapsto \varnothing_g$ assertion in the precondition reserves an empty spot ($\varnothing_g$) in the grove for the children of $\textsc{s}'$ which are to be removed. The pure assertion $\mathsf{grove}(\textsc{tf}, \textsc{g})$ simply coerces the ordered set of text nodes in the text forest $\textsc{tf}$ to an unordered set of nodes in the grove $\textsc{g}$. The $\mathsf{grove}(\textsc{t}, \textsc{g})$ is a derived assertion defined in Fig. 5.4. In the postcondition, the removed nodes in $\textsc{g}$ are moved to the empty grove spot at δ. On the other hand, when $\textsc{n}$ does not contain an attribute named $\textsc{s}'$ (second axiom), its attribute set ($\textsc{a}$) is extended with a new attribute named $\textsc{s}'$. The value of $\textsc{s}'$ in turn is set to $\textsc{s}''$ by creating a new text node with value $\textsc{s}''$ and attaching it to $\textsc{s}'$.

`r := n.removeAttributeNode(a)`: when `n`=N identifies an element node and `a`=A denotes an attribute node, this operation removes A from the attribute set of N, moves A to the document grove and then returns A in `r`. As before, the $\delta \mapsto \varnothing_g$ assertion in the precondition reserves an empty spot ($\varnothing_g$) in the grove. In the postcondition, node A is removed from the attribute set of N and moved to the grove.

`r := n.getElementsByTagName(s)`: when `n`=N identifies an element or document node and `s`=S denotes a string value, this operation returns a tag listener NodeList listening on the forest underneath N for element nodes named S. The axiom in Fig. 5.3 describes the case when N identifies an element node. As with the `r := n.childNodes` operation described above, due to an under-specification in the standard, when asked for a tag listener NodeList a node may either return an existing one, or generate a fresh one and extend its set with it. Thus, in the postcondition the original set E is extended to E′ ($\text{E} \subseteq \text{E}'$). The return value is a NodeList identified by some R such that $(\text{S}, \text{R}) \in \text{E}'$.

`r := n.value`: when `n`=N identifies an attribute node, the value of node N is returned in `r`.

`r := n.documentElement`: when `n`=N identifies a document node, the identifier of its document element is returned in `r` when it exists; otherwise `null` is returned. The axiom in Fig. 5.3 shows the case when the document element of node N is not empty.

`r := n.createAttribute(s)`: when `n`=N identifies a document node and `s`=S denotes a safe string value (one not containing the invalid character '#'), then a new attribute node named S is created (in the grove) and its identifier is returned in `r`. The new attribute has no children (i.e. its text forest is empty) and resides in the grove. The original grove in the precondition (γ) is thus extended with the new node in the postcondition. The pure assertion safe(S) is defined in Fig. 5.4 and states that the name S is safe.

Comparison to existing work (locality) The DOM specification in the existing work [24, 52] does not provide small enough axioms for all operations. In particular, the axioms of the `insertBefore`, `replaceChild` and `appendChild` operations require substantial *over-approximations* of their respective footprints. This was indeed noted by the authors at the time. Consider the axiom of the `u.appendChild(n)` operation given below using multi-holed context logic [6] (adapted to our notation)[4]:

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * (C \bullet_{\alpha} \mathrm{S}_{\mathrm{U}}[\mathrm{A}, \gamma]^{\mathrm{E}}_{\mathrm{F}}) \bullet_{\beta} \mathrm{S}'_{\mathrm{N}}[\mathrm{A}', \mathrm{T}]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$
$$\mathtt{r} := \mathtt{u.appendChild(n)}$$
$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) * (C \bullet_{\alpha} \mathrm{S}_{\mathrm{U}}[\mathrm{A}, \gamma \otimes \mathrm{S}'_{\mathrm{N}}[\mathrm{A}', \mathrm{T}]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}}) \bullet_{\beta} \varnothing_{f}\right\}$$

The precondition specifies that the DOM tree can be split into a subtree with top element node N, and a tree context with context hole variable β satisfying the $C \bullet_{\alpha} \mathrm{S}_{\mathrm{U}}[\mathrm{A}, \gamma]^{\mathrm{E}}_{\mathrm{F}}$ formula. This formula states that the context can be further split into a subcontext with top element node U and a context C with hole α. In the postcondition the tree at N is moved to be the last child of U and its place holder (β) is replaced by the empty forest. The surrounding context, C, remains the same. This axiom is not small enough in that it does not capture the intuitive footprint of `appendChild`. The only part of the tree that `appendChild` requires is the tree at N which is being moved, and the element node U whose children are being extended by N. However, the precondition above also requires the surrounding *linking* context C. This context is needed to describe the entire sub-tree containing both node U and N (provided that N is not an ancestor of U). This results in a significant overapproximation of the footprint. It is possible to put additional constraints on C to insist that it is minimal. But this would still not be small enough.

Comparison to existing work (live collections) In [24, 52] an axiom for `r := n.childNodes` is given as follows (adapted to our notation):

$$\left\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{r}{:}\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta,\gamma]^{\mathrm{E}}_{\mathrm{F}}\right\} \mathtt{r:=n.childNodes} \left\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N},\mathtt{r}{:}\mathrm{F}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta,\gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

where F is a single forest listener. This axiom is too strong. In particular, we can verify the following specification, stating that when the program

[4]The axiom in [24, 52] is given using single-holed context logic [7]. The multi-holed axiom is simpler and smaller, but still not small enough.

terminates we have $\mathtt{x}=\mathtt{y}=\text{F}$.

$$\left\{\mathsf{vars}(\mathtt{n} : \text{N}, \mathtt{x} : \text{X}, \mathtt{y} : \text{Y}) * \alpha \mapsto \text{S}_{\text{N}}[\beta, \gamma]^{\text{E}}_{\text{F}}\right\}$$
$$\mathtt{x := n.childNodes;\ y := n.childNodes}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \text{N}, \mathtt{x} : \text{F}, \mathtt{y} : \text{F}) * \alpha \mapsto \text{S}_{\text{N}}[\beta, \gamma]^{\text{E}}_{\text{F}}\right\}$$

However, this behaviour is not guaranteed by the DOM standard.

We correct this in the axiom of Fig. 5.2 by associating each node with a *set* of forest listeners.

Recall from §3.2 that given a programming language PL with an SL-based program logic PLogic, we extend PLogic with SSL in order to reason about client programs written in PL. In what follows we instantiate the methodology presented in §3.2 to extend the SL-based JavaScript program logic of [21] in order to reason about several DOM client programs written in JavaScript.

5.3. JSLogic$_{\mathbb{DOM}}$ Reasoning Framework

In order to reason about the client programs of DOM, we appeal to the SL-based JavaScript (JS) program logic presented in [21], hereafter referred to as JSLogic. In §3.2 we presented a general methodology for extending an SL-based program logic PLogic to PLogic$_{\mathbb{A}}$ in order to enable $\mathbb{A}$ client reasoning, provided that PLogic meets certain assumptions delineated in grey boxes labelled "PLogic Parameter". We describe how we instantiate this methodology in order to extend JSLogic to JSLogic$_{\mathbb{DOM}}$ and use it to reason about the client programs of the DOM library. As before, we present the various JSLogic components enclosed in dashed boxes labelled "JSLogic Instance (Parameter X)", where X is the reference to the corresponding PLogic parameter. We then use JSLogic$_{\mathbb{DOM}}$ to reason about two ad-blocker programs that call DOM operations.

Recall that PLogic requires several of its components to be defined by *generic constructors* parametric in the choice of their primitive building blocks. For instance, Par. 12 stipulates that the PL operations be defined by a parametric set $\text{Op}\langle\text{O}\rangle$, where O denotes a set of primitive operations such that $\text{O} \subseteq \text{Op}\langle\text{O}\rangle$. As we demonstrated for WLogic in §3.2, we can typically meet these stipulations by defining the required generic constructor via an inductive grammar where the primitive building blocks correspond to the base cases of the grammar. For instance, we define the generic WL operation set $\text{WOp}\langle \text{O} \rangle$ (Def. 27) by an inductive grammar of the form $\texttt{C} ::= o \mid \texttt{C}_1\texttt{;}\texttt{C}_2 \mid \ldots$, where $o \in \text{O}$ and $\texttt{C} \in \text{WOp}\langle \text{O} \rangle$. We then identify the WL operations as either *primitive* (i.e. a base case of the grammar and thus in O, e.g. $\texttt{x} = \texttt{e}$), or *composite* (i.e. an inductive case of the grammar and thus in $\text{WOp}\langle \text{O} \rangle$, e.g. $\texttt{C}_1\texttt{;}\,\texttt{C}_2$).

It is straightforward to follow the same pattern for JSLogic: we can define the sundry generic constructors by parametric inductive grammars and then categorise their constituents as either primitive or composite. We believe this format to be familiar to the reader by now. As such, rather than defining parametric inductive grammars and then distinguishing primitive constituents from composite ones, we opt for simple (non-parametric) inductive grammars to keep the definitions concise. Note that in order to lift a simple inductive grammar to a parametric one, it suffices to replace the base cases of the grammar with $o \in \text{O}$, where O denotes a superset of the base cases (e.g. $\text{O} \supseteq \text{PrimWOp}$ for the WL operations in Def. 27).

JS program values The set of JS program values comprises *basic values*, *memory locations*, and *lambda abstractions*. Basic values include numbers, strings, the special constant `undefined` and the `null` location. Lambda abstractions are used to represent the values of JavaScript functions in the heap and are of the form $\lambda \texttt{x}_1 \cdots \texttt{x}_n \,.\, \texttt{e}$.

JSLogic Instance (Parameter 10)

Definition 67 (JS program values)**.** The set of JS *basic values*, $\texttt{v} \in$ JSBasicVal, is defined by the following grammar where n denotes a number and s denotes a string:

$$\texttt{v} ::= n \mid s \mid \texttt{null} \mid \texttt{undefined}$$

The set of JS *locations* is $l \in \mathcal{L} \subseteq \mathbb{N}^+$. The set of JS *program values*, $v \in$ JSPVal, is defined by the following grammar where

$\mathtt{v} \in \textsc{JSBasicVal}$, $\mathtt{x}_1 \cdots \mathtt{x}_n \in \textsc{PVar}$ (Def. 1) and $\mathtt{e} \in \textsc{JSOp}$ (Def. 69).

$$v ::= \mathtt{v} \mid l \mid \lambda \mathtt{x}_1 \cdots \mathtt{x}_n \,.\, \mathtt{e}$$

JS program states The JS program states are *JavaScript heaps*. A JavaScript heap is a partial function mapping *references*, which are pairs of memory locations and field names, to values. Field names are a subset of program variable names, and do not include keywords such as `var`, or the reserved internal names *@scope*, *@body*, *@proto* and *@this*(described in due course). A heap cell is written $(l, \mathtt{x}) \mapsto 7$, stating that the object at l has a field named `x` and holds value 7.

JSLogic Instance (Parameter 11)

Definition 68 (JS program heaps)**.** Given the set of program variables PVar (Def. 1), the set of JS *field names* is $\mathtt{x} \in \mathcal{X} \subset \textsc{PVar}$. Given the set of JS locations $\mathcal{L}$ (Def. 67), the set of JS *references* is $r \in \textsc{JSRef} \triangleq \mathcal{L} \times \mathcal{X}$.
Let $\mathrm{V} \supseteq \textsc{JSPVal}$ denote an extension of JS program values (Def. 67).
The set of *parametric program heaps* is:

$$h \in \textsc{JSPHeap}\,\langle \mathrm{V} \rangle \triangleq \textsc{JSRef} \overset{\text{fin}}{\rightharpoonup} \mathrm{V}$$

Programming language The JS operations, $\mathtt{e} \in \textsc{JSOp}$, comprise variable lookup (`x`), basic values (`v`), `this`, local variable declaration (`var x`), sequential composition (`e1;e2`), object property lookup (`e.x`), computed access (`e[e']`), assignment (`e1 = e2`), arithmetic and boolean expressions ($\mathtt{e1} \ominus \mathtt{e2}$ where $\ominus \in \{+, -, *, /, \&\&, ||, ==\}$), string concatenation (`e1.e2`), conditional expressions (`if (e) then {e1} else {e2}`), loops (`while (e) {e'}`), function call (`e(e')`), anonymous function declaration (`function (x){e}`), function declaration (`function f(x){e}`), the with statement (`with (e) {e'}`), constructors (`new e1(e2)`), literal object declaration (`{x1:e1...xn:en}`), and deletion (`delete e`).

We define the JS operations by an inductive grammar where `x`, `v`, `this`, and `var x` constitute the base cases, and the remaining operations form the inductive cases. As discussed above, it is straightforward to lift this grammar to a generic one parametric in the choice of its primitive operations O (where $\mathrm{O} \supseteq \{\mathtt{x}, \mathtt{v}, \mathtt{this}, \mathtt{var\ x}\}$ denotes an extension of the bases cases), by replacing the base cases with $o \in \mathrm{O}$.

JSLogic Instance (Parameter 12)

Definition 69 (JS operations)**.** The set of JS *operations*, $\mathtt{e} \in \textsc{JSOp}$, is defined by the following grammar where $\mathtt{x}, \mathtt{f} \in \textsc{PVar}$ (Def. 1), $\mathtt{v} \in \textsc{JSBasicVal}$ (Def. 67) and $\ominus \in \{+, -, *, /, \&\&, ||, ==\}$:

$$\begin{aligned}\mathtt{e} ::= {} & \mathtt{x} \mid \mathtt{v} \mid \mathtt{this} \mid \mathtt{var\ x} \mid \mathtt{e1;e2} \mid \mathtt{e.x} \mid \mathtt{e[e']} \mid \mathtt{e1} = \mathtt{e2} \mid \mathtt{e1} \ominus \mathtt{e2} \\ & \mid \mathtt{e1.e2} \mid \mathtt{if\,(e)\,then\,\{e1\}\,else\,\{e2\}} \mid \mathtt{while(e)\{e'\}} \\ & \mid \mathtt{e(e')} \mid \mathtt{function\,(x)\{e\}} \mid \mathtt{function\,f\,(x)\{e\}} \mid \mathtt{with(e)\{e'\}} \\ & \mid \mathtt{new\ e1(e2)} \mid \mathtt{\{x1:e1...xn:en\}} \mid \mathtt{delete\ e}\end{aligned}$$

Definition 70 ($\mathrm{JS}_{\mathbb{DOM}}$ operations)**.** Given the JS operations JSOp (Def. 69), and the DOM operations $\textsc{Op}_{\mathbb{DOM}}$ (Def. 61), the set of $\mathrm{JS}_{\mathbb{DOM}}$ *operations*, $\textsc{JSOp}_{\mathbb{DOM}}$, is constructed from JSOp and $\textsc{Op}_{\mathbb{DOM}}$ as described in Def. 26.

JS semantics The JS semantics is defined via a big-step operational semantics relation. In $\mathrm{JS}_{\mathbb{DOM}}$ we extend the JS operational semantics to a generic semantics function and extend it with the semantics of the DOM library as described in §3 (Def. 28). As we demonstrated in §3 for WLogic, it is straightforward to lift a big-step operational semantics relation to a generic semantics function. Lifting the JS operational semantics to a generic semantics function is in the same vein and requires no additional machinery. We thus omit the JS semantics here; the detailed operational semantics of JS is given in [21, 22].

Recall that as per the methodology described in §3.2, in $\mathrm{JS}_{\mathbb{DOM}}$ the semantics of JS is extend to incorporate the low-level semantics of the DOM operations. In keeping with the axiomatic spirit of the DOM standard [44], we do not devise a low-level DOM semantics and opt instead for the default semantics provided by Par. 20 in §3. As we mentioned earlier, often when the low-level semantics of a library is not provided, the axiomatic specification of the library operations is justified with respect to an implementation of the library operations. In §7 we provide an implementation of the operations of our DOM fragment and describe how to justify the correctness of our axiomatic DOM specification against this implementation.

JSLogic logical values The set of logical values for JSLogic is defined as the extension of JS program values (Def. 57) with lists, sets, JS operations (expressions) in Def. 69, and a special value $\varnothing$ (read *none*). As we describe shortly, the $\varnothing$ value is used to describe the absence of a field from an object in the JavaScript heap.

JSLogic Instance (Parameter 14)

Definition 71 (JSLogic logical values)**.** The set of *JavaScript logical values*, $v \in \text{JSLVal}$, is defined by the following grammar where $w \in \text{JSPVal}$ (Def. 67), $L \in \text{List}\langle\text{JSLVal}\rangle$, $S \in \mathcal{P}(\text{JSLVal})$ and $\mathsf{e} \in \text{JSOp}$ (Def. 69):

$$v ::= w \mid L \mid S \mid \mathsf{e} \mid \varnothing$$

JSLogic logical states The JSLogic logical states are *logical* JS *heaps*. A logical JS heap is a program heap which may additionally contain the special $\varnothing$ value (read *none*) in its range. That is, a logical JS heap is a partial function mapping references (Def. 68) to program values (Def. 67) or the $\varnothing$ value. As with program heaps, a logical heap cell is written $(l, \mathtt{x}) \mapsto 7$, stating that the object at l has a field named $\mathtt{x}$ and holds value 7. Additionally, the absence of a field is denoted by the special $\varnothing$ value. That is, we write $(l, \mathtt{y}) \mapsto \varnothing$ to state that the object at l has no field named $\mathtt{y}$. Logical heap composition, $\circ$, is defined as the standard disjoint function union. An empty logical heap is defined as the function with an empty domain. For brevity, $(l, \mathtt{x}_1) \mapsto v_1 \circ \ldots \circ (l, \mathtt{x}_n) \mapsto v_n$ is written as $l \mapsto \{\mathtt{x}_1 : v_1, \ldots, \mathtt{x}_n : v_n\}$.

JSLogic Instance (Parameter 15)

Definition 72 (JSLogic partial commutative monoid)**.** Let $\mathrm{V} \supseteq \textsc{JSPVal} \uplus \{\varnothing\}$ denote an extension of JS program values (Def. 67) combined with $\varnothing$.
Given the set of JS references JSRef (Def. 68), the set of *parametric logical heaps* is $h \in \textsc{JSLHeap}\langle \mathrm{V} \rangle \triangleq \textsc{JSRef} \overset{\text{fin}}{\rightharpoonup} \mathrm{V}$.
The *parametric logical heap composition*, $\circ_{\langle \mathrm{V} \rangle} : \textsc{JSLHeap}\langle \mathrm{V} \rangle \times \textsc{JSLHeap}\langle \mathrm{V} \rangle \rightharpoonup \textsc{JSLHeap}\langle \mathrm{V} \rangle$, is defined as the standard disjoint function union $\uplus$.
The *parametric unit set*, $\textsc{JSUnit}\langle \mathrm{V} \rangle \in \mathcal{P}(\textsc{JSLHeap}\langle \mathrm{V} \rangle)$, is defined as $\textsc{JSUnit}\langle \mathrm{V} \rangle \triangleq \{\mathbf{0}\}$, where $\mathbf{0}$ denotes a function with an empty domain.
The *parametric* JSLogic *PCM* is $\textsc{JSPCM}\langle \mathrm{V} \rangle \triangleq (\textsc{JSLHeap}\langle \mathrm{V} \rangle, \circ_{\langle \mathrm{V} \rangle}, \textsc{JSUnit}\langle \mathrm{V} \rangle)$.
The JSLogic *PCM* is $(\textsc{JSLHeap}, \circ, \textsc{JSUnit}) \triangleq \textsc{JSPCM}\langle \textsc{JSPVal} \rangle$.

Definition 73 ($\textsc{JSLogic}_{\mathbb{DOM}}$ partial commutative monoid)**.** Given the separation algebra of DOM logical heaps $(\textsc{LHeap}_{\mathbb{DOM}}, \bullet, \mathbf{0})$ in Def. 59, the parametric JS heaps $\textsc{JSLHeap}\langle . \rangle$ (Def. 72), the JS program values JSPVal (Def. 67) and the DOM program values $\textsc{PVal}_{\mathbb{DOM}}$ (Def. 57), the set of $\textsc{JSLogic}_{\mathbb{DOM}}$ *logical states* is:

$$\textsc{JSLHeap}_{\mathbb{DOM}} \triangleq \textsc{JSLHeap}\langle \textsc{JSPVal} \cup \textsc{PVal}_{\mathbb{DOM}} \rangle \times \textsc{LHeap}_{\mathbb{DOM}}$$

Given the parametric JS heap composition $\circ_{\langle . \rangle}$ (Def. 72), the *state composition for* $\textsc{JSLogic}_{\mathbb{DOM}}$ is defined component-wise as $+ \triangleq (\circ_{\langle \textsc{PVal}_{\mathbb{DOM}} \rangle}, \bullet)$ and is not defined when the composition on either component is undefined.

Given the parametric JS heap unit set $\textsc{JSUnit}\langle . \rangle$ (Def. 72), the *unit set for* $\textsc{JSLogic}_{\mathbb{DOM}}$ is $\textsc{JSUnit}_{\mathbb{DOM}} \triangleq \{(h, \mathbf{0}) \mid h \in \textsc{JSUnit}\langle \textsc{PVal}_{\mathbb{DOM}} \rangle\}$.

The $\textsc{JSLogic}_{\mathbb{DOM}}$ *partial commutative monoid of* is $(\textsc{JSLHeap}_{\mathbb{DOM}}, +, \textsc{JSUnit}_{\mathbb{DOM}})$.

JavaScript variable store Unlike most languages, JavaScript does not have a standard notion of variable store. Instead, JavaScript variables are stored in the heap, in a structure that emulates a variable store. This structure comprises a list of scope objects, referred to as the *scope chain*, analogous to stack frames in other languages. The standard list notation is used to describe scope chains (e.g. $[\,]$, $l{:}L$, $L_1{+\!\!+}L_2$). Every object in the scope chain has a pointer to a list of *prototypes*, referred to as the *prototype chain*, providing prototype-based inheritance. Prototype chains either end with the designated object l_{op} or are empty. The l_{op} denotes the designated JavaScript *object prototype*, describing the universal object akin to the Java universal superclass "Object". Scope objects inherit properties from their prototypes. As such, the value of a variable cannot be resolved by traversing the objects in the scope chain alone. Instead, variable resolution is carried out with respect to the scope chain *and* the prototype chains of its constituent scope objects. A variable `x` is then resolved as the property named "`x`" of the first object in the scope chain whose prototype chain defines "`x`". That is, given a scope chain $L{=}[l_1 \ldots l_n]$, if any of the objects in the prototype chain of l_1 contain a property named "`x`", then the resolution of `x` yields l_1; otherwise `x` is resolved with respect to $[l_2 \ldots l_n]$. If L is exhausted (i.e. none of the objects in the prototype chains of the objects in L contain an "`x`" property), then the resolution of `x` yields `null`. This process is described by the scope function $\sigma(h, L, \mathtt{x})$ defined in Fig. 5.5, which inspects heap h and returns the location of the first object in the scope chain L that defines variable `x`. The scope function is defined via the prototype function $\pi(h, l, \mathtt{x})$, which similarly inspects h and returns the location of the first object in the prototype chain of l that defines `x`. Lastly, the evaluation of a variable `x` does not return its value, but rather the reference $l.\mathtt{x}$ where l is obtained using the σ predicate. When the value of a variable `x` is required, the semantics implicitly calls the lookup (get-value) function γ, defined in Fig. 5.5, which returns the value denoted by the reference $l.\mathtt{x}$. In general, the return values in JavaScript may be program values $v \in \textsc{JSPVal}$ (Def. 67) or references $r \in \textsc{JSRef}$ (Def. 68). When a return value is to be inspected, the semantics calls the γ function. In case of a program value v, the γ function simply returns the value v itself. In case of a reference r, the γ returns the value denoted by r.

All programs are evaluated starting from the default scope chain $[l_g]$, where l_g denotes the location of the global JavaScript object. The final object in any scope chain is always l_g, but duplicates in a scope chain are allowed. Scope chains may change during program execution. In particular, scoping constructs such as function calls and the `with` statement

$$\sigma(h, [\,], \mathtt{x}) \triangleq \mathtt{null} \qquad \frac{\pi(h, l, \mathtt{x}) \neq \mathtt{null}}{\sigma(h, l : L, \mathtt{x}) \triangleq l} \qquad \frac{\pi(h, l, \mathtt{x}) = \mathtt{null}}{\sigma(h, l : L, \mathtt{x}) \triangleq \sigma(h, L, \mathtt{x})}$$

$$\pi(h, \mathtt{null}, \mathtt{x}) \triangleq \mathtt{null} \qquad \frac{(l, \mathtt{x}) \in dom(h) \quad h(l, x) \neq \varnothing}{\pi(h, l, \mathtt{x}) \triangleq l} \qquad \frac{h(l, x) = \varnothing \quad h(l, @proto) = l'}{\pi(h, l, \mathtt{x}) \triangleq \pi(h, l', \mathtt{x})}$$

$$\frac{v \in \text{JSPV}{\small\text{AL}}}{\gamma(h, v) \triangleq v} \qquad \frac{l \neq \mathtt{null} \quad \pi(h, l, \mathtt{x}) = l'}{\gamma(h, l.\mathtt{x}) \triangleq h(l', \mathtt{x})} \qquad \frac{l \neq \mathtt{null} \quad \pi(h, l, \mathtt{x}) = \mathtt{null}}{\gamma(h, l.\mathtt{x}) \triangleq \mathtt{undefined}}$$

Figure 5.5.: JavaScript scope chain, prototype chain and lookup functions

cause sub-expressions to be evaluated with respect to a local scope object, by putting the relevant local scope object at the beginning of the scope chain and then removing it after the sub-expressions have been evaluated. For convenience, JSLogic designates a global logical expression **l** denoting the current scope chain.

JSLogic logical expressions The logical expressions of JSLogic comprise logical values and variables, the scope chain expression **l**, arithmetic and boolean expressions, string concatenation, list and set expressions, reference constructions and lambda values. JSLogic expressions are evaluated with respect to an *evaluation environment* comprising a logical environment and a scope chain. The latter component is necessary to ensure the correct evaluation of the scope chain expression **l**.

JSLogic Instance (Parameter 16)

Definition 74 (JSLogic logical expressions)**.** The set of JSLogic *logical expressions*, $E \in$ JSLExp, is defined by the following grammar, where $v \in$ JSLVal (Def. 71), $\mathrm{x} \in$ LVar (Def. 2), $\ominus \in \{+, -, *, /, \&\&, ||, ==\}$ and $\oplus \in \{\cup, \uplus, \cap, \setminus\}$:

$E ::= v$	Logical value
$\mid \mathrm{x}$	Logical variable
$\mid \mathbf{l}$	Scope chain

$\mid E_1 \ominus E_2$	Arithmetic and boolean expressions
$\mid E_1 \bullet E_2$	String concatenation
$\mid E_1 : E_2$	List cons
$\mid E_1 \oplus E_2$	Set expressions
$\mid E_1 . E_2$	Reference construction
$\mid \lambda E_1 \cdots E_n . E$	Lambda values

Given the JS locations $\mathcal{L}$ (Def. 68), the set of JSLogic *scope chains* is $L \in \textsc{Scope} \triangleq \textsc{List}\langle \mathcal{L} \uplus \{\texttt{null}\}\rangle$.
Given the JSLogic logical values JSLVal (Def. 71) and the logical environments $\text{LEnv}\,\langle\text{JSLVal}\rangle$ (Def. 3), the set of JSLogic *evaluation environments* is $\epsilon \in \textsc{Env} \triangleq \text{LEnv}\,\langle\text{JSLVal}\rangle \times \textsc{Scope}$.
The JSLogic *expression evaluation function*, $(\!|.|\!)^{(\cdot)} : (\text{JSLExp} \times \textsc{Env}) \rightharpoonup \text{JSLVal}$, is defined as follows, for all $\epsilon, (\Gamma, L) \in \textsc{Env}$:

$$(\!|v|\!)^{\epsilon} \triangleq v \qquad (\!|\mathrm{x}|\!)^{(\Gamma,L)} \triangleq \begin{cases} v & \text{if } \Gamma(\mathrm{x}){=}v \\ \text{undefined} & \text{otherwise} \end{cases} \qquad (\!|\mathbf{l}|\!)^{(\Gamma,L)} \triangleq L$$

$$(\!|E_1 \ominus E_2|\!)^{\epsilon} \triangleq \begin{cases} v_1 \ominus v_2 & \text{if } (\!|E_1|\!)^{\Gamma}{=}v_1 \text{ and } (\!|E_2|\!)^{\Gamma}{=}v_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|E_1 \bullet E_2|\!)^{\epsilon} \triangleq \begin{cases} s_1 \bullet s_2 & \text{if } (\!|E_1|\!)^{\epsilon}{=}s_1 \text{ and } (\!|E_2|\!)^{\epsilon}{=}ss_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|E_1 : E_2|\!)^{\epsilon} \triangleq \begin{cases} v : L & \text{if } (\!|E_1|\!)^{\epsilon}{=}v \text{ and } (\!|E_2|\!)^{\epsilon}{=}L \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|E_1 \oplus E_2|\!)^{\epsilon} \triangleq \begin{cases} S_1 \oplus S_2 & \text{if } (\!|E_1|\!)^{\epsilon}{=}S_1 \text{ and } (\!|E_2|\!)^{\epsilon}{=}S_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|E_1 . E_2|\!)^{\epsilon} \triangleq \begin{cases} l.\texttt{x} & \text{if } (\!|E_1|\!)^{\epsilon}{=}l \text{ and } (\!|E_2|\!)^{\epsilon}{=}\texttt{x} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\!|\lambda E_1 \cdots E_n . E|\!)^{\epsilon} \triangleq \begin{cases} \lambda\texttt{x.e} & \text{if } (\!|E_1|\!)^{\epsilon}{=}\mathrm{x}_1 \cdots (\!|E_n|\!)^{\epsilon}{=}\mathrm{x}_n \\ & \text{and } (\!|E|\!)^{\epsilon}{=}\texttt{e} \\ \text{undefined} & \text{otherwise} \end{cases}$$

JSLogic assertions and their semantics The assertions of JSLogic comprise the standard classical and boolean assertions, JavaScript heap assertions of the form $(E_1, E_2) \mapsto E$, standard SL assertions comprising emp and the $*$ and $-\!\!*$ connectives, and the *overlapping conjunction* connective $\uplus\!\!\!*$ (also referred to as "sepish"). JSLogic assertions are interpreted over sets of JavaScript heaps, with the classical, boolean and SL assertions interpreted in the standard way. The $(E_1, E_2) \mapsto E$ assertion describes a JavaScript heap cell. When E describes a concrete value (i.e not $\varnothing$), this assertion states that the object at location E_1 has a field named E_2 and holds value E. Analogously, when E evaluates to $\varnothing$, this assertion states that the object at location E_1 has no field named E_2. The $P \uplus\!\!\!* Q$ assertion describes a JavaScript heap comprising two (potentially) overlapping sub-heaps described by P and Q, respectively. That is, $P \uplus\!\!\!* Q$ describes a heap that can be split into three sub-heaps, such that the composition of the first two sub-heaps satisfies P and the composition of the last two satisfies Q. Note that $P \land Q \Rightarrow P \uplus\!\!\!* Q$ and $P * Q \Rightarrow P \uplus\!\!\!* Q$, but neither of the reverse implications hold.

JSLogic Instance (Parameter 17)

Definition 75 (JSLogic assertions and their semantics)**.** The set of JSLogic *assertions*, $P, Q \in \text{JSAst}$, is defined by the following grammar where $\mathrm{x} \in \text{LVar}$ (Def. 2), $E_1, E_2 \in \text{JSLExp}$ (Def. 74) and $\ominus \in \{=, <, \in, \subset\}$:

$$\begin{aligned} P, Q ::= \;& \mathsf{false} \mid P \Rightarrow Q \mid \exists \mathrm{x}.\ P \mid E_1 \ominus E_2 \\ & \mid \mathsf{emp} \mid (E_1, E_2) \mapsto E \mid P * Q \mid P -\!\!* Q \mid P \uplus\!\!\!* Q \end{aligned}$$

The *ordering relation on* JSLHeap is $\mathsf{id} \subseteq \text{JSLHeap} \times \text{JSLHeap}$, where id denotes the identity relation.
The JSLogic *satisfaction relation,* $\models_{\text{JS}}$: $(\text{Env} \times \text{JSLHeap}) \times \text{JSAst}$, is defined as follows, for all $\epsilon, (\Gamma, L) \in \text{Env}$ (Def. 74) and $h \in \text{JSLHeap}$ (Def. 72):

$$\begin{array}{lll} (\epsilon, h) \models_{\text{JS}} \mathsf{false} & & \text{never} \\ (\epsilon, h) \models_{\text{JS}} (P \Rightarrow Q) & \text{iff} & (\epsilon, h) \models_{\text{JS}} P \;\Rightarrow\; (\epsilon, h) \models_{\text{JS}} Q \end{array}$$

$$
\begin{array}{lll}
((\Gamma, L), h) \models_{\mathrm{JS}} (\exists \mathrm{X}.\ P) & \text{iff} & \exists v.\ ((\Gamma[\mathrm{X} \mapsto v], L), h) \models_{\mathrm{JS}} P \\
(\epsilon, h) \models_{\mathrm{JS}} E_1 \ominus E_2 & \text{iff} & (\!|E_1|\!)^{\epsilon} \ominus (\!|E_2|\!)^{\epsilon} \\
(\epsilon, h) \models_{\mathrm{JS}} \mathsf{emp} & \text{iff} & h \in \mathrm{JSU{\scriptsize NIT}} \\
(\epsilon, h) \models_{\mathrm{JS}} (E_1, E_2) \mapsto E & \text{iff} & \exists l, \mathtt{x}, v.\ \mathit{dom}(h){=}(l, \mathtt{x}) \wedge h(l, \mathtt{x}){=}v \\
& & \quad \wedge\ (\!|E_1|\!)^{\epsilon}{=}l \wedge (\!|E_2|\!)^{\epsilon}{=}\mathtt{x} \wedge (\!|E|\!)^{\epsilon}{=}v \\
(\epsilon, h) \models_{\mathrm{JS}} (P * Q) & \text{iff} & \exists h_1, h_2.\ h{=}h_1 \circ h_2 \\
& & \quad \wedge (\epsilon, h_1) \models_{\mathrm{JS}} P \wedge (\epsilon, h_2) \models_{\mathrm{JS}} Q \\
(\epsilon, h) \models_{\mathrm{JS}} (P \mathrel{-\!\!*} Q) & \text{iff} & \forall h'.\ (\epsilon, h') \models_{\mathrm{JS}} P \Rightarrow (\epsilon, h \circ h') \models_{\mathrm{JS}} Q \\
(\epsilon, h) \models_{\mathrm{JS}} (P \uplus\!\!\!*\, Q) & \text{iff} & \exists h_1, h_2, h_3.\ h{=}h_1 \circ h_2 \circ h_3 \\
& & \quad \wedge (\epsilon, h_1 \circ h_2) \models_{\mathrm{JS}} P \\
& & \quad \wedge (\epsilon, h_2 \circ h_3) \models_{\mathrm{JS}} Q
\end{array}
$$

As before, we write $E_1 \dot{\ominus} E_2$ for $E_1 \ominus E_2 \wedge \mathsf{emp}$.

To streamline client reasoning and handle the complexity of variable resolution in JavaScript, the authors in [21] define an abstract predicate store that describes the values associated with program variables. The $\mathsf{store}_{\mathrm{L}}(\mathtt{y}_1 \ldots \mathtt{y}_m \mid \mathtt{x}_1 : v_1 \ldots \mathtt{x}_n : v_n)$ predicate describes a heap emulating a variable store given by the scope chain described by L, in which the variables $\mathtt{y}_1 \ldots \mathtt{y}_m$ are not present (have value $\varnothing$), and variables $\mathtt{x}_1 \ldots \mathtt{x}_n$ have values $v_1 \ldots v_n$, respectively. The variables $\mathtt{y}_1 \ldots \mathtt{y}_m$ can be re-ordered, as can the variables $\mathtt{x}_1 \ldots \mathtt{x}_n$. The definition of the store predicate uses the σ, π and γ functions (in Fig. 5.5) to assert the absence of $\mathtt{y}_1 \ldots \mathtt{y}_m$ as well as the presence and values of $\mathtt{x}_1 \ldots \mathtt{x}_n$. The exact structure of the emulated store and the locations of variables are hidden as they are of no concern at this level of abstraction. The definition of store includes the resources needed for traversing the scope chain L and their respective prototype chains. In particular, since the prototype chains of scope objects may arbitrarily overlap, the definition of the store predicate appeals to the overlapping conjunction connective $\uplus\!\!\!*$. We omit the definition of store here as we carry out all our client reasoning abstractly, by appealing to the store predicate. The full definition of store is given in [21].

The $\mathsf{vars}(\overline{\mathtt{x}_i : \mathrm{V}_i}^{i=1\ldots n})$ assertion is a derived assertion in JSL{\scriptsize OGIC} defined as store in the current scope chain $\mathbf{l}$:

$$\mathsf{vars}(\mathrm{S}) \triangleq \mathsf{store}_{\mathbf{l}}(|\,\mathrm{S})$$

Definition 76 (JS$_{\mathbb{DOM}}$ assertions)**.** Given the JSLogic logical values JSLVal (Def. 71) and the DOM logical values LVal$_{\mathbb{DOM}}$ (Def. 58), the set of *logical values for* JSLogic$_{\mathbb{DOM}}$, denoted JSLVal$_{\mathbb{DOM}}$, is constructed as described in Def. 30, defined as $\text{JSLVal}_{\mathbb{DOM}} \triangleq \text{JSLVal} \cup \text{LVal}_{\mathbb{DOM}}$.

Given the JSLogic assertions JSAst (Def. 75) and the DOM heap assertions Ast$_{\mathbb{DOM}}$ (Def. 66), the set of JSLogic$_{\mathbb{DOM}}$ *assertions*, JSAst$_{\mathbb{DOM}}$, is constructed from JSAst and Ast$_{\mathbb{DOM}}$ as described in Def. 36.

SSL $\mathbb{T}$ Instance (Parameter 18)

Definition 77 (DOM axioms)**.** The *axioms of DOM operation*, Axiom$_{\mathbb{DOM}}$, are given in §A.

Machine states and reification Recall that a JS logical heap (Def. 72) is a JS program heap (Def. 68) that may additionally include the special $\varnothing$ value in its range. In other words, a JS program heap is a JS logical heap without the $\varnothing$ value in its range. As such, the reification of a JS logical heap h is a program heap h' such that for all references r in the domain of h, the values of $h(r)$ and $h'(r)$ agree if and only if $h(r){\neq}\varnothing$; otherwise, $h'(r)$ is undefined.

JSLogic Instance (Parameter 19)

Definition 78 (JSLogic reification)**.** The JSLogic *reification function*, $\lfloor . \rfloor_{\text{JS}} : \text{JSLHeap} \to \mathcal{P}\left(\text{JSPHeap}\right)$, is defined as follows:

$$\lfloor h \rfloor_{\text{JS}} \triangleq \{\overline{h}\}$$

where for all $(l, \mathtt{x}) \in \text{JSRef}$:

$$\overline{h}(l, \mathtt{x}) \triangleq \begin{cases} h(l, \mathtt{x}) & \text{if} \quad (l, \mathtt{x}) \in \mathit{dom}(h) \land h(l, \mathtt{x}) \neq \varnothing \\ \text{undefined} & \text{otherwise} \end{cases}$$

Proof rules and soundness The full set of JSLogic triples are given in [21, 22] and include the standard rules of frame, existential elimination, disjunction, consequence and so forth. In JSLogic$_{\mathbb{DOM}}$ we lift the JSLogic triples to a set of parametric triples and extend them with the DOM axioms (Def. 77). Lifting the JSLogic triples to parametric ones is straightforward.

In Fig. 5.6 we present a select number of JSLogic$_{\mathbb{DOM}}$ triples including those used in our client reasoning in the upcoming section.

Note that the postcondition for each of the assignment rules ($\mathtt{x}=\mathtt{v}$, $\mathtt{x}=\mathtt{y}$, $\mathtt{x}=\mathtt{y} \oplus \mathtt{z}$, $\mathtt{x}=\mathtt{y.f}$ and $\mathtt{x.f}=\mathtt{v}$) includes true. This is because overriding a variable may render a portion of the emulated variable store superfluous. As it is not sound to simply forget parts of the heap, this redundant portion is hidden in the weak assertion true.

The postcondition of each rule contains an assertion of the form $\mathbf{r} \doteq \cdots$, recording the return value of the operation in the designated variable **r**. For instance, in the cases of assignment rules, the return value corresponds to the value of the right-hand-side of the assignment. In the case of the `while` statement, the return value is `undefined`. In the remaining rules, bar that of DOM axioms, the return value is hidden in the postcondition Q (or $Q_1 \lor Q_2$ in the disjunction rule). In the case of DOM axioms, the return value depends on the DOM operation `C` being considered and is extracted from the postcondition Q via the ret function, defined shortly. The DOM operations fall into one of two categories: i) those with an explicit return value which are of the form $\mathtt{r} := \cdots$ (e.g. `r := n.getAttribbute(s)`); and ii) those without a return value which do not contain '`r :=`' on the left-hand-side (e.g. `n.setAttribute(s, v)`). Intuitively, for all DOM operations of the form $\mathtt{r} := \cdots$, the return value corresponds to the value of `r` in the variable store. Conversely, for those DOM operations without a return value, the return value is `undefined`. The definition of the ret

$$
\begin{array}{rcl}
\{\mathsf{vars}(\mathtt{x}:\mathrm{X},\overline{\mathtt{x}_i:\mathrm{X}_i})\} & \mathtt{x}=\mathtt{v} & \{\mathsf{vars}(\mathtt{x}:\mathrm{V},\overline{\mathtt{x}_i:\mathrm{X}_i}) * \mathbf{r}\doteq\mathrm{V} * \mathsf{true}\} \\
\{\mathsf{vars}(\mathtt{x}:\mathrm{X},\mathtt{y}:\mathrm{Y},\overline{\mathtt{x}_i:\mathrm{X}_i})\} & \mathtt{x}=\mathtt{y} & \{\mathsf{vars}(\mathtt{x}:\mathrm{Y},\mathtt{y}:\mathrm{Y},\overline{\mathtt{x}_i:\mathrm{X}_i}) * \mathbf{r}\doteq\mathrm{Y} * \mathsf{true}\} \\
\left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}:\mathrm{X},\mathtt{y}:\mathrm{Y},\overline{\mathtt{x}_i:\mathrm{X}_i}) \\ * (\mathrm{Y},\mathtt{f}) \mapsto \mathrm{V}\end{array}\right\} & \mathtt{x}=\mathtt{y.f} & \left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}:\mathrm{V},\mathtt{y}:\mathrm{Y},\overline{\mathtt{x}_i:\mathrm{X}_i}) \\ * (\mathrm{Y},\mathtt{f}) \mapsto \mathrm{V} * \mathbf{r}\doteq\mathrm{V} * \mathsf{true}\end{array}\right\} \\
\{\mathsf{vars}(\mathtt{x}:\mathrm{X},\mathtt{y}:\mathrm{Y},\mathtt{z}:\mathrm{Z},\overline{\mathtt{x}_i:\mathrm{X}_i})\} & \mathtt{x}=\mathtt{y}\oplus\mathtt{z} & \left\{\begin{array}{l}\exists \mathrm{R}.\,\mathsf{vars}(\mathtt{x}:\mathrm{R},\mathtt{y}:\mathrm{Y},\mathtt{z}:\mathrm{Z},\overline{\mathtt{x}_i:\mathrm{X}_i}) \\ * \mathrm{R}\doteq\mathrm{Y}\oplus\mathrm{Z} * \mathbf{r}\doteq\mathrm{R} * \mathsf{true}\end{array}\right\} \\
\{\mathsf{vars}(\mathtt{x}:\mathrm{X},\overline{\mathtt{x}_i:\mathrm{X}_i}) * (\mathrm{X},\mathtt{f}) \mapsto \mathrm{V}\} & \mathtt{x.f}=\mathtt{v} & \left\{\begin{array}{l}\mathsf{vars}(\mathtt{x}:\mathrm{X},\overline{\mathtt{x}_i:\mathrm{X}_i}) * (\mathrm{X},\mathtt{f}) \mapsto \mathrm{V} \\ * \mathbf{r}\doteq\mathrm{V} * \mathsf{true}\end{array}\right\}
\end{array}
$$

$$
\frac{(P,\mathtt{C},Q) \in \textsc{Axiom}_{\mathbb{DOM}}}{\{P\}\ \mathtt{C}\ \{Q * \mathbf{r}\doteq\mathsf{ret}(P,\mathtt{C},Q) * \mathsf{true}\}}
\qquad
\frac{P \vdash P' \quad \{P'\}\ \mathtt{e}\ \{Q'\} \quad Q' \vdash Q}{\{P\}\ \mathtt{e}\ \{Q\}}
$$

$$
\frac{\begin{array}{l} P \equiv \mathsf{vars}(\mathtt{x}:\mathrm{X},\overline{\mathtt{x}_i:\mathrm{X}_i}) * P' \\ \{P * \mathsf{True}(\mathrm{X})\}\ \mathtt{e}_1\ \{Q\} \\ \{P * \mathsf{False}(\mathrm{X})\}\ \mathtt{e}_2\ \{Q\} \end{array}}{\{P\}\ \mathtt{if\ (x)\ then}\,\{\mathtt{e}_1\}\,\mathtt{else}\,\{\mathtt{e}_2\}\ \{Q\}}
\qquad
\frac{\{P_1\}\ \mathtt{e}\ \{Q_1\} \quad \{P_2\}\ \mathtt{e}\ \{Q_2\}}{\{P_1 \vee P_2\}\ \mathtt{e}\ \{Q_1 \vee Q_2\}}
$$

$$
\frac{\begin{array}{l} P \equiv \mathsf{vars}(\mathtt{x}:\mathrm{X},\overline{\mathtt{x}_i:\mathrm{X}_i}) * P' \\ \{P * \mathsf{True}(\mathrm{X})\}\ \mathtt{e}\ \{P\} \qquad \mathbf{r}\,\dot{\notin}\,\mathit{fv}(P) \end{array}}{\{P\}\ \mathtt{while(x)\{e\}}\ \{P * \mathsf{False}(\mathrm{X}) * \mathbf{r}\doteq\mathtt{undefined}\}}
\qquad
\frac{\{P\}\ \mathtt{e}\ \{Q\}}{\{\exists \mathrm{X}.\,P\}\ \mathtt{e}\ \{\exists \mathrm{X}.\,Q\}}
$$

$$
\frac{\{P\}\ \mathtt{e}_1\ \{R\} \quad \{R\}\ \mathtt{e}_2\ \{Q\}}{\{P\}\ \mathtt{e}_1;\mathtt{e}_2\ \{Q\}}
\qquad
\frac{\{P\}\ \mathtt{e}\ \{Q\} \qquad \mathbf{r}\,\dot{\notin}\,\mathit{fv}(R)}{\{P * R\}\ \mathtt{e}\ \{Q * R\}}
$$

where

$$
\mathsf{False}(\mathrm{V}) \triangleq \mathrm{V}\,\dot{\in}\{0,\mathtt{null},\mathtt{undefined},\text{“”}\} \qquad \mathsf{True}(\mathrm{V}) \triangleq \neg\mathsf{False}(\mathrm{V})
$$

Figure 5.6.: Selected JSLogic$_{\mathbb{DOM}}$ proof rules

function is given by induction over the structure of DOM axioms, with each case falling into one of the two categories described above. The full definition of ret is rather verbose, albeit straightforward, due to the large number of DOM operations and their axioms. As such, rather than giving the full definition of ret, we define it semi-formally as follows:

$$\mathsf{ret}(P, \mathtt{C}, Q) \triangleq \begin{cases} \mathrm{R} & \mathtt{C} = \mathtt{r} := \cdots \quad \text{and} \quad Q = \mathsf{vars}(\mathtt{r} : \mathrm{R}, \cdots) * \cdots \\ \mathtt{undefined} & \text{otherwise} \end{cases}$$

Note that the proof rules presented in Fig. 5.6 are a specialised subset of those in [21, 22]. For instance, the assignment rule given in [21, 22] is for the general assignment of the form e1 = e2. In order to tackle the intricacies of JavaScript assignment and its interaction with the emulated variable store, the premise of this generalised rule appeals to the lookup (get-value) function γ given in Fig. 5.5. That is, in this general case, one cannot reason about assignment using the high-level abstract predicate vars, and must instead turn into low-level JavaScript heap reasoning. Analogously, the while rule in Fig. 5.6 is simplified from that of [21, 22] given below:

$$\frac{\begin{array}{ll} \{P\}\ \mathtt{e1}\ \{R * \mathbf{r} \doteq \mathrm{V}_1\} & R \equiv S * \gamma(-, \mathrm{V}_1, \mathrm{V}_2) \\ \{R * \mathsf{True}(\mathrm{V}_2)\}\ \mathtt{e2}\ \{P\} & \mathbf{r} \mathrel{\dot{\notin}} \mathit{fv}(R) \end{array}}{\{P\}\ \mathtt{while(e1)\{e2\}}\ \{R * \mathsf{False}(\mathrm{V}_2) * \mathbf{r} \doteq \mathtt{undefined}\}}$$

The $\gamma(-, \mathrm{V}_1, \mathrm{V}_2)$ predicate in the premise is the logical counterpart of the get-value function γ in Fig. 5.5 and asserts that the return value of e1 (namely V_1) amounts to V_2 when inspected (informally, in the underlying heap h we have $\gamma(h, \mathrm{V}_1) = \mathrm{V}_2$). Once again, we must look into the low-level JavaScript heap and the emulated variable store therein.

As such, we opt for the simpler specialised rules of Fig. 5.6, in lieu of the general proof rules in [21, 22]. These simplified rules are sufficient for reasoning about our client programs in the proceeding section, and the vars predicate allows us to escape the complexities of the emulated variable store. Later in §7, we present a JavaScript implementation of our DOM fragment and establish its correctness with respect to the axiomatic DOM specification presented here. To do this, we appeal to the general JSLogic proof rules of [21, 22].

The JSLogic triples have a partial fault-avoiding interpretation. The soundness of JSLogic triples are established with respect to the big-step operational semantics of JSLogic, the JSLogic reification function (Def. 78) and the JSLogic satisfaction relation (Def. 75). The full proof of JSLogic soundness is given in [22]. This proof is by induction over the structure of JSLogic triples. As we demonstrated in §4 for WLogic, it is straightforward to lift the soundness proof of JSLogic triples and establish the soundness of JSLogic$_{\mathbb{DOM}}$ triples. In particular, the soundness of primitive JSLogic$_{\mathbb{DOM}}$ triples follows from Lemma 2. The soundness of inductive triples in each case follows from the (lifted) inductive hypotheses.

5.4. Reasoning about $\mathbb{DOM}$ Client Programs

We demonstrate how to use our DOM specification and the JSLogic$_{\mathbb{DOM}}$ program logic to reason about JavaScript client programs that call the DOM. We first study a JavaScript *image sanitiser* subprogram in §5.4.1 that sanitises an attribute node when its value is black-listed. Our image sanitiser program is rather simple and we present it as a didactic example in order to demonstrate our DOM client reasoning techniques. Later in §5.4.2, we use this image sanitiser code to implement an *ad blocker* to filter untrusted contents of a web page by sanitising them. Finally, in §5.4.3 we study an additional ad blocker that filters untrusted contents of a web page by removing them.

5.4.1. The santiseImg Client Program

We study a JavaScript *image sanitiser* that sanitises the "src" attribute of an element node (if it exists) by replacing its value with a trusted URL when the value is black-listed. To determine whether a value is black-listed, a remote database is queried. The results of successful lookups are stored in a local cache to minimise the number of queries. Later in §5.4.2 we use this sanitiser to implement an ad blocker that filters untrusted contents of a web page. The code of this sanitiser, `sanitiseImg`, is given in Fig. 5.7. It inspects the `img` element node for its "src" attribute (line 1). When such an attribute exists (line 2), it consults the local cache (`cache`) to check whether or not its value (`url`) is black-listed (line 3). If so, it

```
sanitiseImg ≜
     1. url := img.getAttribute("src");
     2. if (url){ // img has an attribute named "src"
     3.   isB = cache.url;
     4.   if (isB){  // url is in cache (and thus black-listed)
     5.       img.setAttribute("src",cat)
     6.   } else { // url is not in cache
     7.     isB := isBlackListed(url);
     8.     if (isB){ // url is black-listed
     9.       img.setAttribute("src", cat);
    10.       cache.url = 1
    11. } } }
```

$$\{\mathsf{st} * P_{\mathsf{out}}\} \quad \texttt{sanitiseImg} \quad \{\mathsf{st}' * P_{\mathsf{out}}\} \tag{5.10}$$

$$\{\mathsf{st} * P * (\mathrm{C}, \mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\} \quad \texttt{sanitiseImg} \quad \{\mathsf{st}' * Q * (\mathrm{C}, \mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\} \tag{5.11}$$

$$\{\mathsf{st} * P * (\mathrm{C}, \mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\} \quad \texttt{sanitiseImg} \quad \{\mathsf{st}' * Q * (\mathrm{C}, \mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\} \tag{5.12}$$

$$\{\mathsf{st} * P * (\mathrm{C}, \mathrm{S}_1) \mapsto 0 * \neg\mathsf{isB}(\mathrm{S}_1)\} \quad \texttt{sanitiseImg} \quad \{\mathsf{st}' * P * (\mathrm{C}, \mathrm{S}_1) \mapsto 0 * \neg\mathsf{isB}(\mathrm{S}_1)\} \tag{5.13}$$

where

$$\begin{aligned}
\mathsf{st} &\triangleq \mathsf{vars}(\texttt{img}{:}\mathrm{N}, \texttt{cat}{:}\mathrm{S}_2, \texttt{cache}{:}\mathrm{C}, \texttt{url}{:}-, \texttt{isB}{:}-) \qquad \mathsf{st}' \triangleq \mathsf{st} * \mathsf{true} \\
P_{\mathsf{out}} &\triangleq \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \gamma]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{out}_{\mathsf{n}}(\mathrm{A}, \text{“src”}) * \delta \mapsto \varnothing_g \\
P &\triangleq \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{src}_{\mathrm{M}}[\mathrm{T}]_{\mathrm{F}'}, \gamma]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{val}(\mathrm{T}, \mathrm{S}_1) * \delta \mapsto \varnothing_g \\
Q &\triangleq \exists \mathrm{R}, \mathrm{F}''.\ \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{src}_{\mathrm{M}}[\#\mathrm{text}_{\mathrm{R}}[\mathrm{S}_2]_{\mathrm{F}''}]_{\mathrm{F}'}, \gamma]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \mathrm{T}
\end{aligned}$$

and

$$\left\{\begin{matrix}\mathsf{vars}(\texttt{url}{:}\mathrm{S}_1, \texttt{isB}{:}-) \\ * \,\mathsf{isB}(\mathrm{S}_1)\end{matrix}\right\} \quad \texttt{isB := isBlackListed(url)} \quad \left\{\begin{matrix}\mathsf{vars}(\texttt{url}{:}\mathrm{S}_1, \texttt{isB}{:}\mathsf{true}) \\ * \,\mathsf{isB}(\mathrm{S}_1)\end{matrix}\right\}$$

$$\left\{\begin{matrix}\mathsf{vars}(\texttt{url}{:}\mathrm{S}_1, \texttt{isB}{:}-) \\ * \,\neg\mathsf{isB}(\mathrm{S}_1)\end{matrix}\right\} \quad \texttt{isB := isBlackListed(url)} \quad \left\{\begin{matrix}\mathsf{vars}(\texttt{url}{:}\mathrm{S}_1, \texttt{isB}{:}\mathsf{false}) \\ * \,\neg\mathsf{isB}(\mathrm{S}_1)\end{matrix}\right\}$$

Figure 5.7.: The `sanitiseImg` program (above) and its specification (below)

changes its value to the trusted `cat` value. If the local cache lookup is unsuccessful (line 6), the database is queried by the `isBlackListed` library call (line 7). If the value is deemed black-listed (line 8), the value of “src” is set to the trusted `cat` value (line 9), and the local cache is updated to store the lookup result (line 10).

The `sanitiseImg` program uses a combination of DOM operations and

JavaScript code to update the DOM web contents (e.g. line 9) and modify local data in the JavaScript heap (e.g. line 10).

The behaviour of `sanitiseImg` is specified in Fig. 5.7. The specifications in (5.10)-(5.13) capture different cases of the code as follows: in (5.10) `img` has no “src” attribute; in (5.11) the value of “src” is black-listed in the local cache; in (5.12) the value is black-listed and the cache has no record of it; and in (5.13) the value is not black-listed and the cache has no record of it. We focus on (5.12) here as it is the most interesting case and the remaining ones are analogous.

The precondition of (5.12) consists of four assertions: the variable store assertion `st` describing the values associated with the program variables used in the program; the P assertion describes an element node with an attribute named “src” and value s_1; the $(\text{c}, \text{s}_1) \mapsto 0$ asserts that there is no record of value s_1 in the cache (c); and $\mathsf{isB}(\text{s}_1)$ states that the value s_1 is black-listed. This assertion will be used in the `isB := isBlackListed` library call of line 7 with its behaviour as specified in Fig. 5.7.

A proof sketch of (5.12) is given in Fig. 5.8. At each proof point, we have highlighted the effect of the preceding command, where applicable.

5.4.2. The `adblocker1` Client Program

We use JSLogic$_{\mathbb{DOM}}$ to reason about a JavaScript *ad blocker* script blocking images from untrusted sources in a DOM tree. The `adblocker1` program in Fig. 5.9 compiles a NodeList containing all “img” elements in the tree rooted at `n` by calling the `getElementsByTagName` operation. It then iterates over this NodeList, sanitising each image by running the `sanitiseImg` code studied in §5.4.1 (Fig. 5.7) and replacing each untrusted “src” attribute value with the trusted `cat` value.

The specification of `adblocker1` is given in Fig. 5.9.[5,6]

Observe that as per interpretation of Hoare triples, all free logical variables included in the definitions of P and Q are universally quantified outside the triple $\{P\}$ `adblocker1` $\{Q\}$. This is to ensure that the tree un-

[5]We write e.g. $\overline{\text{x}}^{\text{L}}$ to denote a list of logical variables of size $|\text{L}|$. We use a suggestive notation and write x_{J} for the J^{th} variable in x (i.e. $|\text{x}|^{\text{J}}$.)

[6] All free logical variables in P and Q are universally quantified outside the triple as they do not change throughout the execution. By contrast, the iteration number I, and the tag listeners E associated with element node N may change (the latter may grow by `getElementsByTagName`) and are explicitly parameterised when relevant.

```
$\{\mathsf{vars}(\texttt{img}{:}\textsc{n}, \texttt{cat}{:}\textsc{s}_2, \texttt{cache}{:}\textsc{c}.\texttt{url};-, \texttt{isB};-) * P * (\textsc{c}, \textsc{s}_1) \mapsto 0 * \mathsf{isB}(\textsc{s}_1)\}$
url := img.getAttribute("src");
  $\{\mathsf{vars}(\texttt{img}{:}\textsc{n}, \texttt{cat}{:}\textsc{s}_2, \texttt{cache}{:}\textsc{c}, \texttt{url}{:}\textsc{s}_1, \texttt{isB}{:}-) * P * (\textsc{c}, \textsc{s}_1) \mapsto 0 * \mathsf{isB}(\textsc{s}_1) * \mathsf{true}\}$
if (url){ // img has an attribute named “src”
  isB = cache.url;
  $\{\mathsf{vars}(\texttt{img}{:}\textsc{n}, \texttt{cat}{:}\textsc{s}_2, \texttt{cache}{:}\textsc{c}, \texttt{url}{:}\textsc{s}_1, \texttt{isB}{:}0) * P * (\textsc{c}, \textsc{s}_1) \mapsto 0 * \mathsf{isB}(\textsc{s}_1) * \mathsf{true}\}$
  if (isB){
    img.setAttribute("src",cat)
  } else {
  $\{\mathsf{vars}(\texttt{img}{:}\textsc{n}, \texttt{cat}{:}\textsc{s}_2, \texttt{cache}{:}\textsc{c}, \texttt{url}{:}\textsc{s}_1, \texttt{isB}{:}0) * P * (\textsc{c}, \textsc{s}_1) \mapsto 0 * \mathsf{isB}(\textsc{s}_1) * \mathsf{true}\}$
    isB := isBlackListed(url);
  $\{\mathsf{vars}(\texttt{img}{:}\textsc{n}, \texttt{cat}{:}\textsc{s}_2, \texttt{cache}{:}\textsc{c}, \texttt{url}{:}\textsc{s}_1, \texttt{isB}{:}1) * P * (\textsc{c}, \textsc{s}_1) \mapsto 0 * \mathsf{isB}(\textsc{s}_1) * \mathsf{true}\}$
    if (isB){
  $\{\mathsf{vars}(\texttt{img}{:}\textsc{n}, \texttt{cat}{:}\textsc{s}_2, \texttt{cache}{:}\textsc{c}, \texttt{url}{:}\textsc{s}_1, \texttt{isB}{:}1) * P * (\textsc{c}, \textsc{s}_1) \mapsto 0 * \mathsf{isB}(\textsc{s}_1) * \mathsf{true}\}$
      img.setAttribute("src", cat);
  $\{\mathsf{vars}(\texttt{img}{:}\textsc{n}, \texttt{cat}{:}\textsc{s}_2, \texttt{cache}{:}\textsc{c}, \texttt{url}{:}\textsc{s}_1, \texttt{isB}{:}1) * Q * (\textsc{c}, \textsc{s}_1) \mapsto 0 * \mathsf{isB}(\textsc{s}_1) * \mathsf{true}\}$
      cache.url = 1
  $\{\mathsf{vars}(\texttt{img}{:}\textsc{n}, \texttt{cat}{:}\textsc{s}_2, \texttt{cache}{:}\textsc{c}, \texttt{url}{:}\textsc{s}_1, \texttt{isB}{:}1) * Q * (\textsc{c}, \textsc{s}_1) \mapsto 1 * \mathsf{isB}(\textsc{s}_1) * \mathsf{true}\}$
} } }
$\{\mathsf{vars}(\texttt{img}{:}\textsc{n}, \texttt{cat}{:}\textsc{s}_2, \texttt{cache}{:}\textsc{c}, \texttt{url}{:}-, \texttt{isB}{:}-) * Q * (\textsc{c}, \textsc{s}_1) \mapsto 1 * \mathsf{isB}(\textsc{s}_1) * \mathsf{true}\}$
```

Figure 5.8.: A proof sketch of specification (5.12)

derneath N remains unchanged by `adblocker1` except for the values of its untrusted “src” attributes. More concretely, when `n`=N, `c`=C and `cat`=S, we use the following universally quantified logical variables to track the various components of the tree at N:

- F and E for the initial forest and tag listeners of N;
- L for the list of “img” elements underneath N (i.e. the contents of the NodeList resulting from `getElementsByTagName("img")`);
- $\textsc{s}_e$ for the set of empty nodes in L (those without a “src” attribute);
- $\textsc{s}_u$ for the set of untrusted nodes in L;
- $\textsc{s}_t$ for the set of trusted nodes in L;
- $\textsc{t}_0$ for the initial tree underneath N, $\overline{\textsc{t}}^{\textsc{l}}$ for the trees resulting from the subsequent iterations with $\textsc{t}_\textsc{i}$ denoting the tree at iteration I (see footnote 5);

`adblocker1` $\triangleq$

```
imgs := n.getElementsByTagName("img");
len := imgs.length();
i = 0;
while( i < len){
  img := imgs.item(i);
  sanitiseImg;
  i = i+1;
}
```

$$\{P\}\ \texttt{adblocker1}\ \{Q\}$$

where

$$
\begin{aligned}
P &\triangleq \mathsf{vars}_{\mathsf{ab1}} * \neg\mathsf{isB}(\textsc{s}) * \mathsf{cache}(\textsc{c}) * \mathsf{fld}(0, \textsc{e}) * \mathsf{rem}(0)\\
Q &\triangleq \exists \textsc{e}', \textsc{r}.\ \mathsf{vars}_{\mathsf{ab1}} * \neg\mathsf{isB}(\textsc{s}) * \mathsf{cache}(\textsc{c}) * \mathsf{fld}(|\textsc{l}|, \textsc{e}') * \mathsf{rem}(|\textsc{l}|)\\
&\qquad * \textsc{e} \dot{\subseteq} \textsc{e}' * (\text{“img”}, \textsc{r}) \dot{\in} \textsc{e}' * \mathsf{true}\\
\mathsf{vars}_{\mathsf{ab1}} &\triangleq \mathsf{vars}(\texttt{n}{:}\textsc{n}, \texttt{cache}{:}\textsc{c}, \texttt{imgs}{:}-, \texttt{len}{:}-, \texttt{i}{:}-, \texttt{img}{:}-, \texttt{cat}{:}\textsc{s}, \texttt{url}{:}-, \texttt{isB}{:}-)\\
\mathsf{cache}(\textsc{c}) &\triangleq \mathsf{CF}(\textsc{c}, \mathcal{X})\\
\mathsf{CF}(\textsc{c}, \textsc{s}) &\triangleq (\textsc{s} \doteq \emptyset) \vee \Big(\exists \textsc{f}, \textsc{s}'.\ \textsc{s} \doteq \{\textsc{f}\} \uplus \textsc{s}' * \big((\textsc{c}, \textsc{f}) \mapsto 1 \vee (\textsc{c}, \textsc{f}) \mapsto 0\big) * \mathsf{CF}(\textsc{c}, \textsc{s}')\Big)\\
\mathsf{fld}(\textsc{i}, \textsc{e}) &\triangleq \mathsf{tree}(\textsc{i}, \textsc{e}) * ((\forall \textsc{i}', \textsc{e}'.\, \mathsf{tree}(\textsc{i}', \textsc{e}') -\!\!\circledast\ \mathsf{unfld}(\textsc{i}', \textsc{e}')) \wedge \mathsf{emp})\\
\mathsf{tree}(\textsc{i}, \textsc{e}) &\triangleq \alpha \mapsto \textsc{s}_{\textsc{n}}[\textsc{a}, \textsc{t}_{\textsc{i}}]_{\textsc{f}}^{\textsc{e}} * \mathsf{srch}(\textsc{t}_{\textsc{i}}, \text{“img”}, \textsc{l})\\
\mathsf{unfld}(\textsc{i}, \textsc{e}) &\triangleq \exists \overline{\alpha, \beta, \gamma}^{\textsc{l}}.\ \mathsf{partition}(\textsc{i}) * (\forall \textsc{i}'.\ \mathsf{partition}(\textsc{i}') -\!\!\circledast \mathsf{tree}(\textsc{i}', \textsc{e}))\\
\mathsf{partition}(\textsc{i}) &\triangleq \textsc{l} \dot{\equiv} \textsc{s}_e \uplus \textsc{s}_u \uplus \textsc{s}_t * \exists \textsc{s}_s.\ \textsc{s}_s \doteq \textsc{s}_u \cap \{|\textsc{l}|^{\textsc{j}} \mid \textsc{j} < \textsc{i}\}\\
&\quad \mathop{\circledast}_{\textsc{j} \in \textsc{s}_e} \big(\alpha_{\textsc{j}} \mapsto \mathrm{img}_{\textsc{j}}[\textsc{a}_{\textsc{j}}, \gamma_{\textsc{j}}]_{\textsc{f}_{\textsc{j}}}^{\textsc{e}_{\textsc{j}}} * \mathsf{out}_{\mathsf{n}}(\textsc{a}_{\textsc{j}}, \mathrm{src})\big)\\
&\quad \mathop{\circledast}_{\textsc{j} \in \textsc{s}_u \setminus \textsc{s}_s} \big(\alpha_{\textsc{j}} \mapsto \mathrm{img}_{\textsc{j}}[\beta_{\textsc{j}} \odot \mathrm{src}_{\textsc{m}_{\textsc{j}}}[\textsc{a}_{\textsc{j}}]_{\textsc{f}'_{\textsc{j}}}, \gamma_{\textsc{j}}]_{\textsc{f}_{\textsc{j}}}^{\textsc{e}_{\textsc{j}}} * \mathsf{val}(\textsc{a}_{\textsc{j}}, \textsc{v}_{\textsc{j}}) * \mathsf{isB}(\textsc{v}_{\textsc{j}})\big)\\
&\quad \mathop{\circledast}_{\textsc{j} \in \textsc{s}_t} \big(\alpha_{\textsc{j}} \mapsto \mathrm{img}_{\textsc{j}}[\beta_{\textsc{j}} \odot \mathrm{src}_{\textsc{m}_{\textsc{j}}}[\textsc{a}_{\textsc{j}}]_{\textsc{f}'_{\textsc{j}}}, \gamma_{\textsc{j}}]_{\textsc{f}_{\textsc{j}}}^{\textsc{e}_{\textsc{j}}} * \mathsf{val}(\textsc{a}_{\textsc{j}}, \textsc{v}_{\textsc{j}}) * \neg\mathsf{isB}(\textsc{v}_{\textsc{j}})\big)\\
&\quad \mathop{\circledast}_{\textsc{j} \in \textsc{s}_s} \big(\alpha_{\textsc{j}} \mapsto \mathrm{img}_{\textsc{j}}[\beta_{\textsc{j}} \odot \mathrm{src}_{\textsc{m}_{\textsc{j}}}[\#\mathrm{text}_{-}[\textsc{s}]]_{\textsc{f}'_{\textsc{j}}}, \gamma_{\textsc{j}}]_{\textsc{f}_{\textsc{j}}}^{\textsc{e}_{\textsc{j}}}\big)\\
\mathsf{rem}(\textsc{i}) &\triangleq \exists \textsc{s}_s.\ \textsc{s}_s \doteq \textsc{s}_u \cap \{|\textsc{l}|^{\textsc{j}} \mid \textsc{j} < \textsc{i}\} * \beta \mapsto \varnothing_g \bigoplus_{\textsc{j} \in \textsc{s}_s} \textsc{a}_{\textsc{j}}
\end{aligned}
$$

Figure 5.9.: The `adBlocker1` client program and its specification

- $\overline{\textsc{f}}^{\textsc{l}}$ and $\overline{\textsc{e}}^{\textsc{l}}$ for the forest and tag listeners of nodes in $\textsc{l}$;
- $\overline{\textsc{m}}^{\textsc{l}}$ for the “src” attribute identifiers of nodes in $\textsc{l}$;
- $\overline{\textsc{f}'}^{\textsc{l}}$ for the forest listeners of nodes in $\overline{\textsc{m}}^{\textsc{l}}$;

- $\overline{\mathrm{A}}^{\mathrm{L}}$ for either the text forests of nodes $\overline{\mathrm{M}}^{\mathrm{L}}$, or for the attribute sets of nodes in L.

Lastly, observe that in the definitions of $\mathsf{partition}$ and rem predicates we use the set comprehension expression, $\mathrm{S}_s \dot{=} \mathrm{S}_u \cap \{|\mathrm{L}|^{\mathrm{J}} \mid \mathrm{J} < \mathrm{I}\}$. Although set comprehension is not included in the JSLogic$_{\mathbb{DOM}}$ assertion language, we can rewrite the above in JSLogic$_{\mathbb{DOM}}$ as follows. However, for brevity and clarity we opt for set the comprehension notation instead.

$$\mathrm{S}_s \dot{\subseteq} \mathrm{S}_u * (\forall \mathrm{J}.\ \mathrm{J} \dot{<} \mathrm{I} * |\mathrm{L}|^{\mathrm{J}} \in \mathrm{S}_u \Rightarrow |\mathrm{L}|^{\mathrm{J}} \dot{\in} \mathrm{S}_s)$$

The precondition of `adblocker1`, P, comprises four assertions. The first assertion, $\mathsf{vars}_{\mathsf{ab1}}$, describes the values associated with the program variables and further asserts that the value of `cat` (i.e. S) is trusted. The second assertion, $\mathsf{cache}(\mathrm{C})$, describes the cached results of URL inspections. Recall that `sanitiseImg` (Fig. 5.7) maintains a local cache of blacklisted URLs, implemented as an object at C with one field per URL (where $(\mathrm{C}, \mathrm{F}) \mapsto 1$ asserts that the URL F is blacklisted in the cache, and $(\mathrm{C}, \mathrm{F}) \mapsto 0$ asserts that the URL F is not blacklisted in the cache). We thus define the cache as the collection of all valid field s in $\mathcal{X}$ (Def. 68) on C, with value 1 or 0 as defined in Fig. 5.9. The third and fourth assertions, $\mathsf{fld}(0, \mathrm{E}) * \mathsf{rem}(\mathrm{I})$, describe the DOM footprint of `adblocker1` (i.e. the DOM resources needed for running `adblocker1`). Let us turn our focus to the DOM footprint of `adblocker1`.

Since `adblocker1` calls the NodeList operation `item` at each iteration I (line 5), to compile a list L of all "img" elements below node `n`, the footprint must contain enough resources to allow this DOM call. When `n`$=\mathrm{N}$, as a first attempt we can describe the DOM resources needed at iteration I as follows with the srch predicate as defined in Fig. 5.4:

$$\mathsf{tree}(\mathrm{I}, \mathrm{E}) \triangleq \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \mathrm{T}_{\mathrm{I}}]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{srch}(\mathrm{T}_{\mathrm{I}}, \mathrm{img}, \mathrm{L})$$

where E denotes the tag listener set associated with N and T_{I} denotes the child forest of node N at iteration I. However, this is not enough. Consider the following assertions where (5.14) describes the subtree rooted at element node named "ad" in Fig. 5.1a, and (5.15) describes the same subtree with the attributes of its descendants framed off (β and γ):

$$\alpha \mapsto \mathrm{ad}_9\left[\varnothing, \begin{pmatrix} \mathrm{img}_3\left[\begin{pmatrix}\mathrm{src}_{13}[\#\mathrm{text}_-[\mathrm{goo.gl/K4S0d0}]] \\ \odot\ \mathrm{width}_{17}[\#\mathrm{text}_-[800\mathrm{px}]]\end{pmatrix}, \varnothing\right] \\ \otimes\, \mathrm{img}_8[\varnothing,\varnothing]\end{pmatrix}\right] \tag{5.14}$$

$$\alpha \mapsto \mathrm{ad}_9[\varnothing, (\mathrm{img}_3[\beta,\varnothing] \otimes \mathrm{img}_8[\gamma,\varnothing])] \tag{5.15}$$

While both assertions imply $\mathsf{tree}(-,-)$, neither contain enough information. Since the program iterates over the element nodes in L and inspects their attributes, for each element in L, either we need to know the value of its "src" attribute if it exists (so that we can sanitise it if necessary), or we need to know that it has no such attribute. However, in (5.15) the attributes of "img" elements have been framed off and thus (5.15) does not contain the necessary resources. As for (5.14), we need to transform it to the following form with the "img" elements *unfolded* so that we can inspect them individually at each iteration (as per the precondition of `sanitiseImg`):

$$\alpha \mapsto \mathrm{ad}_9[\varnothing, (\beta \otimes \gamma)] * \beta \mapsto \mathrm{img}_3[-"-] * \gamma \mapsto \mathrm{img}_8[-"-] \tag{5.16}$$

More concretely, the footprint needs to unfold all "img" elements in the tree at N and *partition* them into three categories: i) $\underline{e}$*mpty*: without a "src" attribute; ii) $\underline{u}$*ntrusted*: with a "src" attribute and a blacklisted value; iii) $\underline{t}$*rusted*: with a "src" attribute and a trusted value. At each iteration I, if the node considered is untrusted, it is sanitised and removed from the untrusted category. We thus define a fourth category, $\underline{s}$*anitised*, including those "img" elements whose value were initially blacklisted and have now been sanitised. This partitioning of "img" element nodes at iteration I is described by the $\mathsf{partition}(\mathrm{I})$ predicate defined in Fig. 5.9, with out and val as defined in Fig. 5.4 and isB as described in §5.4.1.

The first part of $\mathsf{partition}(\mathrm{I})$ states that the list of "img" elements L can be partitioned into the three categories described above where $\mathrm{L} \equiv \mathrm{S}$ states that set S is a permutation of list L ($\mathrm{L} \equiv \mathrm{S}$ iff $\forall k.\ k \in \mathrm{L} \Leftrightarrow k \in \mathrm{S}$; note that L has no duplicates). The second part states that list L has been processed up to index I; i.e. the sanitised category S_s includes all the untrusted elements in L up to index I. The last four parts describe the "img" elements according to their category. Observe that in the "sanitised" category (i.e. those in S_s) the text forest of the node has been replaced by a new text node containing the trusted value S where $\neg\mathsf{isB}(\mathrm{S})$ holds (see the definition of $\mathsf{vars}_{\mathsf{ab1}}$).

The $\mathsf{partition}$ predicate describes the "img" elements in L only and does not include the *remainder* of the subtree at N. At every iteration, this remainder is untouched and the modified parts are contained in the partitions. We thus describe the remainder for an arbitrary iteration I' as $\forall \mathrm{I}'.\ \mathsf{partition}(\mathrm{I}') \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I}', \mathrm{E})$; that is, the entire tree for that iteration, $\mathsf{tree}(\mathrm{I}', \mathrm{E})$, *minus* its partitions. The *unfolded* tree (with "img" elements singled out) at iteration I thus consists of the partitions at I and the remainder. This is described by the $\mathsf{unfld}(\mathrm{I}, \mathrm{E})$ predicate in Fig. 5.9.

Observe that for Nodelist operations such as `item` (line 5), we need the *folded* tree (i.e. $\mathsf{tree}(\mathrm{I}, \mathrm{E})$) with the entire subtree containing the "img" list L, as required by their axioms (Fig. 5.2). Conversely, for the `sanitiseImg` program (line 6), we need the *unfolded* "img" elements (i.e. $\mathsf{partition}(\mathrm{I})$) so that we can access the relevant "img" node at each iteration. We thus need to move between the folded and unfolded tree depending on the operation considered.

The $\mathsf{fld}(\mathrm{I}, \mathrm{E})$ predicate defined in Fig. 5.9 describes the folded tree at iteration I. The first part, $\mathsf{tree}(\mathrm{I}, \mathrm{E})$, describes the resources of the folded tree at iteration I, as described above. The second part contains no resources (emp); it simply states that at any iteration I', the folded tree $\mathsf{tree}(\mathrm{I}', \mathrm{E}')$, can be *exchanged* for the unfolded tree $\mathsf{unfld}(\mathrm{I}', \mathrm{E}')$. That is, as shown in the derivation below, the second part provides a mechanism for shifting from folded to unfolded resources (5.17-5.19) and vice versa (5.19-5.22), for arbitrary I' and E'. We use this derivation in the proof sketch of Fig. 5.10.

The bi-implication of (5.17) follows from the definition of fld and the fact that empty resources (emp) can be freely duplicated. In (5.18) we simply eliminate the first universal quantifier. We then apply the adjunct elimination rule ($P * (P \mathrel{-\!\!*} Q) \Rightarrow Q$) to arrive at (5.19). The implication of (5.20) follows from the definition of unfld and the elimination of the first universal quantifier. In (5.21) we apply the adjunct elimination rule and eliminate the existential quantifiers. Finally in (5.22) we wrap the definition of fld.

$$
\begin{aligned}
\mathsf{fld}(\text{I}, \text{E}) \Leftrightarrow\ & \mathsf{tree}(\text{I}, \text{E}) * ((\forall \text{I}', \text{E}'.\ \mathsf{tree}(\text{I}', \text{E}') \mathrel{-\!\!*} \mathsf{unfld}(\text{I}', \text{E}')) \wedge \mathsf{emp}) \\
& \quad * \ ((\forall \text{I}', \text{E}'.\ \mathsf{tree}(\text{I}', \text{E}') \mathrel{-\!\!*} \mathsf{unfld}(\text{I}', \text{E}')) \wedge \mathsf{emp}) && (5.17) \\
\Rightarrow\ & \mathsf{tree}(\text{I}, \text{E}) * (\mathsf{tree}(\text{I}, \text{E}) \mathrel{-\!\!*} \mathsf{unfld}(\text{I}, \text{E})) \\
& \quad * \ ((\forall \text{I}', \text{E}'.\ \mathsf{tree}(\text{I}', \text{E}') \mathrel{-\!\!*} \mathsf{unfld}(\text{I}', \text{E}')) \wedge \mathsf{emp}) && (5.18) \\
\Rightarrow\ & \mathsf{unfld}(\text{I}, \text{E}) * ((\forall \text{I}', \text{E}'.\ \mathsf{tree}(\text{I}', \text{E}') \mathrel{-\!\!*} \mathsf{unfld}(\text{I}', \text{E}')) \wedge \mathsf{emp}) && (5.19) \\
\Rightarrow\ & \exists \overline{\alpha, \beta, \gamma}^{\text{L}}.\ \mathsf{partition}(\text{I}) * (\mathsf{partition}(\text{I}) \mathrel{-\!\!*} \mathsf{tree}(\text{I}, \text{E})) \\
& \quad * \ ((\forall \text{I}', \text{E}'.\ \mathsf{tree}(\text{I}', \text{E}') \mathrel{-\!\!*} \mathsf{unfld}(\text{I}', \text{E}')) \wedge \mathsf{emp}) && (5.20) \\
\Rightarrow\ & \mathsf{tree}(\text{I}, \text{E}) * ((\forall \text{I}', \text{E}'.\ \mathsf{tree}(\text{I}', \text{E}') \mathrel{-\!\!*} \mathsf{unfld}(\text{I}', \text{E}')) \wedge \mathsf{emp}) && (5.21) \\
\Leftrightarrow\ & \mathsf{fld}(\text{I}, \text{E}) && (5.22)
\end{aligned}
$$

Recall that when the value of an attribute node `n` is updated via the `n.setAttribute` operation, the text forest of `n` is replaced with a new text node containing the new value, and the old text forest of `n` is added to the grove (see Fig. 5.3). As such, at each iteration I if we sanitise the value of the I^{th} node in L, then the old text forest of the node is moved to the grove. This is described by the $\mathsf{rem}(\text{I})$ assertion in Fig. 5.9 asserting that for each sanitised attribute node in S_s, the old text forest (A_J) has been added to the grove.

We give a proof sketch of `adBlocker1` in Fig. 5.10. The precondition (defined as P in Fig. 5.9) comprises the variable store, the cache, the folded *unprocessed* (iteration 0) tree, and an empty spot in the grove reserved for the text forests of nodes to be sanitised. Similarly, the postcondition comprises the variable store, the cache, the folded, *fully processed* (iteration $|\text{L}|$) tree with its tag listeners extended with a new listener for "img", and the grove extended with the text forests of all sanitised nodes.

5.4.3. The `adblocker2` Client Program

The `adblocker2` program in Fig. 5.11 compiles a NodeList containing all "img" elements of the tree rooted at `n` by calling the `getElementsByTagName` operation. It then iterates over this NodeList, inspecting each element in order. At each iteration, if the element has no "src" attribute or if the value of its "src" attribute is not blacklisted, then the element is left unchanged. On the other hand, if the value of "src" attribute is blacklisted,

```
$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}-, \mathtt{len}{:}-, \mathtt{i}{:}-, \mathtt{img}{:}-, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * \mathsf{fld}(0, E) * \mathsf{rem}(0)\end{array}\right\}$
1. imgs := n.getElementsByTagName("img");
   $\left\{\begin{array}{l}\exists E', R.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}R, \mathtt{len}{:}-, \mathtt{i}{:}-, \mathtt{img}{:}-, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * \mathsf{fld}(0, E') * \mathsf{rem}(0) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * \mathsf{true}\end{array}\right\}$
2. len := imgs.length();
   $\left\{\begin{array}{l}\exists E', R.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}R, \mathtt{len}{:}\,|L|, \mathtt{i}{:}-, \mathtt{img}{:}-, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * \mathsf{fld}(0, E') * \mathsf{rem}(0) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * \mathsf{true}\end{array}\right\}$
3. i = 0;
   $\left\{\begin{array}{l}\exists E', R.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}R, \mathtt{len}{:}\,|L|, \mathtt{i}{:}0, \mathtt{img}{:}-, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * \mathsf{fld}(0, E') * \mathsf{rem}(0) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * \mathsf{true}\end{array}\right\}$
   $\left\{\begin{array}{l}\exists E', R, I.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}R, \mathtt{len}{:}\,|L|, \mathtt{i}{:}I, \mathtt{img}{:}-, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * \mathsf{fld}(I, E') * \mathsf{rem}(I) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * 0 \dot{\leq} I \dot{\leq} |L| * \mathsf{true}\end{array}\right\}$
4. while(i<len){
   $\left\{\begin{array}{l}\exists E', R, I.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}R, \mathtt{len}{:}\,|L|, \mathtt{i}{:}I, \mathtt{img}{:}-, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * \mathsf{fld}(I, E') * \mathsf{rem}(I) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * 0 \dot{\leq} I \dot{<} |L| * \mathsf{true}\end{array}\right\}$
5.    img := imgs.item(i);
   $\left\{\begin{array}{l}\exists E', R, I.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}R, \mathtt{len}{:}\,|L|, \mathtt{i}{:}I, \mathtt{img}{:}\,|L|^{I}, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * \mathsf{fld}(I, E') * \mathsf{rem}(I) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * 0 \dot{\leq} I \dot{<} |L| * \mathsf{true}\end{array}\right\}$
   // Apply steps in (5.17)-(5.19); abstract allocation at $\epsilon$
   $\left\{\begin{array}{l}\exists E', R, I, \epsilon.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}R, \mathtt{len}{:}\,|L|, \mathtt{i}{:}I, \mathtt{img}{:}\,|L|^{I}, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * 0 \dot{\leq} I \dot{<} |L| * \mathsf{true} \\ * \delta \mapsto \left(\epsilon \bigoplus_{J \in S_u \cap \{|L|^{J} \mid J < I\}} A_J\right) * \epsilon \mapsto \varnothing_g \\ * \mathsf{unfld}(I, E') * ((\forall I', E'.\ \mathsf{tree}(I', E') \mathrel{-\!\!*} \mathsf{unfld}(I', E')) \wedge \mathsf{emp})\end{array}\right\}$
6.    sanitiseImg
   $\left\{\begin{array}{l}\exists E', R, I, \epsilon.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}R, \mathtt{len}{:}\,|L|, \mathtt{i}{:}I, \mathtt{img}{:}\,|L|^{I}, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * 0 \dot{\leq} I \dot{<} |L| * \mathsf{true} \\ * \delta \mapsto \left(\epsilon \bigoplus_{J \in S_u \cap \{|L|^{J} \mid J < I\}} A_J\right) * \epsilon \mapsto \left(\varnothing_g \bigoplus_{J \in S_u \cap \{|L|^{I}\}} A_J\right) \\ * \mathsf{unfld}(I+1, E') * ((\forall I', E'.\ \mathsf{tree}(I', E') \mathrel{-\!\!*} \mathsf{unfld}(I', E')) \wedge \mathsf{emp})\end{array}\right\}$
   // Apply steps in (5.19)-(5.22); abstract deallocation at $\epsilon$
   $\left\{\begin{array}{l}\exists E', R, I.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}R, \mathtt{len}{:}\,|L|, \mathtt{i}{:}I, \mathtt{img}{:}\,|L|^{I}, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * \mathsf{fld}(I+1, E') * \mathsf{rem}(I+1) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * 0 \dot{\leq} I \dot{<} |L| * \mathsf{true}\end{array}\right\}$
7.    i = i+1;
   $\left\{\begin{array}{l}\exists E', R, I.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}R, \mathtt{len}{:}\,|L|, \mathtt{i}{:}I, \mathtt{img}{:}\,|L|^{I}, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * \mathsf{fld}(I, E') * \mathsf{rem}(I) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * 0 \dot{\leq} I \dot{\leq} |L| * \mathsf{true}\end{array}\right\}$
8. }
$\left\{\begin{array}{l}\exists E', R.\ \mathsf{vars}(\mathtt{n}{:}N, \mathtt{cache}{:}C, \mathtt{imgs}{:}-, \mathtt{len}{:}-, \mathtt{i}{:}-, \mathtt{img}{:}-, \mathtt{cat}{:}S, \mathtt{url}{:}-, \mathtt{isB}{:}-) \\ * \neg\mathsf{isB}(S) * \mathsf{cache}(C) * \mathsf{fld}(|L|, E') * \mathsf{rem}(|L|) * E \dot{\subseteq} E' * (\text{“img”}, R) \dot{\in} E' * \mathsf{true}\end{array}\right\}$
```

Figure 5.10.: A proof sketch of the `adblocker1` program

then the entire subtree at that element is removed from the DOM tree.

Observe that when an “img” element is removed from the tree (i.e. when its “src” attribute holds a blacklisted value), the iteration counter `i` is not increased. This is because DOM NodeLists are *live* and dynamically reflect the changes to the DOM tree. Consider executing `adblocker2` with `n`=4 denoting the identifier of the element named “body” in Fig. 5.1a. Calling `n.getElementsByTagName("img")` on line 1 then yields `imgs`=$[3, 8, 2]$. Let us suppose that the value of the “src” attribute on node 3 is blacklisted. At iteration `i`=0, node 3 is removed and the `imgs` NodeList is accordingly updated to `imgs`=$[8, 2]$. Were we to increment `i` to 1, we would proceed with inspecting element node 2 and would erroneously skip inspecting element 8. As such, unlike `adblocker1`, the progress of the loop cannot be measured by the iteration index $\textsc{i}$ alone, rather a combination of the iteration index $\textsc{i}$ and the *removal index* $\textsc{j}$, denoting the number of “img” elements removed so far (i.e. the number of times line 9 is executed). That is, initially $\textsc{i}=\textsc{j}=0$, and at each iteration either $\textsc{i}$ is incremented by 1, or $\textsc{j}$ is incremented by 1. Both $\textsc{i}$ and $\textsc{j}$ are non-negative; we omit this assumption from our predicates for better readability. At the end of the program, the $\textsc{i}$ denotes the number of (safe) “img” elements (those with either no “src” attribute or a whitelisted “src” attribute) remaining in the tree. Analogously, the $\textsc{j}$ denotes the number of (unsafe) “img” elements removed from the tree (those with a blacklisted “src” attribute).

Since the `imgs` NodeList evolves each time an “img” element is removed, we enumerate the lists describing the value of `imgs` after each removal. That is, when there are n elements to be removed, we define $\textsc{l}=[\textsc{l}_0, \textsc{l}_1, \ldots, \textsc{l}_n]$ where $\textsc{l}_0$ denotes the initial value of `imgs`, and $\textsc{l}_k$ denotes the value of `imgs` after k removals. In other words, since $\textsc{j}$ denotes the removal index, the $|\textsc{l}|^{\textsc{j}}$ describes the value of `imgs` after $\textsc{j}$ elements are removed. As before, for any given index $\textsc{m}$, we use a suggestive notation and write $\textsc{l}_{\textsc{m}}$ as a shorthand for $|\textsc{l}|^{\textsc{m}}$ (see footnote 5).

Note that $\textsc{r}=|\textsc{l}|-1$ denotes the total number of unsafe “img” elements (those to be removed). For brevity we write $\textsc{r}$ for $|\textsc{l}|-1$ in our predicates. Similarly, the last list in $\textsc{l}$, namely $\textsc{l}_{\textsc{r}}$, comprises only safe “img” elements under the tree (otherwise $\textsc{l}_{\textsc{r}}$ must evolve to yet another list $\textsc{l}_{\textsc{r}+1}$ excluding the unsafe elements). Recall that at the end of the program the $\textsc{i}$ denotes the number of (safe) “img” elements remaining in the tree and $\textsc{j}$ denotes the number of (unsafe) “img” elements removed. Consequently, at the end of the program $\textsc{i}{=}|\textsc{l}_\textsc{r}|$ and $\textsc{j}{=}\textsc{r}$.

The specification of `adblocker2` is given in Fig. 5.11 and is similar to that of `adblocker1`. The precondition P comprises three assertions. The $\mathsf{vars}_{\mathsf{ab2}}$ assertion describes the values associated with the program variables. The fld assertion describes the DOM footprint of the operation. The $\zeta \mapsto \varnothing_g$ reserves an empty spot in the grove where the removed elements (those with blacklisted “src” attributes) may be moved.

Similar to the fld assertion in `adblocker1`, the $\mathsf{fld}(\textsc{i}, \textsc{j}, \textsc{e})$ assertion describes the DOM footprint of the program for iteration $\textsc{i}$ with $\textsc{j}$ elements removed (i.e. ($\textsc{i}$, $\textsc{j}$) tracks the progress of the loop). As before, the fld assertion comprises two parts, with the first part describing the folded DOM footprint (the tree assertion), and the second part providing an unfolding mechanism to get at the target element node at each iteration.

The $\mathsf{unfld}(\textsc{i}, \textsc{j}, \textsc{e})$ assertion describes the unfolded tree resources at step $(\textsc{i}, \textsc{j})$ and comprises three parts. The first part, $\mathsf{S}(\textsc{i}, \textsc{j})$, describes the safe “img” elements inspected so far (those that have been processed at previous iterations and have been deemed safe). The second part, $\mathsf{t}(\textsc{i}, \textsc{j})$, describes the next “img” element $\textsc{c}$ to be inspected. The third part, $((\cdots * \mathsf{p}(\textsc{i}, \textsc{j}) \mathrel{-\!\!*} \ldots) \wedge (\cdots * \mathsf{r}(\textsc{i}, \textsc{j}) \mathrel{-\!\!*} \ldots))$, describes the *remainder* of the tree such that i) if combined with the previous safe elements and the *$\underline{p}$rocessed* element $\mathsf{p}(\textsc{i}, \textsc{j})$ (where $\textsc{c}$ is left untouched), then it produces the unfolded tree with the iteration index $\textsc{i}$ incremented; and ii) if combined with the previous safe elements and the modified tree $\mathsf{r}(\textsc{i}, \textsc{j})$ (where $\textsc{c}$ is $\underline{r}$emoved), then it results in the unfolded tree with the removal index $\textsc{j}$ incremented. That is, this third part allows us to advance the progress of the loop depending on the result of inspecting the current “img” element.

As described above, the $\mathsf{t}(\textsc{i}, \textsc{j})$, describes the “img” element $\textsc{c}$ to be inspected at step $\textsc{i}{+}\textsc{j}$ (i.e. $\textsc{c}{=}|\textsc{l}_\textsc{j}|^\textsc{i}$, the $\textsc{i}^{\text{th}}$ item in the current NodeList $\textsc{l}_\textsc{j}$). Either element $\textsc{c}$ has no “src” attribute, or the “src” attribute is included in the footprint and its value may or may not be blacklisted. Recall that if the value of “src” on $\textsc{c}$ is blacklisted, then $\textsc{c}$ is removed from the tree. As such, the parent of $\textsc{c}$, namely $\textsc{p}_\textsc{c}$, is also included in the footprint to allow for the removal of $\textsc{c}$ from the child list of $\textsc{p}_\textsc{c}$ (as per the precondition of `removeChild` operation - see Fig. 5.2).

The $\mathsf{p}(\textsc{i}, \textsc{j})$ describes the result of inspecting $\textsc{c}$ when $\textsc{c}$ either has no "src" attribute, or the value of its "src" attribute is not blacklisted: both $\textsc{c}$ and its parent $\textsc{p}_\textsc{c}$ remain unchanged. Analogously, the $\mathsf{r}(\textsc{i}, \textsc{j})$ describes the result of inspecting $\textsc{c}$ when it has a "src" attribute with a blacklisted value: $\textsc{c}$ is removed from the child forest of its parent $\textsc{p}_\textsc{c}$.

We give a proof sketch of `adBlocker2` in Figs. 5.12-5.13. The precondition (defined as P in Fig. 5.11) comprises the variable store, the folded *unprocessed* ($\textsc{i}{=}\textsc{j}{=}0$) tree, and an empty spot in the grove for the removed "img" elements. Similarly, the postcondition comprises the variable store, the folded, *fully processed* ($\textsc{i}{=}|\textsc{l}_\textsc{r}|$ and $\textsc{j}{=}\textsc{r}{=}|\textsc{l}|-1$) tree with its tag listeners extended with a new listener for "img" and the grove extended with the removed "img" elements. Observe that rather than explicitly tracking the removed "img" elements in the grove, we describe them by the weaker assertion `true`. However, as we demonstrated in the specification of `adblocker1` (Fig. 5.9), it is straightforward to track the removed elements in the grove. We opt for the weaker specification to keep the proof less verbose.

```
adblocker2 ≜
    imgs := n.getElementsByTagName("img");
    len := imgs.length(); i = 0;
    while(i < len){
      c := imgs.item(i);
      url := c.getAttribute("src");
      isB := isBlackListed(url);
      if(url && isB){
        p := c.parentNode;
        p.removeChild(c);
        len := imgs.length();
      } else {
        i = i+1;
    } }
```

$$\{P\}\ \texttt{adblocker2}\ \{Q\}$$

where

$$
\begin{aligned}
P &\triangleq \mathsf{vars}_{\mathsf{ab2}} * \mathsf{fld}(0, 0, \mathrm{E}) * \zeta \mapsto \varnothing_g \\
Q &\triangleq \exists \mathrm{E}', \mathrm{R}'.\ \mathsf{vars}_{\mathsf{ab1}} * \mathsf{fld}(|\mathrm{L_R}|, \mathrm{R}, \mathrm{E}') * \zeta \mapsto \varnothing_g \oplus \mathsf{true} \\
&\qquad * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}') \dot{\in} \mathrm{E}' * \mathsf{true} \\
\mathsf{vars}_{\mathsf{ab2}} &\triangleq \mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}-, \texttt{len}{:}-, \texttt{i}{:}-, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-) \\
\mathsf{fld}(\mathrm{I}, \mathrm{J}, \mathrm{E}) &\triangleq \mathsf{tree}(\mathrm{I}, \mathrm{J}, \mathrm{E}) * ((\forall \mathrm{I}', \mathrm{J}', \mathrm{E}'.\mathsf{tree}(\mathrm{I}', \mathrm{J}', \mathrm{E}') \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}', \mathrm{J}', \mathrm{E}')) \wedge \mathsf{emp}) \\
\mathsf{tree}(\mathrm{I}, \mathrm{J}, \mathrm{E}) &\triangleq \alpha \mapsto \mathrm{S_N}[\eta, \mathrm{T_J}]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{srch}(\mathrm{T_J}, \text{“img”}, \mathrm{L_J}) \\
\mathsf{unfld}(\mathrm{I,J,E}) &\triangleq \exists \overline{\beta, \gamma, \delta, \delta^1, \delta^2, \epsilon}^{\mathrm{L_J}}.\ \mathsf{S}(\mathrm{I}, \mathrm{J}) * \mathsf{t}(\mathrm{I}, \mathrm{J}) \\
&\qquad * \big((\mathsf{S}(\mathrm{I,J}) * \mathsf{p}(\mathrm{I,J}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I{+}1,J,E})) \wedge (\mathsf{S}(\mathrm{I,J}) * \mathsf{r}(\mathrm{I,J}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I,J{+}1,E}))\big) \\
\mathsf{t}(\mathrm{I}, \mathrm{J}) &\triangleq \exists \mathrm{C}.\ \mathrm{C} \doteq |\mathrm{L_J}|^{\mathrm{I}} * \beta_{\mathrm{C}} \mapsto (\mathrm{S_C})_{\mathrm{P_C}}[\gamma_{\mathrm{C}}, \delta^1_{\mathrm{C}} \otimes \mathrm{img}_{\mathrm{C}}[\mathrm{A_C}, \delta_{\mathrm{C}}]^{\mathrm{E_C}}_{\mathrm{F_C}} \otimes \delta^2_{\mathrm{C}}]^{\mathrm{E}'_{\mathrm{C}}}_{\mathrm{F}'_{\mathrm{C}}} \\
&\qquad * \big(\mathsf{out}_{\mathsf{n}}(\mathrm{A_C}, \text{“src”}) \vee (\mathrm{A_C} \doteq \epsilon_{\mathrm{C}} \odot \mathrm{src}_{\mathrm{M_C}}[\mathrm{T_C}]_{\mathrm{F}''_{\mathrm{C}}} * \mathsf{val}(\mathrm{T_C}, \mathrm{V_C}))\big) \\
\mathsf{p}(\mathrm{I}, \mathrm{J}) &\triangleq \exists \mathrm{C}.\ \mathrm{C} \doteq |\mathrm{L_J}|^{\mathrm{I}} * \beta_{\mathrm{C}} \mapsto (\mathrm{S_C})_{\mathrm{P_C}}[\gamma_{\mathrm{C}}, \delta^1_{\mathrm{C}} \otimes \mathrm{img}_{\mathrm{C}}[\mathrm{A_C}, \delta_{\mathrm{C}}]^{\mathrm{E_C}}_{\mathrm{F_C}} \otimes \delta^2_{\mathrm{C}}]^{\mathrm{E}'_{\mathrm{C}}}_{\mathrm{F}'_{\mathrm{C}}} \\
&\qquad * \big(\mathsf{out}_{\mathsf{n}}(\mathrm{A_C}, \text{“src”}) \vee (\mathrm{A_C} \doteq \epsilon_{\mathrm{C}} \odot \mathrm{src}_{\mathrm{M_C}}[\mathrm{T_C}]_{\mathrm{F}''_{\mathrm{C}}} * \mathsf{val}(\mathrm{T_C}, \mathrm{V_C}) * \neg\mathsf{isB}(\mathrm{V_C}))\big) \\
\mathsf{r}(\mathrm{I}, \mathrm{J}) &\triangleq \exists \mathrm{C}.\ \mathrm{C} \doteq |\mathrm{L_J}|^{\mathrm{I}} * \beta_{\mathrm{C}} \mapsto (\mathrm{S_C})_{\mathrm{P_C}}[\gamma_{\mathrm{C}}, \delta^1_{\mathrm{C}} \otimes \delta^2_{\mathrm{C}}]^{\mathrm{E}'_{\mathrm{C}}}_{\mathrm{F}'_{\mathrm{C}}} \\
&\qquad * \mathrm{A_C} \doteq \epsilon_{\mathrm{C}} \odot \mathrm{src}_{\mathrm{M_C}}[\mathrm{T_C}]_{\mathrm{F}''_{\mathrm{C}}} * \mathsf{val}(\mathrm{T_C}, \mathrm{V_C}) * \mathsf{isB}(\mathrm{V_C}) \\
\mathsf{S}(\mathrm{I}, \mathrm{J}) &\triangleq \exists \mathrm{S}_s.\ \mathrm{S}_s \doteq \{(\mathrm{N}, \mathrm{K}) \mid \mathrm{N} {=} |\mathrm{L_J}|^{\mathrm{K}} \wedge \mathrm{K} < \mathrm{I}\} \\
&\qquad \mathop{\circledast}_{(\mathrm{C,K}) \in \mathrm{S}_s} \left(\begin{array}{l} \beta_{\mathrm{K}} \mapsto (\mathrm{S_C})_{\mathrm{P_C}}[\gamma_{\mathrm{K}}, \delta^1_{\mathrm{K}} \otimes \mathrm{img}_{\mathrm{C}}[\mathrm{A_C}, \delta_{\mathrm{K}}]^{\mathrm{E_C}}_{\mathrm{F_C}} \otimes \delta^2_{\mathrm{K}}]^{\mathrm{E}'_{\mathrm{C}}}_{\mathrm{F}'_{\mathrm{C}}} * \\ (\mathsf{out}_{\mathsf{n}}(\mathrm{A_C}, \text{“src”}) \vee (\mathrm{A_C} \doteq \epsilon_{\mathrm{K}} \odot \mathrm{src}_{\mathrm{M_C}}[\mathrm{T_C}]_{\mathrm{F}''_{\mathrm{C}}} * \mathsf{val}(\mathrm{T_C}, \mathrm{V_C}) * \neg\mathsf{isB}(\mathrm{V_C}))) \end{array} \right)
\end{aligned}
$$

Figure 5.11.: The `adBlocker2` client program and its specification

```
$\{\mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}-, \texttt{len}{:}-, \texttt{i}{:}-, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-) * \mathsf{fld}(0,0,\mathrm{E}) * \zeta \mapsto \varnothing_g\}$
1. imgs := n.getElementsByTagName("img");
   $\{\exists \mathrm{R}', \mathrm{E}'.\ \mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}\mathrm{R}', \texttt{len}{:}-, \texttt{i}{:}-, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-)$
   $* \zeta \mapsto \varnothing_g * \mathrm{E}' \dot{\subseteq} \mathrm{E} * (\text{“img”}, \mathrm{R}') \dot{\in} \mathrm{E}' * \mathsf{true}$
   $* \alpha \mapsto \mathrm{S}_\mathrm{N}[\eta, \mathrm{T}_0]_\mathrm{F}^{\mathrm{E}'} * \mathsf{srch}(\mathrm{T}_0, \text{“img”}, \mathrm{L}_0) * ((\forall \mathrm{I},\mathrm{J},\mathrm{E}.\ \mathsf{tree}(\mathrm{I},\mathrm{J},\mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I},\mathrm{J},\mathrm{E})) \land \mathsf{emp})\}$
2. len := imgs.length; i = 0;
   $\{\exists \mathrm{R}', \mathrm{E}'.\ \mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}\mathrm{R}', \texttt{len}{:}\,|\mathrm{L}_0|\,, \texttt{i}{:}0, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-)$
   $* \zeta \mapsto \varnothing_g * \mathrm{E}' \dot{\subseteq} \mathrm{E} * (\text{“img”}, \mathrm{R}') \dot{\in} \mathrm{E}' * \mathsf{true}$
   $* \alpha \mapsto \mathrm{S}_\mathrm{N}[\eta, \mathrm{T}_0]_\mathrm{F}^{\mathrm{E}'} * \mathsf{srch}(\mathrm{T}_0, \text{“img”}, \mathrm{L}_0) * ((\forall \mathrm{I},\mathrm{J},\mathrm{E}.\ \mathsf{tree}(\mathrm{I},\mathrm{J},\mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I},\mathrm{J},\mathrm{E})) \land \mathsf{emp})\}$
   $\{\exists \mathrm{R}', \mathrm{E}'.\ \mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}\mathrm{R}', \texttt{len}{:}\,|\mathrm{L}_0|\,, \texttt{i}{:}0, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-)$
   $* \zeta \mapsto \varnothing_g * \mathrm{E}' \dot{\subseteq} \mathrm{E} * (\text{“img”}, \mathrm{R}') \dot{\in} \mathrm{E}' * \mathsf{fld}(0, 0, \mathrm{E}') * \mathsf{true}\}$
   $\{\exists \mathrm{R}', \mathrm{E}', \mathrm{I}, \mathrm{J}.\ \mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}\mathrm{R}', \texttt{len}{:}\,|\mathrm{L}_\mathrm{J}|\,, \texttt{i}{:}\mathrm{I}, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-)$
   $* \zeta \mapsto \varnothing_g \oplus \mathsf{true} * \mathrm{E}' \dot{\subseteq} \mathrm{E} * (\text{“img”}, \mathrm{R}') \dot{\in} \mathrm{E}' * \mathsf{fld}(\mathrm{I}, \mathrm{J}, \mathrm{E}') * \mathsf{true}\}$
3. while(i<len) {
   $\{\exists \mathrm{R}', \mathrm{E}', \mathrm{I}, \mathrm{J}.\ \mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}\mathrm{R}', \texttt{len}{:}\,|\mathrm{L}_\mathrm{J}|\,, \texttt{i}{:}\mathrm{I}, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-)$
   $* \zeta \mapsto \varnothing_g \oplus \mathsf{true} * \mathrm{E}' \dot{\subseteq} \mathrm{E} * (\text{“img”}, \mathrm{R}') \dot{\in} \mathrm{E}' * \mathsf{fld}(\mathrm{I}, \mathrm{J}, \mathrm{E}') * \mathrm{I} \dot{<} |\mathrm{L}_\mathrm{J}| * \mathsf{true}\}$
   $\{\exists \mathrm{R}', \mathrm{E}', \mathrm{I}, \mathrm{J}.\ \mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}\mathrm{R}', \texttt{len}{:}\,|\mathrm{L}_\mathrm{J}|\,, \texttt{i}{:}\mathrm{I}, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-)$
   $* \zeta \mapsto \varnothing_g \oplus \mathsf{true} * \mathrm{E}' \dot{\subseteq} \mathrm{E} * (\text{“img”}, \mathrm{R}') \dot{\in} \mathrm{E}' * \mathrm{I} \dot{<} |\mathrm{L}_\mathrm{J}| * \mathsf{true}$
   $* \alpha \mapsto \mathrm{S}_\mathrm{N}[\eta, \mathrm{T}_\mathrm{J}]_\mathrm{F}^{\mathrm{E}'} * \mathsf{srch}(\mathrm{T}_\mathrm{J}, \text{“img”}, \mathrm{L}_\mathrm{J}) * ((\forall \mathrm{I},\mathrm{J},\mathrm{E}.\ \mathsf{tree}(\mathrm{I},\mathrm{J},\mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I},\mathrm{J},\mathrm{E})) \land \mathsf{emp})\}$
4.    c := imgs.item(i);
   $\{\exists \mathrm{R}', \mathrm{E}', \mathrm{I}, \mathrm{J}.\ \mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}\mathrm{R}', \texttt{len}{:}\,|\mathrm{L}_\mathrm{J}|\,, \texttt{i}{:}\mathrm{I}, \texttt{c}{:}\,|\mathrm{L}_\mathrm{J}|^\mathrm{I}, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-)$
   $* \zeta \mapsto \varnothing_g \oplus \mathsf{true} * \mathrm{E}' \dot{\subseteq} \mathrm{E} * (\text{“img”}, \mathrm{R}') \dot{\in} \mathrm{E}' * \mathrm{I} \dot{<} |\mathrm{L}_\mathrm{J}| * \mathsf{true}$
   $* \alpha \mapsto \mathrm{S}_\mathrm{N}[\eta, \mathrm{T}_\mathrm{J}]_\mathrm{F}^{\mathrm{E}'} * \mathsf{srch}(\mathrm{T}_\mathrm{J}, \text{“img”}, \mathrm{L}_\mathrm{J}) * ((\forall \mathrm{I},\mathrm{J},\mathrm{E}.\ \mathsf{tree}(\mathrm{I},\mathrm{J},\mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I},\mathrm{J},\mathrm{E})) \land \mathsf{emp})\}$
   $\{\exists \mathrm{R}', \mathrm{E}', \mathrm{I}, \mathrm{J}.\ \mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}\mathrm{R}', \texttt{len}{:}\,|\mathrm{L}_\mathrm{J}|\,, \texttt{i}{:}\mathrm{I}, \texttt{c}{:}\,|\mathrm{L}_\mathrm{J}|^\mathrm{I}, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-)$
   $* \zeta \mapsto \varnothing_g \oplus \mathsf{true} * \mathrm{E}' \dot{\subseteq} \mathrm{E} * (\text{“img”}, \mathrm{R}') \dot{\in} \mathrm{E}' * \mathsf{true} * \exists \overline{\beta, \gamma, \delta, \delta^1, \delta^2, \epsilon}^{\mathrm{L}_\mathrm{J}}.\ \mathsf{S}(\mathrm{I}, \mathrm{J}) * \mathsf{t}(\mathrm{I}, \mathrm{J})$
   $* ((\mathsf{S}(\mathrm{I}, \mathrm{J}) * \mathsf{p}(\mathrm{I}, \mathrm{J}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I}{+}1, \mathrm{J}, \mathrm{E})) \land (\mathsf{S}(\mathrm{I}, \mathrm{J}) * \mathsf{r}(\mathrm{I}, \mathrm{J}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I}, \mathrm{J}{+}1, \mathrm{E})))$
   $* ((\forall \mathrm{I}, \mathrm{J}, \mathrm{E}.\ \mathsf{tree}(\mathrm{I}, \mathrm{J}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{J}, \mathrm{E})) \land \mathsf{emp})\}$
5.    url := c.getAttribute("src");
6.    isB := isBlackListed(url)
   $\{\exists \mathrm{R}', \mathrm{E}', \mathrm{I}, \mathrm{J}, \mathrm{U}, \mathrm{B}.\ \mathsf{vars}(\texttt{n}{:}\mathrm{N}, \texttt{imgs}{:}\mathrm{R}', \texttt{len}{:}\,|\mathrm{L}_\mathrm{J}|\,, \texttt{i}{:}\mathrm{I}, \texttt{c}{:}\,|\mathrm{L}_\mathrm{J}|^\mathrm{I}, \texttt{url}{:}\mathrm{U}, \texttt{isB}{:}\mathrm{B}, \texttt{p}{:}-)$
   $* \zeta \mapsto \varnothing_g \oplus \mathsf{true} * \mathrm{E}' \dot{\subseteq} \mathrm{E} * (\text{“img”}, \mathrm{R}') \dot{\in} \mathrm{E}' * \mathsf{true} * \exists \overline{\beta, \gamma, \delta, \delta^1, \delta^2, \epsilon}^{\mathrm{L}_\mathrm{J}}.\ \mathsf{S}(\mathrm{I}, \mathrm{J})$
   $* \Big( \big(\mathsf{p}(\mathrm{I}, \mathrm{J}) * (\mathrm{U} \doteq \text{“”} \lor (\mathrm{B} \doteq 0)\big) \lor (\mathsf{t}(\mathrm{I}, \mathrm{J}) * \mathrm{U} \doteq \mathrm{V}_\mathrm{C} * \mathrm{B} \doteq 1) \Big)$
   $* ((\mathsf{S}(\mathrm{I}, \mathrm{J}) * \mathsf{p}(\mathrm{I}, \mathrm{J}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I}{+}1, \mathrm{J}, \mathrm{E})) \land (\mathsf{S}(\mathrm{I}, \mathrm{J}) * \mathsf{r}(\mathrm{I}, \mathrm{J}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I}, \mathrm{J}{+}1, \mathrm{E})))$
   $* ((\forall \mathrm{I}, \mathrm{J}, \mathrm{E}.\ \mathsf{tree}(\mathrm{I}, \mathrm{J}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{J}, \mathrm{E})) \land \mathsf{emp})\}$
```

Figure 5.12.: A proof sketch of the `adBlocker2` program

```
       $\left\{\begin{array}{l} \exists R', E', I, J, U, B.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}\,|L_J|\,, \texttt{i}{:}I, \texttt{c}{:}\,|L_J|^I\,, \texttt{url}{:}U, \texttt{isB}{:}B, \texttt{p}{:}-) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{true} * \overline{\exists \beta, \gamma, \delta, \delta^1, \delta^2, \epsilon}^{L_J}.\ \mathsf{S}(I, J) \\ *\,\Big( \big(\mathsf{p}(I, J) * (U \dot{=} \text{“”} \lor (B \dot{=} 0)\big) \lor \big(\mathsf{t}(I, J) * U \dot{=} V_C * B \dot{=} 1\big) \Big) \\ *\,\big((\mathsf{S}(I, J) * \mathsf{p}(I, J) \twoheadrightarrow \mathsf{tree}(I{+}1, J, E)) \land (\mathsf{S}(I, J) * \mathsf{r}(I, J) \twoheadrightarrow \mathsf{tree}(I, J{+}1, E))\big) \\ *\,\big((\forall I, J, E.\ \mathsf{tree}(I, J, E) \twoheadrightarrow \mathsf{unfld}(I, J, E)) \land \mathsf{emp}\big) \end{array}\right\}$
7.     if (url && isB) {
       $\left\{\begin{array}{l} \exists R', E', I, J.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}\,|L_J|\,, \texttt{i}{:}I, \texttt{c}{:}\,|L_J|^I\,, \texttt{url}{:}V_C, \texttt{isB}{:}1, \texttt{p}{:}-) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{true} * \overline{\exists \beta, \gamma, \delta, \delta^1, \delta^2, \epsilon}^{L_J}.\ \mathsf{S}(I, J) * \mathsf{t}(I, J) \\ *\,\big((\mathsf{S}(I, J) * \mathsf{p}(I, J) \twoheadrightarrow \mathsf{tree}(I{+}1, J, E)) \land (\mathsf{S}(I, J) * \mathsf{r}(I, J) \twoheadrightarrow \mathsf{tree}(I, J{+}1, E))\big) \\ *\,\big((\forall I, J, E.\ \mathsf{tree}(I, J, E) \twoheadrightarrow \mathsf{unfld}(I, J, E)) \land \mathsf{emp}\big) \end{array}\right\}$
8.     p := c.parentNode;
9.     p.removeChild(c);
       $\left\{\begin{array}{l} \exists R', E', I, J.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}\,|L_J|\,, \texttt{i}{:}I, \texttt{c}{:}\,|L_J|^I\,, \texttt{url}{:}V_C, \texttt{isB}{:}1, \texttt{p}{:}P_C) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{true} * \overline{\exists \beta, \gamma, \delta, \delta^1, \delta^2, \epsilon}^{L_J}.\ \mathsf{S}(I, J) * \mathsf{r}(I, J) \\ *\,\big((\mathsf{S}(I, J) * \mathsf{p}(I, J) \twoheadrightarrow \mathsf{tree}(I{+}1, J, E)) \land (\mathsf{S}(I, J) * \mathsf{r}(I, J) \twoheadrightarrow \mathsf{tree}(I, J{+}1, E))\big) \\ *\,\big((\forall I, J, E.\ \mathsf{tree}(I, J, E) \twoheadrightarrow \mathsf{unfld}(I, J, E)) \land \mathsf{emp}\big) \end{array}\right\}$
       $\left\{\begin{array}{l} \exists R', E', I, J.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}\,|L_J|\,, \texttt{i}{:}I, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{fld}(I, J{+}1, E') * \mathsf{true} \end{array}\right\}$
10.    len := imgs.length;
       $\left\{\begin{array}{l} \exists R', E', I, J.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}\,|L_{J+1}|\,, \texttt{i}{:}I, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{fld}(I, J{+}1, E') * \mathsf{true} \end{array}\right\}$
       $\left\{\begin{array}{l} \exists R', E', I, J.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}\,|L_J|\,, \texttt{i}{:}I, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{fld}(I, J, E') * \mathsf{true} \end{array}\right\}$
11.    } else {
       $\left\{\begin{array}{l} \exists R', E', I, J.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}\,|L_J|\,, \texttt{i}{:}I, \texttt{c}{:}\,|L_J|^I\,, \texttt{url}{:}U, \texttt{isB}{:}B, \texttt{p}{:}-) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{true} * \overline{\exists \beta, \gamma, \delta, \delta^1, \delta^2, \epsilon}^{L_J}.\ \mathsf{S}(I, J) * \mathsf{p}(I, J) \\ *\,\big((\mathsf{S}(I, J) * \mathsf{p}(I, J) \twoheadrightarrow \mathsf{tree}(I{+}1, J, E)) \land (\mathsf{S}(I, J) * \mathsf{r}(I, J) \twoheadrightarrow \mathsf{tree}(I, J{+}1, E))\big) \\ *\,\big((\forall I, J, E.\ \mathsf{tree}(I, J, E) \twoheadrightarrow \mathsf{unfld}(I, J, E)) \land \mathsf{emp}\big) \end{array}\right\}$
       $\left\{\begin{array}{l} \exists R', E', I, J.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}\,|L_J|\,, \texttt{i}{:}I, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{fld}(I{+}1, J, E') * \mathsf{true} \end{array}\right\}$
12.    i = i+1
       $\left\{\begin{array}{l} \exists R', E', I, J.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}\,|L_J|\,, \texttt{i}{:}I{+}1, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{fld}(I{+}1, J, E') * \mathsf{true} \end{array}\right\}$
       $\left\{\begin{array}{l} \exists R', E', I, J.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}\,|L_J|\,, \texttt{i}{:}I, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{fld}(I, J, E') * \mathsf{true} \end{array}\right\}$
13.    } }
$\left\{\begin{array}{l} \exists R', E'.\ \mathsf{vars}(\texttt{n}{:}N, \texttt{imgs}{:}R', \texttt{len}{:}-, \texttt{i}{:}-, \texttt{c}{:}-, \texttt{url}{:}-, \texttt{isB}{:}-, \texttt{p}{:}-) \\ *\,\zeta \mapsto \varnothing_g \oplus \mathsf{true} * E' \dot{\subseteq} E * (\text{“img”}, R') \dot{\in} E' * \mathsf{fld}(\,|L_R|\,, R, E') * \mathsf{true} \end{array}\right\}$
```

Figure 5.13.: A proof sketch of the `adBlocker2` program (continued)

6. Technical Background: Refinement

In what preceded we discussed abstraction and explored its benefits for library specification and client reasoning. While abstraction is crucial for simpler and more modular specification and client-side reasoning, it is as important to refine the abstraction in order to move closer to the implementation level. In particular, it is important to verify that a particular implementation does indeed satisfy the abstract specification provided to the clients of a library. Given an abstract specification of a library, traditional *refinement* techniques [30, 3] produce a correct implementation that meets that specification.

Filipović, O'Hearn, Torp-Smith and Yang have studied data refinement for local reasoning [20], by considering modules built on top of the standard separation logic heap model. They observed that a module client can break the refinement between an abstract module and its concrete implementation by dereferencing pointers into its internal state, and thereby violating the abstract boundary of the module. In particular, they studied a simple memory allocator maintaining a set of available free cells. They noted that at the concrete level a client may violate the free cell set via memory pointers that have been previously deallocated, whereas at the abstract level the free cell set is unaffected by such accesses. To rectify this, they introduced a modified operational semantics that treats such memory accesses as faulting executions, thereby "blaming the client" for breaking the abstraction boundary. This in turn meant that both the module and its client must use the same state (data) model which is not always feasible.

In [16], Dinsdale-Young, Gardner and Wheelhouse apply data refinement to local reasoning in order to show the soundness of implementations of library modules specified in context logic. In contrast to [20], they worked with the axiomatic, rather than operational semantics of the language and defined proof transformations to show that concrete implementations simulate abstract specifications. This way, they did not need to consider badly behaved client programs as the proof system only makes guarantees about well-behaved client programs. They developed general techniques for verifying the correctness of library implementations with respect to their specifications by defining *translation functions* that relate abstract (high-level) states to concrete (low-level) states. Their translation functions fell in one of two categories described shortly: *locality-breaking* translations or *locality-preserving* translations.

The correctness of both approaches relies on the data refinement technique known as forward simulation (also known as L-simulation) [3]. Simulations relate abstract states and library operations to concrete states and the implementations of operations. Using forward simulation, one must show that the result of transforming an abstract state to another via a library operation and subsequently refining the result is the same as refining the same initial abstract state and then transforming it via the implementation of the same library operation. More concretely, let $\tau \triangleq (\llbracket . \rrbracket, \llbracket . \rrbracket)$ denote a translation function comprising a state translation function $\llbracket . \rrbracket$ relating sets of abstract states to sets of concrete states, and an implementation function $\llbracket . \rrbracket$ mapping library operations onto their implementations. Given any two sets of states p and q and a library client program $\mathtt{C}$ we must then have:

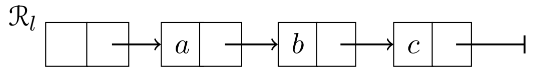
$$\begin{array}{rrcl} & \{p\}\ \mathtt{C}\ \{q\} & \Longrightarrow & \tau : \{p\}\ \mathtt{C}\ \{q\} \\ \text{where} & \tau : \{p\}\ \mathtt{C}\ \{q\} & \overset{\text{def}}{\Longleftrightarrow} & \{\llbracket p \rrbracket\}\ \llbracket \mathtt{C} \rrbracket\ \{\llbracket q \rrbracket\} \end{array} \tag{6.1}$$

In [16, 59], the authors applied this forward simulation technique in order to show the implementation soundness of several pedagogical libraries. Later in [33], Jensen and Birkedal further studied the locality-breaking translations for the refinement of libraries specified in separation logic. The works in all three of [16], [59] and [33] study the refinement of libraries in *sequential* settings. In [23], we revised the results from [16, 59] for the refinement of SSL-specified libraries in *concurrent* settings.

To better understand the two varieties of translations, in §6.1 we present a sequential implementation of the list module studied in §2.1.2 and describe how we establish its correctness using a locality-breaking translation. We highlight the limitations of locality-breaking translations in *scalability* and *concurrency*, motivating the need for locality-preserving translations. In §6.2 we present an overview of locality-preserving translations and describe how they overcome the limitations of locality-breaking translations in scalability and concurrency. We demonstrate a limitation of locality-preserving translations, namely their *complexity*, especially in sequential settings. In §6.3 we introduce *hybrid* translations that combine the strengths of both locality-breaking and locality-preserving translations for *simple*, *scalable* refinement of sequential libraries.

6.1. Locality-breaking Translations

Consider the *sequential* implementation of the list operations in Fig. 6.1 where an abstract list is implemented as a singly-linked list in the heap, with a sentinel node at the head of the list. Each list node comprises two adjacent cells, respectively recording its value and the next pointer. The designated sentinel node similarly comprises two adjacent cells, respectively recording an arbitrary value and the location of the first list node (or `null` when the list is empty). For instance, the abstract list depicted in Fig. 2.1a is implemented as follows where the cell at $\mathcal{R}_l$ denotes the sentinel node, the $\rightarrow$ arrows denote "next" pointers, and $\rightarrow\!\!|$ denotes a `null` pointer:



(6.2)

The list implementation in Fig. 6.1 is given in the while language WL presented in §3, where we write `n.value` for `n` and write `n.next` for `n+1`. We assume that this implementation is used in *sequential* settings. That is, we remove the parallel composition construct ($\texttt{C}_1\texttt{||}\texttt{C}_2$) from the WL language and thus assume that at any given time no client program written in WL may run two list operations in parallel by two distinct threads. Later, we will discuss library implementations in *concurrent* settings.

As described above, in order to establish the correctness of the list operations in Fig. 6.1, we provide a refinement proof demonstrating that everything one can prove about the client programs calling the list library operations specified in Fig. 2.2, can also be proved about the same programs calling the implementation of the list operations in Fig. 6.1 instead.

```
x.add(n) ≜                           x.remove(n) ≜
  var c, m in {                        var c, p, next in {
    c = x;                               p = x;  c = [x.next];
    while([c.next] != null){             while(c != null && [c.value] != n){
      c = [c.next]                         p = c;  c = [p.next]
    }                                    }
    m = alloc(2);                        if (c != null){
    [m.value] = n;                         next = [c.next];
    [m.next] = null;                       [p.next] = next;
    [c.next] = m                           free(c, 2)
  }                                    } }

                  r := x.item(i) ≜
                    if(i >= 0){
                      var j, c in {
                        j = 0;  c = [x.next];
                        while(c != null && i != j){
                          j++;  c = [c.next]
                        }
                        r = c
                    } }
```

Figure 6.1.: A sequential implementation of the list operations

To this end, we first define a *state translation function*, $\llfloor . \rrfloor$, mapping abstract list states at the specification level onto concrete list states at the implementation level. We then define a *substitutive implementation function*, $[\![.]\!]$, that replaces each call to a list library operation in `C` with the correspondingly named program given in Fig. 6.1. For instance, the $[\![$`x.add(n)`$]\!]$ is defined as the `x.add(n)` program in Fig. 6.1 and $[\![\texttt{C}_1\texttt{;}\texttt{C}_2]\!] \triangleq [\![\texttt{C}_1]\!]\,\texttt{;}\,[\![\texttt{C}_2]\!]$.

We must next show that given the translation $\tau \triangleq (\llfloor . \rrfloor , [\![.]\!])$, every triple of the form $\{p\}$ `C` $\{q\}$ one can prove at the specification level is *refined correctly*, written $\tau : \{p\}$ `C` $\{q\}$, and can also be proved at the implementation level:

$$\{p\}\ \texttt{C}\ \{q\} \implies \tau : \{p\}\ \texttt{C}\ \{q\} \qquad \text{(CorrRef)}$$

Finally, we must formulate the definition of $\tau : \{p\}$ `C` $\{q\}$ describing what it means to refine a triple correctly.

We begin by defining the *logical state translation function*, $\llbracket . \rrbracket$, that given an abstract state of the form $((\sigma, h), \mathbf{h})$ – where (σ, h) denotes a WLogic logical state comprising a stack and a heap (Def. 33), and $\mathbf{h}$ denotes a logical list heap such as the ones in Fig. 2.1) – it maps the logical list heap $\mathbf{h}$ onto sets of concrete singly-linked lists such as the one depicted in (6.2), while leaving the stack and the heap (σ, h) unchanged.

Recall that the footprint of `x.add(n)` at the abstract level is limited to the last position of the list, as described by $\mathcal{R}_l \mapsto \alpha$ in the precondition of `add` in Fig. 2.2, repeated below:

$$\left\{\mathsf{vars}(\mathtt{x}{:}\mathcal{R}_l, \mathtt{n}{:}\mathrm{N}) * \mathcal{R}_l \mapsto \alpha\right\} \mathtt{x.add(n)} \left\{\mathsf{vars}(\mathtt{x}{:}\mathcal{R}_l, \mathtt{n}{:}\mathrm{N}) * \mathcal{R}_l \mapsto \alpha {+\!\!+} [\mathrm{N}]\right\}$$

By contrast, the implementation of `x.add(n)` in Fig. 6.1 proceeds by traversing the list from the beginning until it reaches the last element in the list (`p` where `p.next` is `null`). It then extends the list with `n` by redirecting the "next" pointer of the last node (`p`) to `n`. That is, the footprint of `x.add(n)` at the concrete level encompasses the entire list. This results in a *locality mismatch* between the abstract and concrete levels, and our translation function must thus account for the additional resources required by the larger footprint of the implementation.

As such, the translation function must map both *incomplete* list heaps (e.g. $\mathbf{h}{=}\mathcal{R}_l \mapsto [a]{+\!\!+}\mathbf{x}{+\!\!+}[b]$) and *complete* list heaps (e.g. $\mathbf{h}'{=}\mathcal{R}_l \mapsto [a, b, c]$), onto *complete* concrete lists. We thus define a completing translation function that first extends an incomplete abstract list into a set of complete abstract lists, and then translates them to complete concrete lists. For instance, given the incomplete abstract list $\mathbf{h}{=}\mathcal{R}_l \mapsto [a]{+\!\!+}\mathbf{x}{+\!\!+}[b]$ in Fig. 6.2a, Fig. 6.2b depicts three possible completions of $\mathbf{h}$, with their concrete representations in Fig. 6.2c. The intuition behind completing translations is that incomplete list fragments are purely abstract notions, providing a means for reasoning about lists in a local and compositional manner. At the abstract level, the list library itself does not provide operations for creating or destroying partial lists and thus we never need to reason about *full* client programs that concern incomplete lists. As such, it suffices to demonstrate that any specification that we can prove about full client programs at the abstract level, can also be proved about the same full

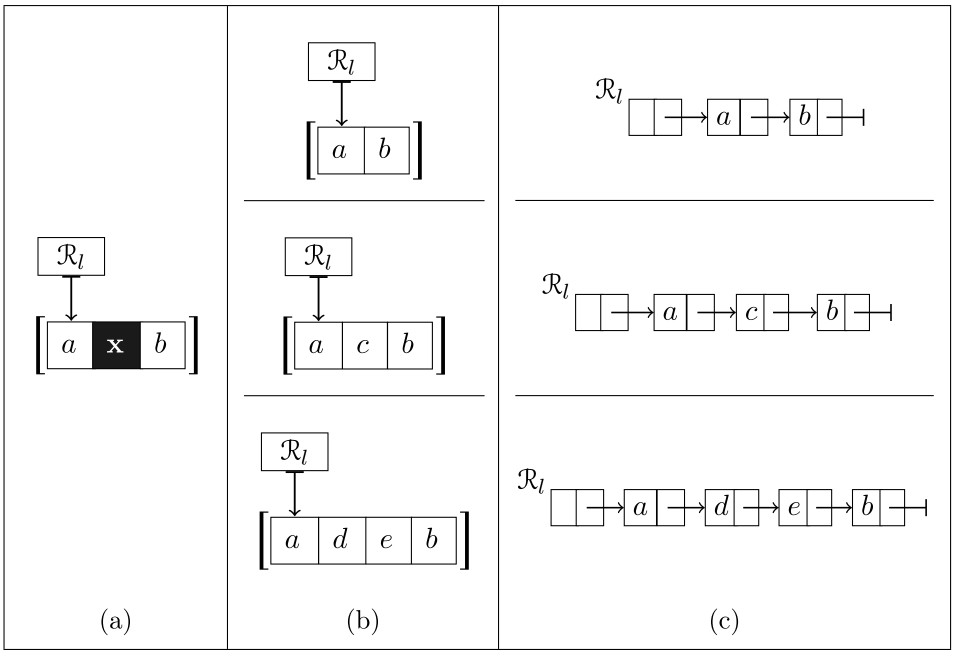


Figure 6.2.: An incomplete abstract list; several completions of (a); implementations of complete lists in (b)

programs at the implementation level. That is, our implementation need not consider incomplete lists and may solely focus on complete lists.

We next formulate the definition of $\tau : \{p\}\ \mathtt{C}\ \{q\}$ used in (CorrRef), describing what it means to refine a triple correctly. Since our translation function $\llparenthesis . \rrparenthesis$ extends the incomplete list heaps into complete ones introducing additional resources, to ensure the translation correctness we must show that the inclusion of these additional resources does not break frame preservation and that the translation function preserves all frames. That is, any compatible frame at the abstract level, remains compatible after translation to the concrete level. As such, we bake this frame preservation requirement into the definition of correct refinement $\tau : \{p\}\ \mathtt{C}\ \{q\}$ as follows:

$$\tau : \{p\}\ \mathtt{C}\ \{q\} \overset{\text{def}}{\iff} \forall r.\ \{\llparenthesis p * r \rrparenthesis\}\ [\![\mathtt{C}]\!]\ \{\llparenthesis q * r \rrparenthesis\} \qquad \text{(Ref)}$$

The above states that abstract specifications are preserved by the translation in all larger states: all triples $\{p\}\ \mathtt{C}\ \{q\}$ that we can prove at the abstract level, are preserved by the translation, while also preserving all compatible frames r.

Lastly, given the definition of refinement in (Ref), we must prove that (CorrRef) holds. The proof of (CorrRef) is by induction on the structure of triples (excluding the parallel composition rule for $\mathtt{C}_1 \,||\, \mathtt{C}_2$ as we do not consider concurrency in this section). The proof of inductive cases follow from the inductive hypotheses. For instance, we can prove the sequential composition case (Seq) by the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\tau : \{p\}\ \mathtt{C}_1\ \{r\}}\text{(I.H.)}}{\forall s.\ \{\llfloor p * s \rrfloor\}\ \llbracket \mathtt{C}_1 \rrbracket\ \{\llfloor r * s \rrfloor\}}\text{(Ref)} \quad \dfrac{\dfrac{}{\tau : \{r\}\ \mathtt{C}_2\ \{q\}}\text{(I.H.)}}{\forall s.\ \{\llfloor r * s \rrfloor\}\ \llbracket \mathtt{C}_2 \rrbracket\ \{\llfloor q * s \rrfloor\}}\text{(Ref)}}{\forall s.\ \{\llfloor p * s \rrfloor\}\ \llbracket \mathtt{C}_1 \rrbracket\ ;\ \llbracket \mathtt{C}_2 \rrbracket\ \{\llfloor q * s \rrfloor\}}\text{(Seq)}}{\forall s.\ \{\llfloor p * s \rrfloor\}\ \llbracket \mathtt{C}_1 ; \mathtt{C}_2 \rrbracket\ \{\llfloor q * s \rrfloor\}}(\llbracket . \rrbracket\text{ def.})}{\tau : \{p\}\ \mathtt{C}_1 ; \mathtt{C}_2\ \{q\}}\text{(Ref)} \tag{6.3}$$

For the base cases, namely those of list axioms in $\text{Axiom}_{\mathbb{L}}$ (Def. 38), we must show the correctness of our sequential implementation in Fig. 6.1. More concretely, we must show that for all list operations $\mathtt{C}$ and all list axioms $(p, \mathtt{C}, q)$ in $\text{Axiom}_{\mathbb{L}}$ (Def. 38), the following holds:

$$\forall r.\ \{\llfloor p * r \rrfloor\}\ \llbracket \mathtt{C} \rrbracket\ \{\llfloor q * r \rrfloor\}$$

where $\llbracket \mathtt{C} \rrbracket$ denotes the implementation of $\mathtt{C}$ given in Fig. 6.1 and $\llfloor . \rrfloor$ denotes the completing translation function described above. This is straightforward to establish for our simple list implementation and we omit the proof here. Instead, later in §7 we present an implementation of the DOM fragment studied in §5 and establish its correctness with respect to our axiomatic DOM specification by proving an analogous result to that of (CorrRef).

Locality-breaking (completing) translations are simple in that they do not require locality at the concrete level to match the locality at the abstract level. However, whilst defining completing translations of locality-breaking translations is simple, the correctness proof may be rather difficult as we have to show that the axioms are preserved for all possible frames and all possible completions. In particular, as we demonstrate shortly, the number of proof obligations increases as we consider more complex libraries where the completion of partial data is not straightforward. Moreover, as we demonstrate below, locality-breaking translations are not suitable for the refinement of *concurrent* libraries as they involve non-trivial linearisability proofs.

6.1.1. Locality-breaking Limitations: Scalability

The completing locality-breaking translation considered above for the list library is rather simple. In particular, the correctness of the inductive cases follows from the inductive hypotheses whilst the correctness of each list library axiom can be established by a *single* proof per axiom. More concretely, for each of the list axioms, the translation function is required to complete the partial lists in the pre- and postconditions by describing the data associated with at most a *single* abstract address α. For instance, in the case of the `x.add(n)` axiom given in Fig. 2.2 (p. 51) and repeated on p. 201, the completing translation must describe the list data associated with the single context hole α. In other words, for each axiom of the list library, it suffices to consider a single arbitrary completion of the list data in its specification (by associating an arbitrary list fragment with α when applicable). As such, we can prove the correctness of our implementation by providing a single proof per library axiom.

In general, it is not always possible to capture all possible completions of the data in library axioms succinctly with a single arbitrary completion. For instance, consider providing a (completing) locality-breaking translation of our tree library $\mathbb{T}$ in §4, with its operations axiomatised in Fig. 4.2.

Regardless of how the tree library operations are implemented, establishing the correctness of the `appendChild(n, m)` operation is going to pose a challenge. Consider the specification of `appendChild(n, m)` in Fig. 4.2, repeated below:

$$\left\{\mathsf{vars}(\mathtt{n}\!:\!\mathrm{N}, \mathtt{m}\!:\!\mathrm{M}) * \alpha \mapsto \mathrm{N}[\beta] * \gamma \mapsto \mathrm{M}[\mathrm{T}] * \mathsf{complete}(\mathrm{T})\right\}$$
$$\mathtt{appendChild(n,m)}$$
$$\left\{\mathsf{vars}(\mathtt{n}\!:\!\mathrm{N}, \mathtt{m}\!:\!\mathrm{M}) * \alpha \mapsto \mathrm{N}[\beta \otimes \mathrm{M}[\mathrm{T}]] * \gamma \mapsto \varnothing\right\}$$

Let us assume that the values associated with the logical variables N and M are as follows: N$=n$ and M$=m$. As depicted in Fig. 6.3, there are three possible ways in which the incomplete tree data in the precondition of

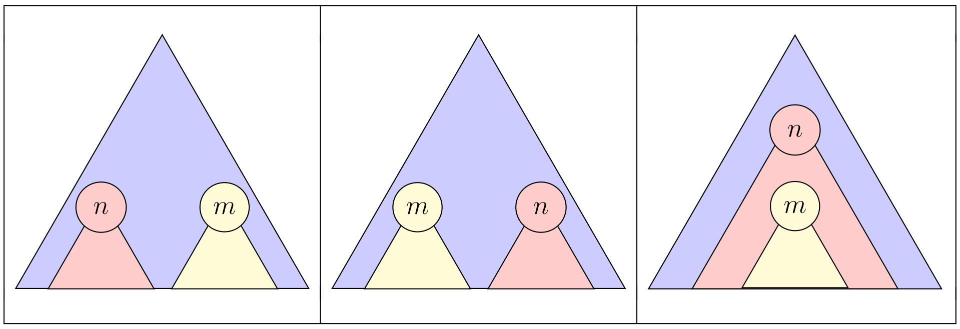


Figure 6.3.: The possible completions of the `appendChild(n, m)` resources

the axiom can be completed. That is, n and m may be in one of three possible orientations with respect to one another. Either the subtree at n is somewhere to the left of m; or the subtree at n is somewhere to the right of m; or the subtree at m is somewhere underneath n.

A particular implementation of `appendChild(n, m)` may depend on the relation between n and m, varying from one orientation to another. For instance, consider an implementation in which each node maintains a pointer to its parent. When n and m are in one of the first two orientations depicted, the implementation of `appendChild(n, m)` must accordingly update the parent pointer of m, redirecting it to n. Let us assume that when m is an immediate child of n, our implementation does not overwrite the parent pointer of m for efficiency (since doing so is an idempotent operation). As such, in those completions of the third orientation in which m is an immediate child of n, the implementation of `appendChild(n, m)` varies subtly.

In order to prove that the implementation of `appendChild(n, m)` is correct with respect to its abstract specification, we must verify the implementation in each of these three orientations. In other words, we need to provide three separate proof sketches for the single axiom of `appendChild(n, m)`. Ideally, we would like to verify the implementation of each operation once and for all: one proof sketch per axiom. That is, we would like to establish the correctness of an implementation regardless of the set of possible completions (frames). This leads us on to *locality-preserving* translations that by definition preserve all frames. This in turn allows us to show the correctness of an implementation independently of the set of possible completions.

6.1.2. Locality-breaking Limitations: Concurrency

Locality-breaking translations are not suitable for refinement in *concurrent* settings. Let us suppose that we have a *concurrent* implementation of the list library $\mathbb{L}$ operations. As before, in order to show that our implementation correctly refines the list specification given by $\text{AXIOM}_{\mathbb{L}}$ (Def. 38), we must establish the correct refinement condition in (CORRREF), with the definition of $\tau : \{p\}\ \mathtt{C}\ \{q\}$ given in (REF).

As described in §6.1, the proof of (CORRREF) is by induction over the structure of $\text{WLOGIC}_{\mathbb{L}}$ triples. The proofs of all cases, with the exception of the parallel rule $\{p_1 * p_2\}\ \mathtt{C}_1 \,||\, \mathtt{C}_2\ \{q_1 * q_2\}$, are straightforward and in most cases follow from the inductive hypotheses as shown in the derivations above (see e.g. derivation 6.3). Let us turn our focus to the parallel composition rule. Our first attempt at a derivation similar to that of (6.3) fails as we cannot connect the top part of the derivation tree to the bottom part:

$$
\dfrac{
\dfrac{
\dfrac{\dfrac{}{\tau : \{p_1\}\ \mathtt{C}_1\ \{q_1\}}\ \text{(I.H.)}}{\forall r.\, \{\lfloor\!\lfloor p_1 * r \rceil\!\rceil\}\ \llbracket \mathtt{C}_1 \rrbracket\ \{\lfloor\!\lfloor q_1 * r \rceil\!\rceil\}}\ \begin{array}{l}\text{(REF)}\\ \text{(???)}\end{array}
\quad
\dfrac{\dfrac{}{\tau : \{p_2\}\ \mathtt{C}_2\ \{q_2\}}\ \text{(I.H.)}}{\forall r.\, \{\lfloor\!\lfloor p_2 * r \rceil\!\rceil\}\ \llbracket \mathtt{C}_2 \rrbracket\ \{\lfloor\!\lfloor q_2 * r \rceil\!\rceil\}}\ \begin{array}{l}\text{(REF)}\\ \text{(???)}\end{array}
}{
\dfrac{\overset{\text{(???)}\qquad\qquad\qquad\qquad\text{(???)}}{\forall r.\ \{\lfloor\!\lfloor p_1 * p_2 * r \rceil\!\rceil\}\ \llbracket \mathtt{C}_1 \rrbracket \,||\, \llbracket \mathtt{C}_2 \rrbracket\ \{\lfloor\!\lfloor q_1 * q_2 * r \rceil\!\rceil\}}}{\forall r.\ \{\lfloor\!\lfloor p_1 * p_2 * r \rceil\!\rceil\}\ \llbracket \mathtt{C}_1 \,||\, \mathtt{C}_2 \rrbracket\ \{\lfloor\!\lfloor q_1 * q_2 * r \rceil\!\rceil\}}\ (\llbracket . \rrbracket\ \text{def.})
}\ \text{(???)}
}{\tau : \{p_1 * p_2\}\ \mathtt{C}_1 \,||\, \mathtt{C}_2\ \{q_1 * q_2\}}\ \text{(REF)}
$$

Intuitively, this is because given a frame r, although each thread in isolation preserves r, their interleaving with one another may not. That is, while executing each of $\llbracket \mathtt{C}_1 \rrbracket$ and $\llbracket \mathtt{C}_2 \rrbracket$ in isolation preserves r, interleaving the execution of $\llbracket \mathtt{C}_1 \rrbracket$ with that of $\llbracket \mathtt{C}_2 \rrbracket$, namely executing $\llbracket \mathtt{C}_1 \,||\, \mathtt{C}_2 \rrbracket$, may not.

Following our failed attempt at an inductive proof, we can alternatively proceed as follows. First, we show that all triples involving *sequential* programs are refined correctly. That is, for all p, q and for all sequential programs $\mathtt{C}$ (where $\mathtt{C}$ does not include the parallel composition construct $||$ and is not interleaved with another thread) we have:

$$\{p\}\ \mathtt{C}\ \{q\} \implies \tau : \{p\}\ \mathtt{C}\ \{q\} \qquad \text{(CORRSEQREF)}$$

Second, we show that the *concurrent* program $\texttt{C}_1 \,||\, \texttt{C}_2$ is *equivalent to a sequential* program $\texttt{C}$, and show $\tau : \{p\}\ \texttt{C}\ \{q\}$ instead, which follows immediately from (CorrSeqRef) above.

We must next define what it means for the concurrent program $\texttt{C}_1 \,||\, \texttt{C}_2$ to be equivalent to a sequential program $\texttt{C}$. The obvious first candidate is to require that our list library implementation be *linearisable* [27, 56]. A library implementation is linearisable if and only if given any two library operations $\texttt{C}_1$ and $\texttt{C}_2$, no matter how $\texttt{C}_1$ and $\texttt{C}_2$ are interleaved, it appears as if $\texttt{C}_1$ was called before $\texttt{C}_2$ (i.e. $\texttt{C}_1 ; \texttt{C}_2$) or vice versa (i.e. $\texttt{C}_2 ; \texttt{C}_1$). In other words, the library operations behave *atomically*: the observable effect of each library operation is carried out instantaneously.

In order to establish the correctness of a *concurrent* implementation with a *locality-breaking* translation, we must show that our implementation is linearisable. However, constructing linearisability proofs is known to be non-trivial and non-modular. As we demonstrate shortly in the following section, *locality-preserving* translations provide a simple alternative to linearisability for establishing the implementation correctness of concurrent libraries.

6.2. Locality-preserving Translations

Locality-breaking (completing) translations do not always scale to the refinement of complex sequential libraries as they result in several proof obligations per library axiom. Moreover, the locality-breaking approach is not suitable for the refinement of concurrent libraries as they require linearisability proofs which are non-trivial and non-modular. As such, it is often more desirable to appeal to *locality-preserving* translations that by definition preserve all frames. In short, given sets of states denoted by p and q, a state translation function $\llbracket . \rrbracket$ is *locality-preserving* if and only if it satisfies the following property:

$$\forall p, q. \quad \llbracket p * q \rrbracket = \llbracket p \rrbracket * \llbracket q \rrbracket \tag{FP}$$

The property above stipulates that locality at the abstract level match the locality at the concrete level. That is, if two sets of states p and q are compatible at the abstract level (if $p * q$ is defined), they remain compatible after translation (then $\llbracket p \rrbracket * \llbracket q \rrbracket$ is also defined), thus ensuring frame preservation.

Equipped with the (FP) property, we can now weaken the refinement correctness definition in (Ref) and thus simplify the proof obligation as:

$$\tau : \{p\}\ \mathtt{C}\ \{q\} \overset{\text{def}}{\iff} \{\llbracket p \rrbracket\}\ \llbracket \mathtt{C} \rrbracket\ \{\llbracket q \rrbracket\} \qquad (\textsc{SimpRef})$$

In other words, since the frame preservation property is maintained by the translation, it is no longer required to be established explicitly for each axiom as stipulated by (Ref), and it suffices to prove the simpler refinement condition in (SimpRef). That is, as shown in the derivation below, the frame preservation property in (FP) together with (SimpRef) imply the stronger condition outlined in (Ref).

$$\dfrac{\dfrac{\dfrac{}{\{\llbracket p \rrbracket\}\ \llbracket \mathtt{C} \rrbracket\ \{\llbracket q \rrbracket\}}\ (\textsc{SimpRef})}{\forall r.\ \{\llbracket p \rrbracket * \llbracket r \rrbracket\}\ \llbracket \mathtt{C} \rrbracket\ \{\llbracket q \rrbracket * \llbracket r \rrbracket\}}\ (\textsc{Frame})}{\forall r.\ \{\llbracket p * r \rrbracket\}\ \llbracket \mathtt{C} \rrbracket\ \{\llbracket q * r \rrbracket\}}\ (\text{FP})$$

We now revisit the shortcomings of locality-breaking translations discussed in the preceding section and describe how they are addressed by locality-preserving translations.

Concurrency Recall that locality-breaking translations are not suitable for the refinement of concurrent libraries as they require non-trivial and non-modular linearisability proofs. As we demonstrate shortly, locality-preserving translations are more suitable in concurrent settings as the frame preservation property (FP) maintained by the translation ensures the preservation of all frames regardless of how the library operations may be interleaved with one another, eliminating the need for a linearisability proof.

In other words, rather than proving the refinement correctness for sequential programs and then establishing the correctness of concurrent programs via a linearisability proof, we can demonstrate the correct refinement of concurrent programs directly by establishing the refinement condition for the parallel composition rule. More concretely, equipped with the simpler proof obligation of (SimpRef) and the frame preservation property in

(FP), we can revisit the failed proof attempt for the parallel composition rule (Par) in the previous section and prove it as follows:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{}{\tau : \{p_1\}\ \mathtt{C}_1\ \{q_1\}}\ (\text{I.H.})}{\{\llbracket p_1 \rrbracket\}\ \llbracket \mathtt{C}_1 \rrbracket\ \{\llbracket q_1 \rrbracket\}}\ (\textsc{SimpRef})
\qquad
\dfrac{\dfrac{}{\tau : \{p_2\}\ \mathtt{C}_2\ \{q_2\}}\ (\text{I.H.})}{\{\llbracket p_2 \rrbracket\}\ \llbracket \mathtt{C}_2 \rrbracket\ \{\llbracket q_2 \rrbracket\}}\ (\textsc{SimpRef})
}{\{\llbracket p_1 \rrbracket * \llbracket p_2 \rrbracket\}\ \llbracket \mathtt{C}_1 \rrbracket \,||\, \llbracket \mathtt{C}_2 \rrbracket\ \{\llbracket q_1 \rrbracket * \llbracket q_2 \rrbracket\}}\ (\textsc{Par})
}{\{\llbracket p_1 * p_2 \rrbracket\}\ \llbracket \mathtt{C}_1 \rrbracket \,||\, \llbracket \mathtt{C}_2 \rrbracket\ \{\llbracket q_1 * q_2 \rrbracket\}}\ (\text{FP})
}{\{\llbracket p_1 * p_2 \rrbracket\}\ \llbracket \mathtt{C}_1 \,||\, \mathtt{C}_2 \rrbracket\ \{\llbracket q_1 * q_2 \rrbracket\}}\ (\llbracket . \rrbracket\ \text{def.})
}{\tau : \{p_1 * p_2\}\ \mathtt{C}_1 \,||\, \mathtt{C}_2\ \{q_1 * q_2\}}\ (\textsc{SimpRef})
$$

Although locality-preserving translations are more complex in that they require locality at the concrete level to match the locality at the abstract level, they are more suitable for the refinement of *concurrent* libraries. This is because locality-preserving translations preserve the correctness of concurrent client programs without the need for linearisability proofs. Moreover, unlike linearisability proofs, the refinement proofs of locality-preserving translations are modular and extending the library with additional operations does not affect the existing proofs.

Scalability Recall that when using locality-breaking translations, the number of proof obligations increases as we consider more complex libraries. This is because due to the footprint mismatch between the specification and implementation, we must provide a *completing* translation that extends incomplete data fragments, providing the additional resources needed. Consequently, we must consider all possible completions of incomplete data as the implementation behaviour may vary in each case. This in turn leads to several proof sketches per library axiom, hindering the scalability of locality-breaking translations.

To remedy this, locality-preserving translations abandon the notion of completing translations altogether, thus eliminating the need for considering the completions of incomplete data. In other words, locality-preserving translations yield only those resources that are required by the operation at the implementation level (rather than the complete data structure), whilst maintaining the the frame preservation property in (FP). More concretely, given an abstract triple $\{p\}$ $\mathtt{C}$ $\{q\}$, the translation of p ($\llbracket p \rrbracket$)

may produce all resources required by the implementation of `C`, provided that the frame preservation property in (FP) is maintained. That is, any frame r compatible with p and q remains compatible after the translation:

$$\forall r.\ \llbracket p * r \rrbracket = \llbracket p \rrbracket * \llbracket r \rrbracket \quad \text{and} \quad \llbracket q * r \rrbracket = \llbracket q \rrbracket * \llbracket r \rrbracket$$

However, as we demonstrated in the preceding section for the `x.add(n)` operation, the footprint of a library operation at the implementation level may be larger than its footprint at the specification level, potentially invalidating the frame preservation property. When this is the case, in order to maintain the frame preservation property, we appeal to an additional concept: the implementation *crust*. We then motivate the need for an additional piece of information required by locality-preserving translations, *interfaces*, facilitating the translation of incomplete abstract heaps.

Crust The *crust* of an implementation denotes those additional resources required by the library operations at the implementation level that are not contained in the footprint at the specification level. For instance, in the case of the list library implementation studied in the previous section (Fig. 6.1), the footprint of the `x.add(n)` implementation spans the entire list which is not included in its footprint at the abstract level. As such, the crust of this implementation encompasses the entire list.

In general, the crust of an implementation may be far more fine-grained than the entire data structure. For instance, consider the tree library $\mathbb{T}$ studied in §4 with its `appendChild(n,m)` axiom in Fig. 4.2 repeated on p. 204. When $\text{N}{=}n$ and $\text{M}{=}m$, the abstract footprint of `appendChild(n,m)` is limited to the tree node n and the subtree at m. Consider an implementation of the tree library where each node maintains a pointer to its parent and each parent node tracks its immediate children via a linked list. Let us assume that the implementation of `appendChild(n,m)` proceeds as follows: i) it looks up the parent of m, iterates over the child list of the parent until it reaches m, removes m from the child array; and ii) iterates over the child list of n until it reaches the end and then appends m to it. As such, the footprint of `appendChild(n,m)` at the implementation level contains the tree nodes n and m (which are included in the specification footprint) as well as the resources associated with the child list of m's parent and the child list of n (which are not included in the specification footprint). In other words, the crust of this implementation comprises the child list of m's parent and the child list of n. More concretely, for every abstract address $\mathbf{x}$ in the domain of an abstract tree heap (e.g. the addresses associated with α and γ in the heap described by the precondition of `appendChild`), the crust includes the child list of the node directly above $\mathbf{x}$ (e.g. the child list of the parent nodes of N and M). Analogously, for every context hole $\mathbf{x}$ in the range of an abstract tree heap (e.g. the context hole associated with β in the heap described by the precondition of `appendChild`), the crust includes the child list of the node directly above $\mathbf{x}$ (e.g. the child list of N).

Let us write $\mathsf{TCrust}(P)$ for the crust of the tree heap described by P and let us define the tree resources P_0, P_1 and P_2 as follows:

$$\begin{array}{ll} & \mathcal{R}_t \mapsto \mathrm{U}[\mathrm{L}[\varnothing] \otimes \mathrm{R}[\varnothing]] \Leftrightarrow \exists \alpha_1, \alpha_2.\ P_0 * P_1 * P_2 \\ \text{with} & P_0 \triangleq \mathcal{R}_t \mapsto \mathrm{U}[\alpha_1 \otimes \alpha_2] \qquad P_1 \triangleq \alpha_1 \mapsto \mathrm{L}[\varnothing] \qquad P_2 \triangleq \alpha_2 \mapsto \mathrm{R}[\varnothing] \end{array} \tag{6.4}$$

Since crust denotes the resources required by the implementation of the library operations, the translation function mapping abstract resources onto concrete ones must include the crust in the translation. For instance, the translation of P_1, namely $\llbracket P_1 \rrbracket$, must include the crust resource $\mathsf{TCrust}(P_1)$. That is, the translation of P_1 must be of the following form:

$$\llbracket P_1 \rrbracket \triangleq \cdots * \mathsf{TCrust}(P_1)$$

where $\mathsf{TCrust}(P_1)$ denotes the crust of P_1, namely the child list of U. Analogously, the $\llbracket P_2 \rrbracket$ must include $\mathsf{TCrust}(P_2)$: $\llbracket P_2 \rrbracket \triangleq \cdots * \mathsf{TCrust}(P_2)$, where $\mathsf{TCrust}(P_2)$ also corresponds the child list of U. Note that at the abstract level the resources described by P_1 and P_2 are compatible in that $P_1 * P_2$ is defined. In order to maintain the frame preservation property of (FP), we must ensure that the translations of these resources remain compatible at the concrete level. That is, we must show:

$$\begin{aligned} \llbracket P_1 * P_2 \rrbracket &= \llbracket P_1 \rrbracket * \llbracket P_2 \rrbracket \\ &= \mathsf{TCrust}(P_1) * \cdots * \mathsf{TCrust}(P_2) * \cdots \end{aligned}$$

However, observe that the resources of $\mathsf{TCrust}(P_1)$ and $\mathsf{TCrust}(P_2)$ are incompatible in that they both contain the child list of U, rendering $\mathsf{TCrust}(P_1) * \mathsf{TCrust}(P_2)$ undefined. To remedy this, we can declare the crust resources produced by the translation as *shared* resources which may be freely duplicated and are accessible by all threads. For instance, we can use similar techniques to the concurrent abstract predicates (CAP) logic [15] and redefine the translation above as follows:

$$\begin{aligned} \llbracket P_1 \rrbracket &\triangleq \cdots * \boxed{\mathsf{TCrust}(P_1)}_U \\ \llbracket P_2 \rrbracket &\triangleq \cdots * \boxed{\mathsf{TCrust}(P_2)}_U \\ \text{where} \quad \boxed{P}_U * \boxed{Q}_U &\Leftrightarrow \boxed{P \wedge Q}_U \qquad \text{for all } P, Q \text{ and } U \end{aligned}$$

The boxed assertion $\boxed{\mathsf{TCrust}(P_1)}_U$ states that $\mathsf{TCrust}(P_1)$ is a shared resource accessible by all threads and may be freely duplicated. The U describes how the shared resources of $\mathsf{TCrust}(P_1)$ may be updated by each thread, provided that it holds the sufficient permissions. As part of the translation, we can then provide each thread with the necessary permissions (ghost resources) to ensure that they can carry out the necessary changes. For instance, translating the precondition of the `appendChild(n,m)` operation may yield the permission required for appending node `m` to the end of the child list of `n`. Moreover, since boxed resources are shared, the may be combined using the $*$ connective as shown above. In particular, we have:

$$\boxed{\mathsf{TCrust}(P_1)}_U * \boxed{\mathsf{TCrust}(P_2)}_U \Leftrightarrow \boxed{\mathsf{TCrust}(P_1) \wedge \mathsf{TCrust}(P_2)}_U$$

By describing the crust as a shared resource, we maintain the frame preservation property of (FP). As discussed above, this liberates us from having to consider all possible completions and thus the need for multiple proof sketches per axiom. We omit the details of our locality-preserving translation for the tree library $\mathbb{T}$. We refer the reader instead to [23] where we present a concurrent implementation of the tree library $\mathbb{T}$ and establish its correctness by providing a locality-preserving translation.

Interfaces Recall that when defining a locality-breaking translation, incomplete heaps (containing abstract addresses) are not translated directly. Rather, incomplete heaps are first extended into complete ones and only then are they translated into concrete resources at the implementation level. As such, the translation function need not account for incomplete abstract heaps. On the other hand, as mentioned above, locality-preserving translations desert the concept of completion altogether and must thus accommodate the translation of incomplete heaps.

Observe that incomplete heaps are agnostic to the shapes of the data associated with the abstract addresses in their domain and range. For instance, given the P_0, P_1 and P_2 assertions in (6.4) above, let the values associated with the logical variables be as follows: $\alpha_1{=}\mathbf{x}_1$, $\alpha_2{=}\mathbf{x}_2$, $\text{U}{=}u$, $\text{L}{=}l$ and $\text{R}{=}r$. The P_0, P_1 and P_2 assertions then describe the tree heaps $\mathbf{h}_0 \triangleq \mathcal{R}_t \mapsto u[\mathbf{x}_1 \otimes \mathbf{x}_2]$, $\mathbf{h}_1 \triangleq \mathbf{x}_1 \mapsto l$, $\mathbf{h}_2 \triangleq \mathbf{x}_2 \mapsto r$, respectively. Observe that the incomplete heap $\mathbf{h}_0$ has no knowledge of the forests placed within $\mathbf{x}_1$ and $\mathbf{x}_2$, namely the shapes of $\mathbf{h}_1$ and $\mathbf{h}_2$. Analogously, the incomplete heaps $\mathbf{h}_1$ and $\mathbf{h}_2$ have no knowledge of the forest containing the $\mathbf{x}_1$ and $\mathbf{x}_2$ context holes, namely the shape of $\mathbf{h}_0$. This however may not the case at the implementation level. Consider the implementation described above where each node maintains a pointer to its parent and each parent node tracks its children via a linked list. The concrete representation of $\mathbf{h}_1$ (respectively $\mathbf{h}_2$) then includes a pointer from l (respectively r) to its parent (u in $\mathbf{h}_0$) and thus relies on some information from $\mathbf{h}_0$. Similarly, the concrete representation of $\mathbf{h}_0$ includes a linked list of the immediate children of u (namely l in $\mathbf{h}_1$ and r in $\mathbf{h}_2$) and hence relies on some information from $\mathbf{h}_1$ and $\mathbf{h}_2$. As such, when translating *incomplete* tree heaps with abstract addresses, we require auxiliary information describing how the abstract addresses and context holes connect together. We track this additional information associated with each abstract address $\mathbf{x}$ through an *interface* function, associating each abstract address with the necessary contextual information. To translate the incomplete heaps correctly, we thus parameterise the translation function with an interface function that records the interfaces associated with abstract addresses.

6.2.1. Locality-preserving Limitations: Complexity

We have briefly presented locality-preserving translations and demonstrated how they improve on locality-breaking translations for scalable refinement of both *sequential* and *concurrent* libraries. However, compared to locality-breaking translations, locality-preserving translations are more complex in that they require locality at the concrete level to match the locality at the abstract level, by requiring the frame preservation property (FP). In general, it may not be trivial to devise a translation function that maintains the (FP) property and achieves the desired locality parity.

More concretely, recall that in order to maintain the (FP) property, we must declare the crust as a shared resource $\boxed{\mathsf{TCrust}(\cdots)}_{U}$. The U denotes an interference relation describing how the shared resource $\mathsf{TCrust}(\cdots)$ may be manipulated by each thread, given a suitable distribution of permissions amongst threads. Describing the shared resources of crust and their interference may be rather involved, making locality-preserving translations more complex than their locality-breaking cousins. This complexity is justified in *concurrent* settings, especially when the library implementation employs fine-grained synchronisation mechanisms. This is because the complexity of locality-preserving translations is offset by the even more complex linearisability proofs required by locality-breaking translations. On the other hand, it is harder to justify this complexity in *sequential* settings where the rival locality-breaking translations offer simpler translations, albeit unscalable.

In what follows, we present a *hybrid* translation approach for the refinement of libraries in *sequential* settings, combining the strengths of the locality-breaking and locality-preserving approaches.

6.3. Hybrid Translations

We have briefly demonstrated how to refine abstract libraries as an alternative justification for the soundness of SSL specifications. As with previous work [16, 59], we have reported on two approaches for proving the refinement correctness of an implementation with respect to an abstract specification: locality-breaking and locality-preserving translations.

The main difference in the two approaches is the burden of the proof of a correct translation. Locality-breaking translations are simple in that they do not require locality at the concrete level to match the locality at the abstract level. However, when refining the libraries of complex data, the number of proof obligations increases rapidly as we must provide several proof sketches per library axiom. Moreover, they are not suitable for the refinement of concurrent modules: the burden of proof significantly increases as one must additionally establish that each operation implementation is linearisable. On the other hand, locality-preserving translations allow us to refine libraries of complex data by providing one proof sketch per library axiom. Moreover, they are more suitable for the refinement of *concurrent* libraries since the (FP) property significantly simplifies the proof obligation by eliminating the need for non-modular linearisability proofs. However, they are more complex in that they require locality at the concrete level to match the locality at the abstract level, and it may be non-trivial to devise a translation function that achieves this.

We present a *hybrid* translation approach for the refinement of libraries in *sequential* settings, borrowing ideas from both locality-breaking and locality-preserving translations. More concretely, as with the locality-preserving approach, we do not provide a completing translation and instead provide each operation with its required resources by identifying the implementation *crust*. This then enables us to verify the correctness of each operation with a single proof sketch per axiom, allowing for scalable proofs. However, unlike locality-preserving translations, we do not declare the crust as a shared (duplicable) resource. Rather, we treat the crust as an extension of the footprint, much like the additional resources yielded by the *completions* of the locality-breaking translations. This of course invalidates the frame preservation property (FP) required by locality-preserving translations. However, whilst the (FP) property is crucial for establishing the correctness of *concurrent* implementations (it eliminates the need for non-trivial linearisability proofs), it is not of import in sequential settings. As such, since we only consider sequential library implementations, losing the (FP) property is of no consequence.

In the following chapter, we present a sequential JavaScript implementation of the DOM library specified in §5 and establish its correctness by providing a hybrid translation as described above. That is, we identify the crust of our implementation and subsequently define a translation function that maps abstract DOM heaps onto corresponding concrete JavaScript heaps extended with the necessary crust resources. As we demonstrate in §7.2, our implementation crust comprises DOM tree resources analogous to that explained above (e.g. the child list of a parent node), as well as additional resources needed for the correct initialisation of our JavaScript implementation.

7. Refinement for $\mathbb{DOM}$

Previously in §5, we presented a formal specification of a fragment of the DOM library [2]. In keeping with the axiomatic style of the DOM standard [1], we specified the behaviour of DOM operations axiomatically rather than operationally. In this chapter, we further justify our axiomatic DOM specification with respect to a reference implementation in JavaScript [44]. In §7.1 we present the JavaScript implementation of our DOM fragment. Later in §7.2 we establish the correctness of this implementation with respect to its specification in §5, by using a *hybrid* translation as described briefly in the previous chapter.

7.1. A DOM Implementation in JavaScript

Recall from §5 that the DOM API is specified in an object-oriented fashion. However, unlike standard object-oriented languages (e.g. JAVA), JavaScript does not have classes. It is thus not possible to statically specify that an object created with a given constructor is of a certain *type*, exposing certain fields and methods. Instead, in JavaScript every object has a *prototype* to which it dynamically delegates the *requests* it cannot handle. For instance, when running `o.f()`, if `o` does not define the method `f`, the interpreter will check if its prototype does, in which case method `f` of `o`'s prototype is executed. As such, method sharing in JavaScript is accomplished via prototype-based inheritance described above. We thus implement the DOM data types (namely the Node, Element, Text, Document, Attribute and NodeList interfaces) as JavaScript prototype objects and their operations as JavaScript functions. For instance, for each node type (e.g. element nodes) we define a prototype that defines the methods corresponding to the operations exposed by the nodes of that type (e.g. `getAttribute`). We proceed with a high-level description of our DOM implementation in [44].

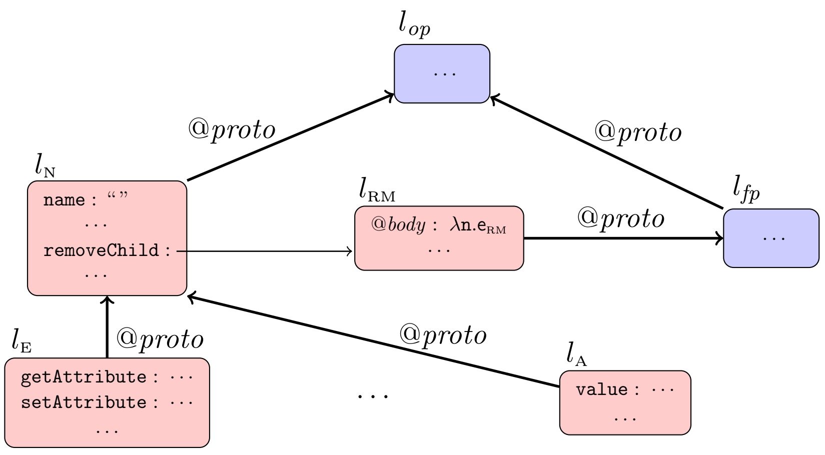


Figure 7.1.: A partial representation of our DOM implementation

DOM Interfaces as JavaScript Prototypes

We implement four special JavaScript prototypes, `DProto`, `EProto`, `TProto` and `AProto` to represent the interfaces of Document, Element, Text, and Attribute nodes, respectively. Each one of these prototypes defines the methods provided by the nodes of its corresponding type. For instance, `EProto` defines the element-specific methods such as `getAttribute` and `setAttribute`. Recall that these specialised interfaces all implement the Node interface which provides the common node methods such as `appendChild` and `removeChild`. We thus implement a JavaScript prototype, `NProto`, that represents the DOM Node interface and implements the common node methods. The `NProto` serves as the prototype of the four specialised node prototypes above.

For the NodeList interface, we implement two prototypes, `FLProto` and `TLProto`, to represent the forest listener and tag listener NodeLists, respectively. Each one of these prototypes defines the methods exposed by the NodeList interface, namely the `length` and `item` methods. Lastly, we implement a common prototype to these two objects, `NLProto`.

Each of these JavaScript prototype objects describe JavaScript heaps (Def. 72). A partial pictorial representation of these prototype objects in the heap is given in Fig. 7.1. The formal heap representations of these objects are given in Figs. 7.2-7.3. We first describe the notation used in Figs. 7.2-7.3 and then proceed with a description of the heap representations.

Notation Given the JavaScript monoid $(\textsc{JSLHeap}, \circ, \textsc{JSUnit})$ of Def. 72, we adopt the following assertion-like shorthand to increase readability, where $p, q \in \mathcal{P}(\textsc{JSLHeap})$, X, Y are values, V is a carrier set and $\mathcal{X}$ denotes the set of all valid JavaScript field names (Def. 68):

$$p \wedge q \ \text{ for } \ p \cap q \qquad p \vee q \ \text{ for } \ p \cup q \qquad \mathsf{true} \ \text{ for } \ \{h \mid h \in \textsc{JSLHeap}\}$$

$$\mathsf{emp} \ \text{ for } \ \{h \mid h \in \textsc{JSUnit}\} \qquad p * q \ \text{ for } \ \{h_1 \circ h_2 \mid h_1 \in p \wedge h_2 \in q\}$$

$$X \doteq Y \ \text{ for } \ \begin{cases} \mathsf{emp} & \text{if } X = Y \\ \emptyset & \text{otherwise} \end{cases} \qquad \exists v \in V.\, p \ \text{ for } \ \{h \mid v \in V \wedge h \in p\}$$

$$\underset{v \in \{v_1, \dots, v_n\}}{\circledast} p(v) \quad \text{for} \quad \{h \mid h \in p(v_1)\} * \cdots * \{h \mid h \in p(v_n)\}$$

$$\mathsf{only}\,(l, S) \quad \text{for} \quad \underset{\mathtt{f} \in \mathcal{X} \setminus S}{\circledast} (l, \mathtt{f}) \mapsto \varnothing$$

$$l \mapsto\!\!\!\!\Rightarrow \{\mathtt{f}_1 : v_1 \dots \mathtt{f}_n : v_n\} \quad \text{for} \quad l \mapsto \{\mathtt{f}_1 : v_1 \dots \mathtt{f}_n : v_n\} * \mathsf{only}\,(l, \{\mathtt{f}_1 \dots \mathtt{f}_n\})$$

The $\mathsf{only}\,(l, S)$ shorthand states that the field names on the object at location l are limited to those in the set S and thus for any other field name the value is $\varnothing$ (none). For brevity, we write h for the singleton set $\{h\}$ (e.g. $(l, \mathtt{f}) \mapsto \varnothing$ for $\{(l, \mathtt{f}) \mapsto \varnothing\}$ in the definition of $\mathsf{only}\,(l, S)$). The $l \mapsto\!\!\!\!\Rightarrow \{\mathtt{f}_1 : v_1 \dots \mathtt{f}_n : v_n\}$ shorthand states that the object at location l *only* contains the fields $\mathtt{f}_1 \dots \mathtt{f}_n$ with values $v_1 \dots v_n$, respectively.

Let us turn our focus to the JavaScript heap representation of our implementation in Figs. 7.2-7.3. Let l_{N} denote the location of the node prototype object `NProto` in the JavaScript heap. To avoid cluttering the definitions with additional arguments, we assume l_{N} to be an implicit argument to the definitions in Fig. 7.2 when relevant. For instance, the definitions of `NProto` and `DProto` in Fig. 7.2 both refer to l_{N} on the right hand side (via $l_{\text{N}} \mapsto\!\!\!\!\Rightarrow \dots$ and $@\mathit{proto} : l_{\text{N}}$, respectively) and we thus assume l_{N} to be an implicit argument to both definitions. Similarly, we assume l_{D}, l_{E}, l_{T} and l_{A} to denote the locations of `DProto`, `EProto`, `TProto` and `AProto`, respectively, and treat them as implicit arguments in the definitions of Fig. 7.2, as described above. As we demonstrate later in Def. 79, we existentially quantify these locations outside the definition of our JS

$$
\begin{array}{l}
\texttt{NProto} \triangleq \exists l_{\text{HC}}, l_{\text{FC}}, l_{\text{LC}}, l_{\text{PS}}, l_{\text{NS}}, l_{\text{IB}}, l_{\text{RP}}, l_{\text{RM}}, l_{\text{AC}}. \\
\quad l_{\text{N}} \mapsto \left\{ \begin{array}{l}
@\mathit{proto} : l_{op}, \\
\text{ELEMENT\_NODE} : 1, \text{ATTRIBUTE\_NODE} : 2, \\
\text{TEXT\_NODE} : 3, \text{DOCUMENT\_NODE} : 9, \\
\texttt{nodeName} : \text{“ ”}, \texttt{nodeValue} : \texttt{null}, \texttt{nodeType} : \texttt{null}, \\
\texttt{parentNode} : \texttt{null}, \texttt{childNodes} : \texttt{null}, \\
\texttt{ownerDocument} : \texttt{null}, \texttt{hasChildNodes} : l_{\text{HC}}, \\
\texttt{firstChild} : l_{\text{FC}}, \texttt{lastChild} : l_{\text{LC}}, \\
\texttt{previousSibling} : l_{\text{PS}}, \texttt{nextSibling} : l_{\text{NS}}, \\
\texttt{insertBefore} : l_{\text{IB}}, \texttt{replaceChild} : l_{\text{RP}}, \\
\texttt{removeChild} : l_{\text{RM}}, \texttt{appendChild} : l_{\text{AC}}
\end{array} \right\} \\
\quad * \mathsf{func}(l_{\text{HC}}, \mathbf{l}, [\,], \mathsf{e}_{\text{HC}}) * \mathsf{func}(l_{\text{FC}}, \mathbf{l}, [\,], \mathsf{e}_{\text{FC}}) * \mathsf{func}(l_{\text{LC}}, \mathbf{l}, [\,], \mathsf{e}_{\text{LC}}) \\
\quad * \mathsf{func}(l_{\text{PS}}, \mathbf{l}, [\,], \mathsf{e}_{\text{PS}}) * \mathsf{func}(l_{\text{NS}}, \mathbf{l}, [\,], \mathsf{e}_{\text{NS}}) * \mathsf{func}(l_{\text{IB}}, \mathbf{l}, [\mathsf{n}, \mathsf{o}], \mathsf{e}_{\text{IB}}) \\
\quad * \mathsf{func}(l_{\text{RP}}, \mathbf{l}, [\mathsf{n}, \mathsf{o}], \mathsf{e}_{\text{RP}}) * \mathsf{func}(l_{\text{RM}}, \mathbf{l}, [\mathsf{n}], \mathsf{e}_{\text{RM}}) * \mathsf{func}(l_{\text{AC}}, \mathbf{l}, [\mathsf{n}], \mathsf{e}_{\text{AC}})
\end{array}
$$

$$
\begin{array}{l}
\texttt{DProto} \triangleq \exists l_{\text{CE}}, l_{\text{CT}}, l_{\text{CA}}, l_{\text{DTN}}. \\
\quad l_{\text{D}} \mapsto \left\{ \begin{array}{l}
@\mathit{proto} : l_{\text{N}}, \texttt{nodeType} : l_{\text{N}}.\text{DOCUMENT\_NODE}, \\
\texttt{nodeName} : \text{“\#document”}, \texttt{createElement} : l_{\text{CE}}, \\
\texttt{createTextNode} : l_{\text{CT}}, \texttt{createAttribute} : l_{\text{CA}}, \\
\texttt{getElementsByTagName} : l_{\text{DTN}}
\end{array} \right\} \\
\quad * \mathsf{func}(l_{\text{CE}}, \mathbf{l}, [\mathsf{s}], \mathsf{e}_{\text{CE}}) * \mathsf{func}(l_{\text{CT}}, \mathbf{l}, [\mathsf{s}], \mathsf{e}_{\text{CT}}) \\
\quad * \mathsf{func}(l_{\text{CA}}, \mathbf{l}, [\mathsf{s}], \mathsf{e}_{\text{CA}}) * \mathsf{func}(l_{\text{DTN}}, \mathbf{l}, [\mathsf{s}], \mathsf{e}_{\text{DTN}})
\end{array}
$$

$$
\begin{array}{l}
\texttt{EProto} \triangleq \exists l_{\text{GA}}, l_{\text{SA}}, l_{\text{RA}}, l_{\text{GAN}}, l_{\text{SAN}}, l_{\text{RAN}}, l_{\text{ETN}}. \\
\quad l_{\text{E}} \mapsto \left\{ \begin{array}{l}
@\mathit{proto} : l_{\text{N}}, \texttt{nodeType} : l_{\text{N}}.\text{ELEMENT\_NODE}, \\
\texttt{getAttribute} : l_{\text{GA}}, \texttt{setAttribute} : l_{\text{SA}}, \\
\texttt{removeAttribute} : l_{\text{RA}}, \texttt{getAttributeNode} : l_{\text{GAN}}, \\
\texttt{setAttributeNode} : l_{\text{SAN}}, \texttt{removeAttributeNode} : l_{\text{RAN}}, \\
\texttt{getElementsByTagName} : l_{\text{ETN}}, \texttt{\_\_attributes\_\_} : \texttt{null}
\end{array} \right\} \\
\quad * \mathsf{func}(l_{\text{GA}}, \mathbf{l}, [\mathsf{s}], \mathsf{e}_{\text{GA}}) * \mathsf{func}(l_{\text{SA}}, \mathbf{l}, [\mathsf{s}_1, \mathsf{s}_2], \mathsf{e}_{\text{SA}}) * \mathsf{func}(l_{\text{RA}}, \mathbf{l}, [\mathsf{s}], \mathsf{e}_{\text{RA}}) \\
\quad * \mathsf{func}(l_{\text{GAN}}, \mathbf{l}, [\mathsf{s}], \mathsf{e}_{\text{GAN}}) * \mathsf{func}(l_{\text{SAN}}, \mathbf{l}, [\mathsf{a}], \mathsf{e}_{\text{SAN}}) \\
\quad * \mathsf{func}(l_{\text{RAN}}, \mathbf{l}, [\mathsf{a}], \mathsf{e}_{\text{RAN}}) * \mathsf{func}(l_{\text{ETN}}, \mathbf{l}, [\mathsf{s}], \mathsf{e}_{\text{ETN}})
\end{array}
$$

$$
\begin{array}{l}
\texttt{TProto} \triangleq \exists l_{\text{DLEN}}, l_{\text{ST}}, l_{\text{SD}}, l_{\text{AD}}, l_{\text{ID}}, l_{\text{DD}}, l_{\text{RD}}. \\
\quad l_{\text{T}} \mapsto \left\{ \begin{array}{l}
@\mathit{proto} : l_{\text{N}}, \texttt{nodeType} : l_{\text{N}}.\text{TEXT\_NODE}, \\
\texttt{nodeName} : \text{“\#text”}, \texttt{data} : \texttt{null}, \texttt{length} : l_{\text{DLEN}} \\
\texttt{splitText} : l_{\text{ST}}, \texttt{substringData} : l_{\text{SD}}, \texttt{appendData} : l_{\text{AD}}, \\
\texttt{insertData} : l_{\text{ID}}, \texttt{deleteData} : l_{\text{DD}}, \texttt{replaceData} : l_{\text{RD}}
\end{array} \right\} \\
\quad * \mathsf{func}(l_{\text{DLEN}}, \mathbf{l}, [\,], \mathsf{e}_{\text{DLEN}}) * \mathsf{func}(l_{\text{ST}}, \mathbf{l}, [\mathsf{o}], \mathsf{e}_{\text{ST}}) * \mathsf{func}(l_{\text{SD}}, \mathbf{l}, [\mathsf{o}, \mathsf{c}], \mathsf{e}_{\text{SD}}) \\
\quad * \mathsf{func}(l_{\text{AD}}, \mathbf{l}, [\mathsf{s}], \mathsf{e}_{\text{AD}}) * \mathsf{func}(l_{\text{ID}}, \mathbf{l}, [\mathsf{o}, \mathsf{s}], \mathsf{e}_{\text{ID}}) \\
\quad * \mathsf{func}(l_{\text{DD}}, \mathbf{l}, [\mathsf{o}, \mathsf{c}], \mathsf{e}_{\text{DD}}) * \mathsf{func}(l_{\text{RD}}, \mathbf{l}, [\mathsf{o}, \mathsf{c}, \mathsf{s}], \mathsf{e}_{\text{RD}})
\end{array}
$$

$$
\texttt{AProto} \triangleq l_{\text{A}} \mapsto \left\{ \begin{array}{l}
@\mathit{proto} : l_{\text{N}}, \texttt{nodeType} : l_{\text{N}}.\text{ATTRIBUTE\_NODE} \\
\texttt{name} : \texttt{null}, \texttt{value} : l_{\text{V}}, \texttt{nodeValue} : l_{\text{V}}
\end{array} \right\} * \mathsf{func}(l_{\text{V}}, \mathbf{l}, [\,], \mathsf{e}_{\text{V}})
$$

where $\mathsf{func}(l, L, \bar{\mathsf{x}}, \mathsf{e}) \triangleq l \mapsto \{@\mathit{proto} : l_{fp}, @\mathit{scope} : L, @\mathit{body} : \lambda \bar{\mathsf{x}}.\mathsf{e}\}$

Figure 7.2.: DOM node interfaces as JavaScript prototypes

$$\texttt{NLProto} \triangleq l_{\text{NL}} \mapsto \{@\mathit{proto} : l_{op}, \texttt{length} : \texttt{null}, \texttt{item} : \texttt{null}\}$$

$$\begin{array}{l}\texttt{FLProto} \triangleq \exists l_{\text{LEN}}, l_{\text{ITEM}}, l_{\text{fc}}, l_{\text{lc}}, l_{\text{ps}}, l_{\text{ns}}, l_{\text{ib}}, l_{\text{rp}}, l_{\text{rm}}, l_{\text{ac}}. \\
\quad l_{\text{FL}} \mapsto \left\{\begin{array}{l} @\mathit{proto} : l_{\text{NL}}, \texttt{length} : l_{\text{LEN}}, \texttt{item} : l_{\text{ITEM}}, \\ \texttt{\_\_contents\_\_} : \texttt{null}, \texttt{\_\_thisNode\_\_} : \texttt{null}, \\ \texttt{\_\_firstChild\_\_} : l_{\text{fc}}, \texttt{\_\_lastChild\_\_} : l_{\text{lc}}, \\ \texttt{\_\_previousSibling\_\_} : l_{\text{ps}}, \texttt{\_\_nextSibling\_\_} : l_{\text{ns}}, \\ \texttt{\_\_insertBefore\_\_} : l_{\text{ib}}, \texttt{\_\_replaceChild\_\_} : l_{\text{rp}}, \\ \texttt{\_\_removeChild\_\_} : l_{\text{rm}}, \texttt{\_\_appendChild\_\_} : l_{\text{ac}} \end{array}\right\} \\
\quad * \mathsf{func}(l_{\text{LEN}}, \mathbf{l}, [\,], \mathsf{e}_{\text{LEN}}) * \mathsf{func}(l_{\text{ITEM}}, \mathbf{l}, [\mathtt{i}], \mathsf{e}_{\text{ITEM}}) * \mathsf{func}(l_{\text{fc}}, \mathbf{l}, [\,], \mathsf{e}_{\text{fc}}) \\
\quad * \mathsf{func}(l_{\text{lc}}, \mathbf{l}, [\,], \mathsf{e}_{\text{lc}}) * \mathsf{func}(l_{\text{ps}}, \mathbf{l}, [\,], \mathsf{e}_{\text{ps}}) * \mathsf{func}(l_{\text{ns}}, \mathbf{l}, [\,], \mathsf{e}_{\text{ns}}) \\
\quad * \mathsf{func}(l_{\text{ib}}, \mathbf{l}, [\mathtt{n}, \mathtt{o}], \mathsf{e}_{\text{ib}}) * \mathsf{func}(l_{\text{rp}}, \mathbf{l}, [\mathtt{n}, \mathtt{o}], \mathsf{e}_{\text{rp}}) \\
\quad * \mathsf{func}(l_{\text{rm}}, \mathbf{l}, [\mathtt{n}], \mathsf{e}_{\text{rm}}) * \mathsf{func}(l_{\text{ac}}, \mathbf{l}, [\mathtt{n}], \mathsf{e}_{\text{ac}})\end{array}$$

$$\begin{array}{l}\texttt{TLProto} \triangleq \exists l'_{\text{LEN}}, l'_{\text{ITEM}}. \\
\quad l_{\text{TL}} \mapsto \left\{\begin{array}{l} @\mathit{proto} : l_{\text{NL}}, \texttt{length} : l'_{\text{LEN}}, \texttt{item} : l'_{\text{ITEM}}, \\ \texttt{\_\_startPoint\_\_} : \texttt{null}, \texttt{\_\_pattern\_\_} : \texttt{null}, \end{array}\right\} \\
\quad * \mathsf{func}(l'_{\text{LEN}}, \mathbf{l}, [\,], \mathsf{e}'_{\text{LEN}}) * \mathsf{func}(l'_{\text{ITEM}}, \mathbf{l}, [\mathtt{i}], \mathsf{e}'_{\text{ITEM}})\end{array}$$

Figure 7.3.: The DOM NodeList interfaces as JavaScript prototypes

heap representation. By treating these existentially quantified variables as implicit arguments when relevant, we keep our definitions cleaner and improve readability.

A partial JavaScript representation of our implementation is depicted in Fig. 7.1, where the objects at locations l_{N}, l_{E} and l_{A} respectively denote the `NProto`, `EProto` and `AProto` prototypes, while the representation of other prototype objects are left out for brevity.

The `NProto` describes the Node prototype object at location l_{N} and states that the prototype of l_{N} (from which it inherits) is stored at address l_{op} (via the $@\mathit{proto}$ field). Recall that the l_{op} denotes the designated JavaScript *object prototype*.[1] The object prototype at l_{op} describes the universal object at the end of all JavaScript prototype chains and is akin to the Java universal superclass "Object". The ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE and DOCUMENT_NODE denote

[1] The $@\mathit{proto}$ field denotes an internal JavaScript field used to determine the prototype chain of an object dynamically. Given an object at location l, the $l.@\mathit{proto}$ records the location of l's prototype; that is,, i.e. the object from which l inherits. To distinguish internal fields from regular ones, they are written in *italics* prefixed with the @ symbol, whereas regular fields are written in the `typewriter` style.

integer constants describing node types. The `NProto` prototype describes an abstract prototype (similar to an abstract class in Java that may not be instantiated) from which all other DOM node prototypes inherit. As such, the values of the fields `nodeValue`, `nodeType`, `parentNode`, `childNodes` and `ownerDocument` are all `null`. Analogously, the value of the `nodeName` field is the empty string “ ”. The remainder of the `NProto` definition describes the DOM methods exposed by the Node interface as JavaScript functions. A JavaScript function is stored in the heap as a *function object* with the function body stored as a lambda abstraction (Def. 67). For instance, the `insertBefore` function is stored as a function object at location l_{IB}. The $\mathsf{func}(l_{\text{IB}}, \mathbf{l}, [\mathtt{n}, \mathtt{o}], \mathsf{e}_{\text{IB}})$ predicate (defined at the bottom of Fig. 7.2) states that the function object at l_{IB} inherits from the designated JavaScript *function prototype* at l_{fp} (via the internal field @*proto*), that this function is to be executed in the current scope chain $\mathbf{l}$ (Def. 74), that this method has two arguments (`n` and `o`), and that the method body is given by the JavaScript expression e_{IB}. The e_{IB} describes the code of the `insertBefore` function as implemented in [44]. We have omitted the code here for brevity. The implementation of the `r := n.firstChild` and `n.removeChild(o)` and operations are given in §B. The implementation of the remaining operations can be found in [44].

The remaining definitions describe the JavaScript heap representations of the specialised DOM node prototypes. Note that each specialised prototype inherits from the `NProto` prototype at l_{N} (via the @*proto* field). As such, each specialised node prototype only implements the methods specific to that node (e.g. `getAttribute` on the element prototype `EProto`) and does not implement the common node methods (e.g. `removeChild`) as these are inherited from the Node prototype `NProto`. The `__attributes__` field on `EProto` tracks the attributes associated with the element node as an object with one correspondingly-named field per attribute. The `__attributes__` field is not part of the DOM Element interface and is a feature of our element node implementation. Another implementation may choose to record the element attributes as e.g. an array or a linked list under another name. As a convention, to distinguish the implementation-specific fields from those provided by the DOM interface, we prefix and suffix their names with `__`.

The NodeList prototypes are given in Fig. 7.3. As before, we assume

l_{NL}, l_{FL} and l_{TL} to denote the locations of `NLProto`, `FLProto` and `TLProto`, respectively. We then assume l_{NL}, l_{FL} and l_{TL} to be implicit arguments to the definitions in Fig. 7.3, when relevant. As mentioned earlier, the `NLProto` serves as a common prototype for the forest listener NodeList prototype (`FLProto`) and the tag listener NodeList prototype (`TLProto`). Recall that the NodeList interface provides two methods: `length` and `item`. The implementations of `length` and `item` for forest listeners differ from those of tag listeners. Forest listener NodeLists are maintained *eagerly*: upon each insertion and removal from the NodeList the associated contents array (`__contents__`) is updated accordingly. By contrast, tag listener NodeLists are maintained *lazily*: the contents of the NodeList are computed on demand upon each inspection via the `length` and `item` methods. As such, each of the `FLProto` and `TLProto` prototypes define their own implementations rather than inheriting them from the common `NLProto` prototype. Therefore, the `NLProto` prototype provides no implementation of these methods (declared simply as `null`).

The `FLProto` describes the forest listener NodeList prototype at address l_{FL}, inheriting from the `NLProto` at address l_{NL} (via the @*proto* field). The `length` and `item` fields record the locations of the function objects implementing the `length` and `item` methods for forest listeners, respectively. The remaining fields track implementation-specific values that are not part of the DOM interface. The `__contents__` tracks the contents of the child list as a JavaScript array. The `__thisNode__` field tracks the location of the node object with which this forest listener NodeList is associated. The remaining fields describe auxiliary functions used in the implementations of the correspondingly-named functions in the Node prototype `NProto`. For instance, the implementation of the `removeChild` function in `NProto` removes the specified node from the child list by calling `__removeChild__` on the child list.

Analogously, the `TLProto` describes the tag listener NodeList prototype at address l_{TL}, inheriting from the `NLProto` at address l_{NL}. The `length` and `item` fields record the locations of the function objects implementing the `length` and `item` methods for tag listeners, respectively. The `__startPoint__` records the location of the node object with which this tag listener is associated, i.e. the starting point for the tag search. The `__pattern__` field records the search string for this tag listener.

We can now define the JavaScript heap representation of our implementation as a collection of the prototype objects defined in Figs. 7.2-7.3. As mentioned earlier, we existentially quantify the implicit arguments used in the definitions of Figs. 7.2-7.3 denoting the locations of the prototype objects (e.g. l_{N}, l_{D}, etc.).

Definition 79 (DOM prototypes)**.** The *JavaScript heap representation of the DOM interfaces* is defined as follows:

$$\begin{aligned}\texttt{Protos} \triangleq \exists l_{\text{N}}, l_{\text{D}}, l_{\text{E}}, l_{\text{T}}, l_{\text{A}}, l_{\text{NL}}, l_{\text{FL}}, l_{\text{TL}}.\ & \\ & \texttt{NProto} * \texttt{DProto} * \texttt{EProto} * \texttt{TProto} * \texttt{AProto} \\ & * \texttt{NLProto} * \texttt{FLProto} * \texttt{TLProto}\end{aligned}$$

DOM Objects as JavaScript Objects

A DOM object in our JavaScript implementation is an instance of the corresponding prototype. For instance, a partial JavaScript representation of the "img" element with identifier 3 in Fig. 5.1a is depicted in Fig. 7.4. The objects above the dashed line represent prototype objects given in Figs. 7.2-7.3, as described in the previous section.

Let us look at the JavaScript heap representation of the "img" element with identifier 3 in more detail. This element node is represented in the JavaScript heap as:

$$\begin{aligned}&\mathsf{ENode}\,(3, \text{“img”}, l_a, \mathit{fid}, 9) * \mathsf{FL}\,(\mathit{fid}, 3, [\,]) * l_a \mapsto \left\{\begin{matrix}@\mathit{proto} : l_{op}, \\ \texttt{src}\!:\!13, \texttt{width}\!:\!17\end{matrix}\right\} \\ &* \mathsf{ANode}\,(13, \text{“src”}, \mathit{fid}_1) * \mathsf{FL}\,(\mathit{fid}_1, 13, [1]) \\ &* \mathsf{ANode}\,(17, \text{“width”}, \mathit{fid}_2) * \mathsf{FL}\,(\mathit{fid}_2, 17, [23]) \\ &* \mathsf{TNode}\,(1, \text{“goo.gl/K4S0d0”}, \mathit{fid}_3, 13) * \mathsf{FL}\,(\mathit{fid}_3, 1, [\,]) \\ &* \mathsf{TNode}\,(23, \text{“800px”}, \mathit{fid}_4, 13) * \mathsf{FL}\,(\mathit{fid}_4, 23, [\,])\end{aligned} \tag{7.1}$$

with the definitions of ENode, ANode, TNode and FL given in Fig. 7.5. As before, we assume l_{N}, l_{D}, l_{E}, l_{T}, l_{A}, l_{NL}, l_{FL} and l_{TL} to denote the locations of `NProto`, `DProto`, `EProto`, `TProto`, `AProto`, `NLProto`, `FLProto` and `TLProto`, respectively, and treat them as implicit arguments in the definitions of Fig. 7.5, when relevant.

The first component, $\mathsf{ENode}\,(3, \text{“img”}, l_a, \mathit{fid}, 9)$, implements the "img" element with identifier 3 as an instance of the `EProto` prototype at address

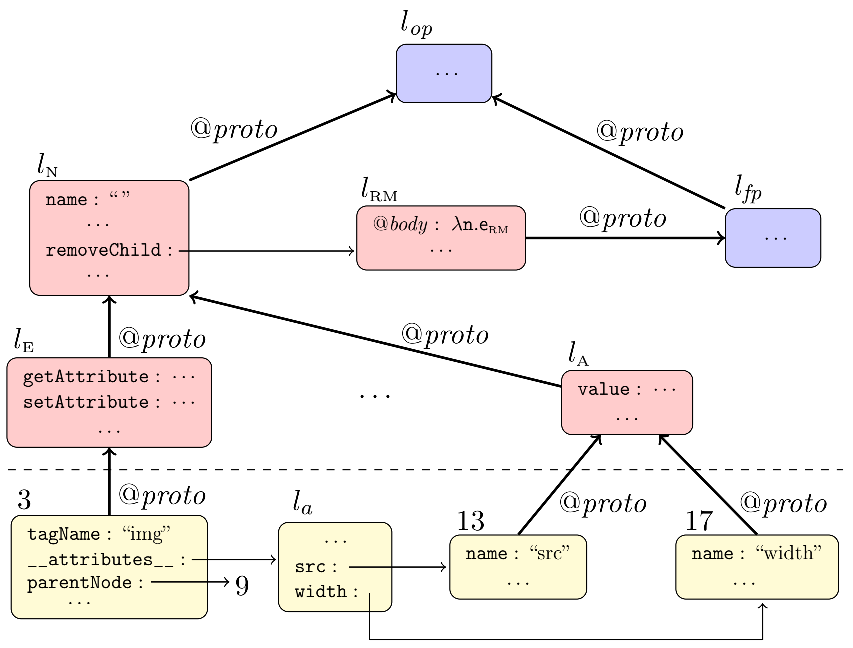


Figure 7.4.: A partial representation of "img" element 3 from Fig. 5.1a

3 with fields: i) @*proto*, storing the address of the `EProto` prototype l_{E} (captured by the arrow labelled @*proto* in Fig. 7.4); ii) `tagName`, storing its tag name ("img"); iii) `nodeName`, the same as the tag name; iv) `__attributes__`, storing the address of an object (l_a) that maps the names of the attributes on the node to their corresponding attribute objects; v) `childNodes`, storing the address of a `FLProto` object containing the child nodes of the node; vi) `parentNode`, storing the address of the parent node (i.e. node 9); and vii) `ownerDocument`, storing the address of the document object (with the designated identifier d). An element node has no other fields and inherits all methods of the `EProto` prototype (e.g. `getAttribute`).

The second component, $\mathsf{FL}(\mathit{fid}, 3, [\,])$, implements the forest listeners of node 3 as a single `FLProto` object at address *fid* with fields: i) @*proto*, storing the address of the `FLProto` prototype l_{FL}; ii) `__thisNode__`, storing the address of the node with which the child list is associated, i.e. 3; and iii) `__contents__`, storing the location of the contents array. The contents (in this case $[\,]$) are represented as a JavaScript array captured by the

$$\mathsf{DNode}\,(l, \mathit{fid}, e) \triangleq l \mapsto \begin{Bmatrix} @\mathit{proto} : l_{\text{D}}, \texttt{childNodes} : \mathit{fid}, \\ \texttt{documentElement} : e \end{Bmatrix}$$

$$\mathsf{ENode}\,(l, \mathrm{s}, l_a, \mathit{fid}, u) \triangleq l \mapsto \begin{Bmatrix} @\mathit{proto} : l_{\text{E}}, \texttt{nodeName} : \mathrm{s}, \texttt{tagName} : \mathrm{s}, \\ \texttt{\_\_attributes\_\_} : l_a, \texttt{childNodes} : \mathit{fid}, \\ \texttt{parentNode} : u, \texttt{ownerDocument} : d \end{Bmatrix}$$

$$\mathsf{TNode}\,(l, \mathrm{s}, \mathit{fid}, u) \triangleq l \mapsto \begin{Bmatrix} @\mathit{proto} : l_{\text{T}}, \texttt{nodeValue} : \mathrm{s}, \texttt{childNodes} : \mathit{fid}, \\ \texttt{parentNode} : u, \texttt{ownerDocument} : d \end{Bmatrix}$$

$$\mathsf{ANode}\,(l, \mathrm{s}, \mathit{fid}) \triangleq l \mapsto \begin{Bmatrix} @\mathit{proto} : l_{\text{A}}, \texttt{nodeName} : \mathrm{s}, \texttt{name} : \mathrm{s}, \\ \texttt{childNodes} : \mathit{fid}, \texttt{ownerDocument} : d \end{Bmatrix}$$

$$\mathsf{FL}\,(l, n, L) \triangleq \exists l_c.\ l \mapsto \begin{Bmatrix} @\mathit{proto} : l_{\text{FL}}, \texttt{\_\_thisNode\_\_} : n, \\ \texttt{\_\_contents\_\_} : l_c \end{Bmatrix} * \mathsf{array}(l_c, L)$$

$$\mathsf{TL}\,(l, \mathrm{s}, n) \triangleq l \mapsto \{@\mathit{proto} : l_{\text{TL}}, \texttt{\_\_pattern\_\_} : \mathrm{s}, \texttt{\_\_startPoint\_\_} : n\}$$

$$\mathsf{TLs}\,(n, ts) \triangleq \mathop{\circledast}_{(\mathrm{s}, \mathit{fid}) \in ts} \mathsf{TL}\,(\mathit{fid}, \mathrm{s}, n)$$

Figure 7.5.: DOM Node and NodeList instances as JavaScript objects

abstract predicate array. A child list has no other fields and inherits all methods of the `FLProto` object for manipulating the child list (e.g. `length`, `item`).

The third component, $l_a \mapsto \{@\mathit{proto} : l_{op}, \texttt{src}{:}13, \texttt{width}{:}17\}$, implements the attributes of node 3 as an object at l_a, with its @*proto* field pointing to the object prototype at l_{op}. For each attribute of node 3 (e.g. "src"), l_a has a field of the same name storing the corresponding attribute node. In this case, the fields named `src` and `width` store 13 and 17, namely the addresses of the respective attribute nodes.

The next two components, $\mathsf{ANode}\,(13, \text{“src”}, \mathit{fid}_1)$ and $\mathsf{FL}\,(\mathit{fid}_1, 13, [1])$, respectively implement the attribute node with identifier 13 and its forest listener (where the underlying text forest contains a single text node with identifier 1). The definition of ANode is analogous to that of ENode and is given in Fig. 7.5. Similarly, the next line describes the attribute with identifier 17 and its forest listener.

The next line, $\mathsf{TNode}\,(1, \text{“goo.gl/K4S0d0”}, \mathit{fid}_3, 13) * \mathsf{FL}\,(\mathit{fid}_3, 1, [\,])$, describes the text node with identifier 1 and value "goo.gl/K4S0d0" (i.e. the only child of the "src" attribute with identifier 13), together with its forest listener at location fid_3 (where the underlying forest is empty as text nodes have no children). Similarly, the last line describes the text node with identifier 23 and its forest listener.

The last definition in Fig. 7.5, $\mathtt{TLs}(n, ts)$, describes the tag listeners associated with node n where l_{TL} denotes the location of the tag listener NodeList prototype (`TLProto`). Each listener object at location *fid* (where $(\text{s}, \mathit{fid}) \in ts$) is an instance of `TLProto` with fields: i) @*proto*, storing the address of the `TLProto` prototype l_{TL}; ii) `__pattern__`, storing the tag name that is searched for (s); and iii) `__startPoint__`, storing the address of the node with which it is associated (n).

This concludes the description of our DOM implementation in JavaScript. In what follows we show that this implementation is correct with respect to the DOM specification given in §5.

7.2. DOM Implementation Correctness

We define what it means to correctly implement the DOM library and then show that our DOM implementation sketched in §7.1 (with more details in [44]) is correct with respect to our specification in §5, by means of a *hybrid* translation as outlined in §6.

Our goal is to show that everything we can prove about JavaScript programs calling the DOM can also be proved about the same programs calling our DOM implementation instead. We do this via a refinement proof using a hybrid translation function that relates DOM heaps onto JavaScript heaps.

Recall from our DOM axioms (Figs. 5.2-5.3) that many DOM operations, such as `nextSibling`, are specified through *incomplete* DOM heaps. To understand the translation of these incomplete heaps, we consider the DOM heap $\mathbf{h}_0$ defined in (7.2) below.

$$\mathbf{h}_0 \triangleq \mathcal{R}_d \mapsto \#\text{doc}_d[\text{s}_u[\varnothing, \#\text{text}_l[\text{s}_1] \otimes \#\text{text}_n[\text{s}_2]]]\&\varnothing \tag{7.2}$$

The $\mathbf{h}_0$ describes a complete DOM heap at address $\mathcal{R}_d$, containing the document node (with identifier d) with an empty grove ($\varnothing$) and a non-empty document element (with identifier u). Moreover, the element node u itself contains no attributes ($\varnothing$) and its child forest comprises two text nodes (with identifiers l and n). Using abstract allocation (Def. 15), this heap can be split as $\mathbf{h}_1 \bullet \mathbf{h}_2$ with $\mathbf{h}_1$ and $\mathbf{h}_2$ defined in (7.3) below:

$$\begin{aligned} \mathbf{h}_1 &\triangleq \mathcal{R}_d \mapsto \#\mathrm{doc}_d[\mathrm{s}_u[\varnothing, \mathbf{x}]] \ \& \ \varnothing \\ \mathbf{h}_2 &\triangleq \mathbf{x} \mapsto \#\mathrm{text}_l[\mathrm{s}_1] \otimes \#\mathrm{text}_n[\mathrm{s}_2] \end{aligned} \tag{7.3}$$

To translate these incomplete heaps correctly, we introduce two additional concepts: *interfaces* and *crust*.

Interfaces When a DOM heap is split through abstract allocation, the constituent heaps are agnostic to one-another's shapes. For instance, the DOM heap $\mathbf{h}_1$ in (7.3) has no knowledge of the forest placed within $\mathbf{x}$; *mutatis mutandis* for $\mathbf{h}_2$. This is however not the case for the implementation: the concrete representation of $\mathbf{h}_2$ relies on information from the concrete representation of $\mathbf{h}_1$ and vice versa. For instance, the representation of the text node with identifier n includes a pointer to its parent (the element node u). Similarly, the representation of the element node u includes an array of pointers to its immediate children, i.e. l and n. Thus, when translating a DOM heap with abstract addresses, we require auxiliary information describing how the abstract addresses and context holes connect together. We track this additional piece of information associated with each abstract address $\mathbf{x} \in \textsc{AAdd}$ through an *interface*. In particular, an *in-interface* records the list of identifiers of the nodes that form the data (in this case a forest) pointed to by $\mathbf{x}$. In (7.3) above, the in-interface of $\mathbf{x}$ is $L \triangleq [l, n]$. An *out-interface* records the identifier of the parent node of the context hole $\mathbf{x}$. In (7.3) above the out-interface of $\mathbf{x}$ is u. The interface associated with $\mathbf{x}$ is (L, u).

In general, the out-interface for all DOM data (Def. 58) comprises a single node identifier denoting the identifier of the parent as described above. The in-interface for all DOM data, *bar attribute sets*, is captured by a list of identifiers of the nodes that form the data. The in-interface for attribute sets records additional information. Consider the partial heap $\mathbf{h}'$ below which describes an element node (with identifier p) with no children ($\varnothing$) and two attributes: one with name s' and identifier a, another with name s'' and identifier b. Using abstract allocation, the $\mathbf{h}'$ heap can be split into $\mathbf{h}'_1 \bullet \mathbf{h}'_2$ as shown below:

$$\mathbf{h}' \triangleq \mathbf{y} \mapsto \mathrm{s}_p[\mathrm{s}'_a[\varnothing] \odot \mathrm{s}''_b[\varnothing], \varnothing]$$

$$\mathbf{h}'_1 \triangleq \mathbf{y} \mapsto \mathrm{s}_p[\mathbf{z}, \varnothing] \qquad \mathbf{h}'_2 \triangleq \mathbf{z} \mapsto \mathrm{s}'_a[\varnothing] \odot \mathrm{s}''_b[\varnothing]$$

As before, the DOM heap $\mathbf{h}'_1$ has no knowledge of the attribute set placed within $\mathbf{z}$. However, recall from the JavaScript presentation of an element node (ENode in Fig. 7.5) that each element tracks its associated attributes via the `__attributes__` field. The value of the `__attributes__` field (l_a) denotes the address of an object that maps each attribute name to the corresponding attribute object. For instance, the representation of the element node p above contains an object of the form $l_a \Mapsto \{@proto : l_{op}, \mathtt{s}' : a, \mathtt{s}'' : b\}$. As such, the in-interface of $\mathbf{z}$ must not only record the identifiers of the attributes placed within it ($[a, b]$), but also their names ($[\mathrm{s}', \mathrm{s}'']$). In the example above, the in-interface of $\mathbf{z}$ is $L'{=}[(\mathrm{s}', a), (\mathrm{s}'', b)]$. As before, the out-interface of $\mathbf{z}$ is p and the interface of $\mathbf{z}$ is (L', p).

To translate the incomplete heaps correctly, we thus parameterise the translation function with an interface function that records the interfaces associated with abstract addresses.

Definition 80 (Interfaces)**.** Given the sets of DOM identifiers $\textsc{Id}$ and DOM strings S (Def. 56), the set of *in-interfaces*, $\textsc{In}$, and the set of *out-interfaces*, $\textsc{Out}$, are defined as follows, where $\textsc{Id}_\emptyset \triangleq \textsc{Id} \uplus \{\mathtt{null}\}$:

$$\textsc{In} \triangleq \textsc{List}\langle\textsc{Id}\rangle \cup \textsc{List}\langle \mathrm{S} \times \textsc{Id}\rangle \qquad \textsc{Out} \triangleq \textsc{Id}_\emptyset$$

The set of *interfaces* is $\iota \in \textsc{Inter} \triangleq \textsc{In} \times \textsc{Out}$. For an interface ι, ι^{in} and ι^{out} denote its first and second projections respectively.

Given the set of abstract addresses $\textsc{AAdd}$ (Def. 8), the set of *in-interface functions*, $\mathcal{I}^{\textsc{in}}$, and the set of *out-interface functions*, $\mathcal{I}^{\textsc{out}}$, are defined as follows:

$$\mathcal{I}^{\textsc{in}} \triangleq \textsc{AAdd} \rightharpoonup \textsc{In} \qquad \mathcal{I}^{\textsc{out}} \triangleq \textsc{AAdd} \rightharpoonup \textsc{Out}$$

The set of *interface functions* is $I \in \mathcal{I} \triangleq \mathcal{I}^{\textsc{in}} \times \mathcal{I}^{\textsc{out}}$. For an interface function I, the I^{in} and I^{out} denote its first and second projections, respectively.

Given an interface function $I \in \mathcal{I}$, and an abstract address $\mathbf{x} \in \textsc{AAdd}$, we write $I(\mathbf{x}){=}(L, u)$ when $I^{\mathsf{in}}(\mathbf{x}){=}L$ and $I^{\mathsf{out}}(\mathbf{x}){=}u$.

Crust When translating *incomplete* DOM heaps to JavaScript heaps, their footprints may not match. For instance, consider the first axiom of the `nextSibling` operation in Fig. 5.2, repeated below:

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}} \otimes \mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$
$$\mathtt{r} := \mathtt{n.nextSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}} \otimes \mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$

When $\mathbf{h}_1$ and $\mathbf{h}_2$ are the incomplete heaps defined in (7.3) above and $\mathtt{n}{=}l$, then $\mathbf{h}_2$ sufficiently captures the footprint of `n.nextSibling`. However, this is not the case at the implementation level. Our implementation of `nextSibling` looks up the parent of l (i.e. u in $\mathbf{h}_1$), iterates over the child list of u until it reaches l and then returns the next node in the list (n). As such, when $\mathtt{n}{=}l$, the footprint of `n.nextSibling` at the implementation level contains nodes l and n as well as the resources associated with the child list of l's parent (u). Therefore, the concrete representation of $\mathbf{h}_2$ must include not only the resources of $\mathbf{h}_2$, but also the resources needed for accessing the child list of u, which lies outside the scope of $\mathbf{h}_2$. This results in a *locality mismatch* between the abstract heap $\mathbf{h}_2$ and its concrete representation. We refer to this additional resource required (i.e. the resources associated with u's child list) as the *crust* of $\mathbf{h}_2$. That is, the crust of an incomplete DOM heap (e.g. $\mathbf{h}_2$) comprises the resources associated with the child lists of those parent nodes (e.g. u) that are *not* included in the heap but fragments of their descendants are (e.g. $[l, n]$). Given an abstract heap $\mathbf{h}$, we refer to the set of such parent nodes in $\mathbf{h}$ and their associated descendant fragments as the *crust set* of $\mathbf{h}$. For instance, the crust set of $\mathbf{h}_2$ is $\{([l, n], u)\}$. Note that $([l, n], u)$ denotes the interface of $\mathbf{x}$, i.e. the address at which the deallocated fragment resides.

Let $\mathsf{ids}(\mathbf{h})$, defined shortly, denote the set of node identifiers present in $\mathbf{h}$. For instance, with $\mathbf{h}_1$ and $\mathbf{h}_2$ in (7.3), we have $\mathsf{ids}(\mathbf{h}_1){=}\{d, u\}$ and $\mathsf{ids}(\mathbf{h}_2){=}\{l, n\}$. Observe that in general the crust set comprises entries of the form (L, u) where i) the parent u is not included in the heap (since otherwise the parent and its child list will already be included in the concrete resources produced by the translation); and ii) the fragment L is abstractly deallocated from the child list of u and now resides at an abstract address $\mathbf{x}$, where (L, u) denotes the interface of $\mathbf{x}$. Given an interface function I, we thus define the crust set of a heap $\mathbf{h}$, written $\mathsf{cset}\,(\mathbf{h}, I)$, as a set comprising pairs of the form (L, u) where i) $u \notin \mathsf{ids}(\mathbf{h})$; and ii) $(L, u){=}I(\mathbf{x})$ for some abstract address $\mathbf{x}$ such that $\mathbf{x} \in \mathit{dom}(\mathbf{h})$. For instance, for $\mathbf{h}_1$ and $\mathbf{h}_2$ in (7.3), we have $\mathsf{cset}\,(\mathbf{h}_1, I)\,{=}\emptyset$ and $\mathsf{cset}\,(\mathbf{h}_2, I)\,{=}\{([l, n], u)\}$, where $I(\mathbf{x}){=}([l, n], u)$.

Definition 81 (Crust set)**.** Given the set of DOM logical heaps $\textsc{LHeap}_{\mathbb{DOM}}$ (Def. 59) and the sets of interfaces $\textsc{Inter}$ and interface functions $\mathcal{I}$ (Def. 80), the *crust set function*, $\mathsf{cset}\,(.,.) : \textsc{LHeap}_{\mathbb{DOM}} \times \mathcal{I} \to \textsc{Inter}$, is defined as follows, for all $\mathbf{h} \in \textsc{LHeap}_{\mathbb{DOM}}$ and $I \in \mathcal{I}$, with $\mathsf{ids}(\mathbf{h})$ defined in Def. 82:

$$\mathsf{cset}\,(\mathbf{h}, I) \triangleq \Big\{(L, u) \,\Big|\, \mathbf{x} \in \mathit{dom}(\mathbf{h}) \wedge I(\mathbf{x}) = (L, u) \wedge u \notin \mathsf{ids}(\mathbf{h})\Big\}$$

Definition 82 (Identifier function)**.** Given the set of DOM logical data $\textsc{LData}_{\mathbb{DOM}}$ (Def. 58) and the set of DOM node identifiers $\textsc{Id}$ (Def. 56), the *DOM data identifier function*, $\mathsf{idsD}(.) : \textsc{LData}_{\mathbb{DOM}} \to \mathcal{P}(\textsc{Id})$, is defined inductively over the structure of DOM logical data as follows, where $\varnothing_\dagger \in \{\varnothing_e, \varnothing_f, \varnothing_a, \varnothing_{tf}, \varnothing_g\}$ and $\ddagger \in \{\otimes, \odot, \oslash, \oplus\}$:

$$\mathsf{idsD}(\varnothing_\dagger) \triangleq \emptyset \qquad \mathsf{idsD}(\mathbf{x}) \triangleq \emptyset \qquad \mathsf{idsD}(\mathbf{d}_1 \ddagger \mathbf{d}_2) \triangleq \mathsf{idsD}(\mathbf{d}_1) \uplus \mathsf{idsD}(\mathbf{d}_2)$$

$$\mathsf{idsD}(\#\mathrm{text}_n[\mathrm{s}]_{fs}) \triangleq \{n\} \qquad \mathsf{idsD}(\mathrm{s}_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs}) \triangleq \{n\} \uplus \mathsf{idsD}(\mathbf{as}) \uplus \mathsf{idsD}(\mathbf{f})$$

$$\mathsf{idsD}(\mathrm{s}_n[\mathbf{tf}]_{fs}) \triangleq \{n\} \uplus \mathsf{idsD}(\mathbf{tf})$$

$$\mathsf{idsD}(\#\mathrm{doc}_n[\mathbf{d}]^{ts}_{fs} \;\&\; \mathbf{g}) \triangleq \{n\} \uplus \mathsf{idsD}(\mathbf{d}) \uplus \mathsf{idsD}(\mathbf{g})$$

Given the set of DOM logical heaps $\textsc{LHeap}_{\mathbb{DOM}}$ (Def. 59), the *DOM identifier function*, $\mathsf{ids}(.) : \textsc{LHeap}_{\mathbb{DOM}} \to \mathcal{P}(\textsc{Id})$, is defined as follows, for all $\mathbf{h} \in \textsc{LHeap}_{\mathbb{DOM}}$:

$$\mathsf{ids}(\mathbf{h}) \triangleq \biguplus_{a \in \mathit{dom}(\mathbf{h})} \mathsf{idsD}(\mathbf{h}(a))$$

Recall that the crust of a heap $\mathbf{h}$ describes the resources associated with the child lists of the parent nodes in the crust set $\mathsf{cset}\,(\mathbf{h}, I)$. Observe that when calculating the crust of an incomplete heap, we have partial information about the child list of those nodes in the crust set. For instance, when calculating the crust of $\mathbf{h}_2$ in (7.3) above, we only know that the child list of its parent (u) contains the list fragment $[l, n]$ and know nothing of the remaining elements in its child list. More concretely, we know that the child list of u is of the form $L_1{+\!+}[l,n]{+\!+}L_2$ for some list fragments L_1 and L_2. We write $\mathsf{clist}(\mathbf{h}, I, u)$, defined shortly, for the set of all possible child lists associated with u.

When defining clist, additional care is required to ensure that we preserve all parent-child relations between nodes. For instance, given the DOM heap $\mathbf{h}_0$ in (7.3) above, let us now split it as $\mathbf{h}_3 \bullet \mathbf{h}_4$ with $\mathbf{h}_3$ and $\mathbf{h}_4$ defined in (7.4) below.

$$\begin{aligned} \mathbf{h}_3 &\triangleq \mathcal{R}_d \mapsto \#\mathrm{doc}_d[\mathrm{s}_u[\varnothing, \mathbf{x} \otimes \mathbf{y}]] \mathbin{\&} \varnothing \\ \mathbf{h}_4 &\triangleq \mathbf{x} \mapsto \#\mathrm{text}_l[\mathrm{s}_1] \bullet \mathbf{y} \mapsto \#\mathrm{text}_n[\mathrm{s}_2] \end{aligned} \tag{7.4}$$

We then have $I(\mathbf{x}){=}([l], u)$, $I(\mathbf{y}){=}([n], u)$ and $\mathsf{cset}\,(\mathbf{h}_4, I) = \{([l], u), ([n], u)\}$. Since both l and r have u as their parents, we must ensure that the possible child lists of u contain both l and r. That is, for all $L \in \mathsf{clist}(\mathbf{h}_4, I, u)$, we must guarantee that there exist some some L_1, L_2, L_3 and L_4 such that $L = L_1{+\!+}[l]{+\!+}L_2 = L_3{+\!+}[n]{+\!+}L_4$. We thus define clist as follows.

Definition 83 (Crust child lists)**.** Given the sets of DOM logical heaps $\textsc{LHeap}_{\mathbb{DOM}}$ (59), interface functions $\mathcal{I}$ (Def. 80) and DOM node identifiers $\textsc{Id}$ (Def. 56), the *crust child list function*, $\mathsf{clist}(.,.,.) : \textsc{LHeap}_{\mathbb{DOM}} \times \mathcal{I} \times \textsc{Id} \to \textsc{List}\langle\textsc{Id}\rangle$, is defined as follows, for all $\mathbf{h} \in \textsc{LHeap}_{\mathbb{DOM}}$, $I \in \mathcal{I}$ and $u \in \textsc{Id}$:

$$\mathsf{clist}(\mathbf{h}, I, u) \triangleq \left\{ L \;\middle|\; \begin{array}{l} (-, u) \in \mathsf{cset}\,(\mathbf{h}, I) \wedge \forall L_0. \\ \quad (L_0, u) \in \mathsf{cset}\,(\mathbf{h}, I) \Rightarrow \exists L_1, L_2.\ L{=}L_1{+\!+}L_0{+\!+}L_2 \end{array} \right\}$$

with the definition of cset as given in Def. 81.

We can now formulate the definition of crust as the resources associated with the child lists of the parent nodes in the crust set. For every parent node u in the crust set (i.e. $(-, u) \in \mathsf{cset}\,(\mathbf{h}, I)$) and its child list $L \in \mathsf{clist}(\mathbf{h}, I, u)$, the crust includes the pointer of u to its child list, $(u, \texttt{childNodes}) \mapsto l_c$, as well as the child list itself (denoted by $\mathsf{FL}\,(l_c, u, L)$).

Definition 84 (Crust)**.** Given the sets of DOM logical heaps $\textsc{LHeap}_{\mathbb{DOM}}$ (Def. 59), interface functions $\mathcal{I}$ (Def. 80) and JavaScript logical heaps $\textsc{JSLHeap}$ (Def. 72), the *crust function*, $\mathsf{Crust}\,(.,.) : \textsc{LHeap}_{\mathbb{DOM}} \times \mathcal{I} \to$

$\mathcal{P}(\text{JSLHeap})$, is defined as follows, for all $\mathbf{h} \in \text{LHeap}_{\mathbb{DOM}}$ and $I \in \mathcal{I}$:

$$\mathsf{Crust}(\mathbf{h}, I) \triangleq \underset{\{u \mid (-,u) \in \mathsf{cset}(\mathbf{h}, I)\}}{\circledast} \begin{pmatrix} \exists l_c, L. L \mathrel{\dot{\in}} \mathsf{clist}(\mathbf{h}, I, u) \\ \quad * (u, \texttt{childNodes}) \mapsto l_c * \mathsf{FL}(l_c, u, L) \end{pmatrix}$$

where FL is as defined in Fig. 7.5, and the definition of cset and clist are as given in Defs. 81 and 83, respectively.

We now have all the ingredients for translating DOM heaps. Given an interface function I, the translation of a DOM heap $\mathbf{h}$, written $\langle\!\langle \mathbf{h} \rangle\!\rangle^{I}_{\text{DOM}}$, includes i) the JavaScript prototypes given by `Protos` (Def. 79) describing the JavaScript heap representation of the DOM objects; ii) the crust of $\mathbf{h}$, i.e. $\mathsf{Crust}(\mathbf{h}, I)$ (Def. 84); and iii) the translation of the DOM resources in $\mathbf{h}$ defined via an auxiliary function $\mathsf{H}(\mathbf{h}, I)$, defined shortly.

Observe that as well as the additional resources of $\mathsf{Crust}(\mathbf{h}, I)$, our translation requires the resources given by `Protos`, recording our implementation of the DOM operations in the JavaScript heap. The need for the additional resources of `Protos` is due to JavaScript: in JavaScript function bodies are stored in the heap and are treated as a resource. As such, in order to ensure the correct despatch of DOM function calls in our implementation, we must hold the necessary resources in the JavaScript heap.

Definition 85 (DOM heap translation)**.** Given the JavaScript representation of DOM objects `Protos` (Def. 79), the crust function Crust (Def. 84), the set of DOM logical heaps $\text{LHeap}_{\mathbb{DOM}}$ (Def. 59), the set of interface functions $\mathcal{I}$ (Def. 80) and the set of JavaScript heaps JSLHeap (Def. 72), the *DOM heap translation* function, $\langle\!\langle . \rangle\!\rangle^{(.)}_{\text{DOM}} : \text{LHeap}_{\mathbb{DOM}} \times \mathcal{I} \to \mathcal{P}(\text{JSLHeap})$, is defined as follows, for all $\mathbf{h} \in \text{LHeap}_{\mathbb{DOM}}$ and $I \in \mathcal{I}$:

$$\langle\!\langle \mathbf{h} \rangle\!\rangle^{I}_{\text{DOM}} \triangleq \texttt{Protos} * \mathsf{Crust}(\mathbf{h}, I) * \mathsf{H}(\mathbf{h}, I) * \mathsf{true}$$

where, given the separation algebra of DOM logical heaps $(\text{LHeap}_{\mathbb{DOM}}, \bullet, \{\mathbf{0}\})$ in Def. 59, the *auxiliary DOM heap translation function*, $\mathsf{H}(.,.) : \text{LHeap}_{\mathbb{DOM}} \times \mathcal{I} \to \mathcal{P}(\text{JSLHeap})$, is defined as follows:

$$\mathsf{H}(\mathbf{0}, I) \triangleq \mathsf{emp} \qquad \mathsf{H}(\mathcal{R}_d \mapsto \mathbf{d}, I) \triangleq \mathsf{D}(\mathbf{d})^{([d], \texttt{null})}_{I}$$

$$\mathsf{H}(\mathbf{x} \mapsto \mathbf{d}, I) \triangleq \mathsf{D}(\mathbf{d})^{I(\mathbf{x})}_{I} \qquad \mathsf{H}(\mathbf{h}_1 \bullet \mathbf{h}_2, I) \triangleq \mathsf{H}(\mathbf{h}_1, I) * \mathsf{H}(\mathbf{h}_2, I)$$

with the data translation function, $\mathsf{D}\left(.\right)^{(.)}_{(.)}$, given in Def. 86.

Observe that when translating a DOM heap $\mathbf{h}$, as well as the resources associated with the prototypes `Protos`, the crust $\mathsf{Crust}\left(\mathbf{h}, I\right)$ and the DOM heap itself $\mathsf{H}\left(\mathbf{h}, I\right)$, the translation includes additional arbitrary resources described by true. This is because our JavaScript implementation may allocate additional resources that are not subsequently deallocated explicitly as this is carried out by the JavaScript garbage collector. For instance, a function call in JavaScript alters the current scope chain by allocating a new scope object l_f and prepending it to the current scope chain, thus causing the function body to be evaluated in the local scope l_f. Upon returning from the function call, the original scope is recovered by removing the newly allocated scope object l_f from the scope chain. However, although l_f is rendered superfluous thereafter, it is not explicitly deallocated and is left to the garbage collector to be freed in due course. As such, the footprint of our implementation may include arbitrary redundant resources such as l_f. As it is not sound to simply forget parts of the heap, this residual portion is hidden in true.

The $\mathsf{H}\left(\mathbf{h}, I\right)$ function provides a mapping from a DOM heap $\mathbf{h}$ to JavaScript heaps. For an empty heap, $\mathbf{0}$, this mapping is defined simply as emp. For a composite heap, $\mathbf{h}_1 \bullet \mathbf{h}_2$, the mapping is defined as the composition of the mapped constituent heaps. For an address $a \in \textsc{Add}_{\mathbb{DOM}}$ (Def. 9), the $\mathsf{H}\left(a \mapsto \mathbf{d}, I\right)$ is defined via the data translation function $\mathsf{D}\left(.\right)^{(.)}_{I}$ indexed by appropriate interfaces. For the document address $\mathcal{R}_d$, the out-interface is `null` (the document node has no parent); while the in-interface is $[d]$ (the only node at address $\mathcal{R}_d$ is the document node with the designated identifier d). For an abstract address $\mathbf{x}$, the interfaces are determined by the interface function as $I(\mathbf{x})$.

We proceed with the formal definition of the data translation function $\mathsf{D}\left(.\right)^{(.)}_{(.)}$, followed by a description of the definition.

Definition 86 (DOM data translation)**.** Given the set of DOM logical data $\textsc{LData}_{\mathbb{DOM}}$ (Def. 58), the sets of interfaces $\textsc{Inter}$ and interface functions $\mathcal{I}$ (Def. 80) and the set of JavaScript logical heaps $\textsc{JSLHeap}$ (Def. 72), the *DOM data translation function*, $\mathsf{D}\left(.\right)^{(.)}_{(.)} : \textsc{LData}_{\mathbb{DOM}} \times \textsc{Inter} \times \mathcal{I} \to \mathcal{P}\left(\textsc{JSLHeap}\right)$, is defined inductively over the structure of DOM data as given in Fig. 7.6, where $\varnothing_\dagger \in \{\varnothing_e, \varnothing_f, \varnothing_a, \varnothing_{tf}, \varnothing_g\}$ and $\ddagger \in \{\otimes, \odot, \oslash, \oplus\}$.

$$\mathsf{D}\left(\varnothing_{\dagger}\right)_{I}^{(L,u)} \triangleq L\doteq[] \qquad\qquad \mathsf{D}\left(\mathbf{x}\right)_{I}^{(L,u)} \triangleq I(\mathbf{x})\doteq(L,u)$$

$$\mathsf{D}\left(\mathbf{d}_1 \ddagger \mathbf{d}_2\right)_{I}^{(L,u)} \triangleq \exists L_1, L_2.\, L\doteq L_1 +\!\!+ L_2 * \mathsf{D}\left(\mathbf{d}_1\right)_{I}^{(L_1,u)} * \mathsf{D}\left(\mathbf{d}_2\right)_{I}^{(L_2,u)}$$

$$\begin{aligned}\mathsf{D}\left(\#\mathrm{doc}_d[\mathbf{d}]_{fs}^{ts} \,\&\, \mathbf{g}\right)_{I}^{(L,u)} \triangleq\ & L\doteq[d] * u\doteq\mathtt{null} * \exists \mathit{fid}, L_e.\, fs\doteq\{\mathit{fid}\} \\ & *\big[(L_e\doteq[\,] * \mathsf{DNode}\left(d, \mathtt{null}, \mathit{fid}\right)) \\ & \quad \vee(\exists e.\ L_e\doteq[e] * \mathsf{DNode}\left(d, e, \mathit{fid}\right))\big] \\ & * \mathsf{FL}\left(\mathit{fid}, d, L_e\right) * \mathsf{TLs}\left(d, ts\right) * \mathsf{D}\left(\mathbf{d}\right)_{I}^{(L_e,d)} * \mathsf{D}\left(\mathbf{g}\right)_{I}^{(-,\mathtt{null})}\end{aligned}$$

$$\begin{aligned}\mathsf{D}\left(\mathrm{s}_n[\mathbf{as}, \mathbf{f}]_{fs}^{ts}\right)_{I}^{(L,u)} \triangleq\ & L\doteq[n] * \exists \mathit{fid}, l_a, L_a, L_f.\, fs\doteq\{\mathit{fid}\} * \mathsf{ENode}\left(n, \mathrm{s}, l_a, \mathit{fid}, u\right) \\ & * \mathsf{FL}\left(\mathit{fid}, n, L_f\right) * \mathsf{TLs}\left(n, ts\right) * \mathsf{D}\left(\mathbf{as}\right)_{I}^{(L_a,\mathtt{null})} * \mathsf{D}\left(\mathbf{f}\right)_{I}^{(L_f,n)} \\ & * l_a \Mapsto \{@\mathit{proto} : l_{op}, \mathrm{s}' : m \mid (\mathrm{s}', m) \in L_a\}\end{aligned}$$

$$\begin{aligned}\mathsf{D}\left(\#\mathrm{text}_n[\mathrm{s}]_{fs}\right)_{I}^{(L,u)} \triangleq\ & L\doteq[n] * \exists \mathit{fid}.\, fs\doteq\{\mathit{fid}\} * \mathsf{TNode}\left(n, \mathrm{s}, \mathit{fid}, u\right) \\ & * \mathsf{FL}\left(\mathit{fid}, n, [\,]\right)\end{aligned}$$

$$\begin{aligned}\mathsf{D}\left(\mathrm{s}_n[\mathbf{tf}]_{fs}\right)_{I}^{(L,u)} \triangleq\ & L\doteq[n] * \exists \mathit{fid}, L_{tf}.\, fs\doteq\{\mathit{fid}\} * \mathsf{ANode}\left(n, \mathrm{s}, \mathit{fid}\right) \\ & * \mathsf{FL}\left(\mathit{fid}, n, L_{tf}\right) * \mathsf{D}\left(\mathbf{tf}\right)_{I}^{(L_{tf},n)}\end{aligned}$$

Figure 7.6.: DOM data translation function

We now describe $\mathsf{D}\left(.\right)_{I}^{(L,u)}$ with the explicit in- and out-interface arguments L and u, following the cases of the inductive definition.
$\mathsf{D}\left(\varnothing_{\dagger}\right)$: There are no resource in the concrete representations; the $L\doteq[]$ simply states that there are no nodes in this data. This is to ensure the correct translation of data such as $\mathbf{f} \otimes \varnothing_f$.

$\mathsf{D}\left(\mathbf{x}\right)$: There are no resources in the concrete representation of a context hole, simply the appropriate connection between the interface passed, (L, u), and the interface of $\mathbf{x}$ given by I.

$\mathsf{D}\left(\mathbf{d}_1 \ddagger \mathbf{d}_2\right)$: The result is the $*$-composition of the constituent data translated, given an appropriate choice of interfaces.

$\mathsf{D}\left(\#\mathrm{doc}_d[\mathbf{d}]_{fs}^{ts} \,\&\, \mathbf{g}\right) \ldots \mathsf{D}\left(\mathrm{s}_n[\mathbf{tf}]_{fs}\right)$: The representation of each node stipulates $L\doteq[n]$ (or $L\doteq[d]$ in case of a document node) since there is only one node in the list denoted by L; as well as $fs\doteq\{\mathit{fid}\}$ asserting that the set of forest listeners is represented as a singleton set with fid denoting the address of the node's child list.

The child list in turn is captured by FL as defined in Fig. 7.5, with the second and third arguments denoting the parent of the child list and its contents, respectively. In the case of a text node, the contents are empty as text nodes have no children. For a document node, the contents are either $L_e{=}[\,]$ when the document element is empty, or $L_e{=}[e]$ where e is the identifier of the document element. For an element node, the contents is the list L_f; for an attribute node, the contents is the list L_{tf}. All three of L_e, L_f and L_{tf} are unknown at this point and are existentially quantified. Their values are later bound when translating the underlying child forests ($\mathbf{d}$ for L_e, $\mathbf{f}$ for L_f, $\mathbf{tf}$ for L_{tf}) with the appropriate interfaces. The in-interfaces are the child list contents, i.e. L_e, L_f and L_{tf}; the out-interfaces are the identifier of their respective parent nodes, i.e. d and n.

Each node is described by the correspondingly-named JavaScript node definition with DNode, ENode, TNode and ANode as defined in Fig. 7.5. The translations of the document and element nodes additionally include the representation of their tag listeners ts via TLs defined in Fig. 7.5.

The translation of a document node also includes the translation of its associated grove $\mathbf{g}$. Observe that since the nodes in the grove are orphaned (have no parents) the out-interface (the parent component) passed to the translation of $\mathbf{g}$ is `null`. Moreover, since the nodes in the grove are of no import to the representation of the document node itself, the in-interface passed to the grove translation is existentially quantified as $-$. This is because the JavaScript representation of a document node does not track the orphaned nodes in the grove. Rather, the nodes in the grove reside somewhere in the memory and may be garbage-collected if not in use (i.e. when no longer referenced). Were we to explicitly track the grove via the document node, all nodes in the grove would persist indefinitely, which is not a behaviour guaranteed by the DOM standard.

The translation of an element node also includes the translation of its associated attribute set $\mathbf{as}$. As attribute nodes have no parents (an attribute is not the child of the element node with which it is associated), the out-interface (the parent component) passed to the translation of $\mathbf{as}$ is `null`. As with the child list of the element L_f, the attribute set L_a is unknown at this point and is existentially quantified. Its value is determined later by $\mathsf{D}\,(\mathbf{as})_I^{(L_a,\texttt{null})}$. Recall that the JavaScript representation of an element node also includes an object tracking the location of each attribute node associated with the element. For instance, as we demonstrated on p. 223, the representation of the element in (7.1) includes the object at l_a tracking attributes associated with element 3. This is captured by $l_a \Mapsto \{\ldots\}$ in the translation of an element node. As described earlier, the list of attributes associated with the element node and their names is determined by L_a.

Implementation Correctness

We show that our implementation of the DOM library presented in §7.1 is correct with respect to its abstract specification in §5. To this end, we give a refinement proof that transforms triples of the form $\{p\}\mathtt{C}\{q\}$ with $p, q \in \text{JSLH}\textsc{eap}_{\mathbb{DOM}}$ and $\mathtt{C} \in \text{JSO}\textsc{p}_{\mathbb{DOM}}$, to corresponding triples of the form $\{p'\}\mathtt{C}'\{q'\}$ with $p', q' \in \mathcal{P}(\text{JSLH}\textsc{eap})$ and $\mathtt{C}' \in \text{JSO}\textsc{p}$. To do this, we first define a substitutive implementation function, $[\![.]\!]$, that maps a program $\mathtt{C} \in \text{JSO}\textsc{p}_{\mathbb{DOM}}$ onto a program $\mathtt{C}' \in \text{JSO}\textsc{p}$, by replacing the calls to DOM operations in $\mathtt{C}$ with calls to their JavaScript implementations. We next define a state translation function that maps a state $(h, \mathbf{h}) \in \text{JSLH}\textsc{eap}_{\mathbb{DOM}}$ to JavaScript heaps. We proceed with the definition of the implementation function.

Definition 87 (Implementation function)**.** Given the JS operations JSO<span style="font-variant:small-caps">p</span> (Def. 69), the DOM operations $\text{O}\textsc{p}_{\mathbb{DOM}}$ (Def. 61) and the $\text{JS}_{\mathbb{DOM}}$ operations $\text{JSO}\textsc{p}_{\mathbb{DOM}}$ (70), the *substitutive implementation function*, $[\![.]\!] : \text{JSO}\textsc{p}_{\mathbb{DOM}} \to \text{JSO}\textsc{p}$, is defined inductively over the structure of $\text{JS}_{\mathbb{DOM}}$ operations as follows, for all $\mathtt{C}_{\mathbb{DOM}} \in \text{O}\textsc{p}_{\mathbb{DOM}}$, where $\mathsf{imp}(\mathtt{C}_{\mathbb{DOM}})$ denotes the JavaScript implementation of $\mathtt{C}_{\mathbb{DOM}}$ in [44]:

$$[\![\mathtt{C}_{\mathbb{DOM}}]\!] \triangleq \mathsf{imp}(\mathtt{C}_{\mathbb{DOM}}) \qquad [\![\mathtt{x}]\!] \triangleq \mathtt{x} \qquad [\![\mathtt{v}]\!] \triangleq \mathtt{v} \qquad [\![\mathtt{this}]\!] \triangleq \mathtt{this}$$

$$[\![\mathtt{var\ x}]\!] \triangleq \mathtt{var\ x} \qquad [\![\mathtt{e1;e2}]\!] \triangleq [\![\mathtt{e1}]\!]\mathtt{;}[\![\mathtt{e2}]\!] \qquad [\![\mathtt{e1.x}]\!] \triangleq [\![\mathtt{e1}]\!]\mathtt{.x}$$

$$[\![\mathtt{e[e']}]\!] \triangleq [\![\mathtt{e}]\!]\mathtt{[}[\![\mathtt{e'}]\!]\mathtt{]} \qquad [\![\mathtt{e1 = e2}]\!] \triangleq [\![\mathtt{e1}]\!] \mathtt{=} [\![\mathtt{e2}]\!]$$

$$[\![\mathtt{e1} \ominus \mathtt{e2}]\!] \triangleq [\![\mathtt{e1}]\!] \ominus [\![\mathtt{e2}]\!] \qquad [\![\mathtt{e1.e2}]\!] \triangleq [\![\mathtt{e1}]\!]\ \mathtt{.}\ [\![\mathtt{e2}]\!]$$

$$[\![\mathtt{while(e)\{e'\}}]\!] \triangleq \mathtt{while(}[\![\mathtt{e}]\!]\mathtt{)\{}[\![\mathtt{e'}]\!]\mathtt{\}} \qquad [\![\mathtt{e(e')}]\!] \triangleq [\![\mathtt{e}]\!]\mathtt{(}[\![\mathtt{e'}]\!]\mathtt{)}$$

$$[\![\mathtt{function\ (x)\{e\}}]\!] \triangleq \mathtt{function\ (x)\{}[\![\mathtt{e}]\!]\mathtt{\}}$$

$$[\![\texttt{function f(x)\{e\}}]\!] \triangleq \texttt{function f(x)\{}[\![\texttt{e}]\!]\texttt{\}}$$

$$[\![\texttt{with(e)\{e'\}}]\!] \triangleq \texttt{with(}[\![\texttt{e}]\!]\texttt{)\{}[\![\texttt{e'}]\!]\texttt{\}} \qquad [\![\texttt{new e(e')}]\!] \triangleq \texttt{new }[\![\texttt{e}]\!]\texttt{(}[\![\texttt{e'}]\!]\texttt{)}$$

$$[\![\texttt{\{x1:e1...xn:en\}}]\!] \triangleq \texttt{\{x1:}[\![\texttt{e1}]\!]\texttt{...xn:}[\![\texttt{en}]\!]\texttt{\}}$$

$$[\![\texttt{delete e}]\!] \triangleq \texttt{delete }[\![\texttt{e}]\!]$$

We must next define a translation function that maps a JSLogic$_{\mathbb{DOM}}$ state $(h, \mathbf{h}) \in$ JSLHeap$_{\mathbb{DOM}}$ (Def. 73) to JavaScript heaps in JSLHeap (Def. 72). The first component of a JSLogic$_{\mathbb{DOM}}$ state, h, denotes a JavaScript heap in JSLHeap and its translation is thus simple. Recall that a JavaScript heap h is a mapping from references to values (Def. 72) where values contain lambda abstractions of the form $\lambda\texttt{x.e}$, representing function bodies. To translate h, it suffices to translate the lambda abstractions in its range by replacing each $\lambda\texttt{x.e}$ with $\lambda\texttt{x.}[\![\texttt{e}]\!]$ where $[\![.]\!]$ denotes the implementation function in Def. 87. That is, since our implementation function $[\![.]\!]$ replaces each DOM library call in the function bodies with a call to our implementation of the DOM operation instead, we must ensure that the heap representation of the functions are accordingly adjusted.

The second component of a JSLogic$_{\mathbb{DOM}}$, $\mathbf{h}$, denotes a DOM heap (Def. 73) which is translated via the DOM heap translation function in Def. 85, namely $\langle\!\langle \mathbf{h} \rangle\!\rangle^{\cdots}_{\text{DOM}}$, with an appropriate choice of interface function. We justify the choice of interface functions through an example.

Consider the axiom of `n.splitText(o)` in Fig. 5.2, repeated below:

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\text{text}_{\mathrm{N}}[\mathrm{S}.\mathrm{S}']_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{S}|\right\}$$
$$\texttt{r := n.splitText(o)}$$
$$\left\{\exists \mathrm{R}, \mathrm{F}'.\ \mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\text{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \otimes \#\text{text}_{\mathrm{R}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$

When $\mathtt{n}{=}\mathrm{N}$, $\alpha{=}\mathbf{x}$, $\mathrm{S}{=}\mathrm{s}$ and $\mathrm{S}'{=}\mathrm{s}'$, the precondition contains a single text node n at abstract address $\mathbf{x}$ with its text data described by the composite string s.s$'$. Observe that the in-interface of the abstract address $\mathbf{x}$ is $[n]$. Later in the postcondition, when $\mathrm{R}{=}r$, the first substring (s) remains with node n and the second (s$'$) forms a new text node r which is added as the right sibling of n. Observe that in doing so, the *in-interface* of $\mathbf{x}$ is changed from $[n]$ to $[n, r]$. As such, our choice of interface function must afford us the freedom to alter the in-interface of $\mathbf{x}$. On the other hand, the control over the *out-interface* of $\mathbf{x}$, as well as the interfaces of other abstract addresses, is beyond the reach of this DOM heap and lies with the context (frame). Therefore, our choice of interface function must be agnostic to these interfaces so that we can show our implementation correct for all valid choices of context interfaces. More generally, given a DOM heap $\mathbf{h}$ and an abstract address $\mathbf{x}$ in the domain of $\mathbf{h}$, our choice of interface function must allow us to modify the in-interfaces of $\mathbf{x}$, while prohibiting the manipulation of the out-interface of $\mathbf{x}$. Conversely, given an abstract address $\mathbf{x}$ in the range of $\mathbf{h}$ (i.e. when $\mathbf{x}$ is a context hole in $\mathbf{h}$), our choice of interface function must allow us to modify the out-interfaces of $\mathbf{x}$, while prohibiting the manipulation of the in-interface of $\mathbf{x}$.

We proceed with the definition of JavaScript heap translation followed by the state translation function with the appropriate choice of interface functions.

Definition 88 (JavaScript heap translation)**.** Given the set of JavaScript heaps JSLHEAP (Def. 72), the *JavaScript heap translation function*, $\langle\!\langle . \rangle\!\rangle_{\text{JS}}$: JSLHEAP $\to$ JSLHEAP, is defined as follows, for all $h \in$ JSLHEAP:

$$\langle\!\langle h \rangle\!\rangle_{\text{JS}}(l, \mathtt{f}) \triangleq \begin{cases} \lambda\mathtt{x}.\, [\![\mathtt{e}]\!] & \text{if } \exists \mathtt{x}, \mathtt{e}.\ h(l, \mathtt{f}) = \lambda\mathtt{x}.\mathtt{e} \\ h(l, \mathtt{f}) & \text{otherwsie} \end{cases}$$

We now formulate the JSLOGIC$_{\mathbb{DOM}}$ state translation function and define what it means to refine a JSLOGIC$_{\mathbb{DOM}}$ triple correctly. In what follows, given a DOM heap $\mathbf{h}$, we write $\mathbf{h}^{\mathsf{in}}$ and $\mathbf{h}^{\mathsf{out}}$ for the set of abstract addresses in its domain and range, respectively as follows, where AADD denotes the set of abstract addresses (Def. 8) and addr(.) denotes the DOM address function (Def. 58):

$$\mathbf{h}^{\mathsf{in}} \triangleq \text{AADD} \cap dom(\mathbf{h}) \qquad\qquad \mathbf{h}^{\mathsf{out}} \triangleq \bigcup_{\mathbf{d} \in rng(\mathbf{h})} \text{addr}(\mathbf{d})$$

Definition 89 (Correct refinement)**.** Given the JSLOGIC$_{\mathbb{DOM}}$ logical states JSLHEAP$_{\mathbb{DOM}}$ (Def. 73), interface functions $\mathcal{I}$ (Def. 80), JavaScript heaps JSLHEAP (Def. 72), the DOM heap translation function $\langle\!\langle . \rangle\!\rangle^{(.)}_{\text{DOM}}$ (Def. 85) and the JavaScript heap translation function $\langle\!\langle . \rangle\!\rangle_{\text{JS}}$ (Def. 88), the *state* *translation* function, $\llparenthesis . \rrparenthesis^{(.)} : \mathcal{P}(\text{JSLHeap}_{\mathbb{DOM}}) \times \mathcal{I} \to \mathcal{P}(\text{JSLHeap})$, is defined as follows, for all $p \in \text{JSLHeap}_{\mathbb{DOM}}$ and $I \in \mathcal{I}$:

$$\llparenthesis p \rrparenthesis^{I_c} \triangleq \left\{ (h_1 \circ h_2) \;\middle|\; \begin{array}{c} \exists h, \mathbf{h}, I_d.(h, \mathbf{h}) \in p \wedge h_1 = \langle\!\langle h \rangle\!\rangle_{\text{JS}} \wedge h_2 \in \langle\!\langle \mathbf{h} \rangle\!\rangle_{\text{DOM}}^{I_c \uplus I_d} \\ \wedge\, dom(I_d^{\text{in}}) = \mathbf{h}^{\text{in}} \wedge dom(I_d^{\text{out}}) = \mathbf{h}^{\text{out}} \end{array} \right\}$$

where $I_c \uplus I_d \triangleq (I_c^{\text{in}} \uplus I_d^{\text{in}}, I_c^{\text{out}} \uplus I_d^{\text{out}})$ with $\uplus$ denoting the standard disjoint function union.

Given the implementation function $[\![.]\!]$ (Def. 87), the *translation function* is defined as $\tau \triangleq (\llparenthesis . \rrparenthesis^{(.)}, [\![.]\!])$.

A $\text{JSLogic}_{\mathbb{DOM}}$ triple $\{P\}\ \texttt{C}\ \{Q\}$ is *refined correctly by* τ, written $\tau : \{P\}\ \texttt{C}\ \{Q\}$, if and only if it satisfies the following property, where $\epsilon \in \text{Env}$ denotes a JSLogic evaluation environment (Def. 74) and $|P|_\epsilon \triangleq \{(h, \mathbf{h}) \mid \epsilon, (h, \mathbf{h}) \models P\}$ (Def. 76).

$$\tau : \{P\}\ \texttt{C}\ \{Q\} \overset{\text{def}}{\iff} \forall \epsilon.\, \forall I_c \in \mathcal{I}.\, \forall r \in \mathcal{P}(\text{JSLHeap}_{\mathbb{DOM}})\,. \\ \left\{ \llparenthesis\, |P|_\epsilon * r \,\rrparenthesis^{I_c} \right\}\ [\![\texttt{C}]\!]\ \left\{ \llparenthesis\, |Q|_\epsilon * r \,\rrparenthesis^{I_c} \right\}$$

A triple is translated correctly, if and only if for all possible frames r and all interface functions I_c, the triple holds for the translated states. As we described above, in the transition from the pre- to the postcondition, the interfaces of some abstract addresses may change. This is realised by the existential quantification of I_d in both pre- and postconditions (in the definition of state translation $\llparenthesis . \rrparenthesis^{(.)}$) with the domain of I_d comprising the abstract addresses that are within the control of $\mathbf{h}$ and may be modified. Dually, I_c captures the interfaces of abstract addresses outside the control of $\mathbf{h}$ and (unlike I_d) does not change from the pre- to postcondition. This is captured by the universal quantification of I_c outside the triple.

Theorem 3 (Correct refinement)**.** *For all $P, Q \in \text{JSAst}_{\mathbb{DOM}}$ (Def. 76) and $\texttt{C} \in \text{JSOp}_{\mathbb{DOM}}$ (Def. 70):*

$$\{P\}\ \texttt{C}\ \{Q\} \implies \tau : \{P\}\ \texttt{C}\ \{Q\}$$

Proof. By induction on the structure of triples $\{P\}\ \texttt{C}\ \{Q\}$. The full proof is given in §B.

Part II.

CoLoSL: Concurrent Local Subjective Logic

8. Technical Background: CoLoSL

The verification of fine-grained concurrent algorithms is nontrivial. There has been much recent progress verifying such algorithms modularly using variants of concurrent separation logic [57, 19, 15, 55, 54, 53, 38, 10, 36, 9]. A key difficulty in verifying properties of shared-memory concurrent programs is to be able to reason compositionally about each thread in isolation, even though in reality the correctness of the whole system is the collaborative result of intricately intertwined actions of the threads. Such compositional reasoning is essential for verifying large concurrent systems, library code and more generally incomplete programs, and for replicating a programmer's intuition about why their implementations are correct.

Rely-guarantee (RG) reasoning [34] is a well-known technique for verifying shared-memory concurrent programs. In this method, each thread specifies its expectations (the rely condition R) of the transitions made by the environment as well as the transitions made by the thread itself (the guarantee condition G) where $R; G$ constitutes the overall *interference* between the current thread and the environment. However, in practice, formulating the rely and guarantee conditions is difficult: the *entire* program state is treated as a shared resource (accessible by all threads) where rely and guarantee conditions *globally* specify the behaviour of threads over the whole shared state and need to be checked throughout the execution of the thread. We proceed with a brief overview of the shortcomings of RG reasoning and how the existing approaches tackle *some* of these limitations. The global nature of RG reasoning limits its compositionality and practicality in the following ways:

1. Even when parts of the state are owned by a single thread, they are exposed to all other threads in the $R; G$ conditions. Simply put, the boundary between private (thread-local) and shared resources is blurred.

2. Since the shared resources are globally known, sharing of *dynamically* allocated resources is difficult. That is, *extending* the shared state is not easy.

3. When parts of the shared state are accessible by only a subset of threads, it is not possible to hide *either* the resources *or* their associated interference ($R; G$ conditions) from the unconcerned threads. In short, reasoning *locally* about threads with *disjoint* footprints is not possible.

4. Similarly, when different threads access different but *overlapping* parts of the shared state, it is not possible to hide *either* the resources *or* their associated interference from the unconcerned threads. In brief, reasoning *locally* about threads with *overlapping* footprints is not possible. As we will demonstrate, this issue is particularly pertinent when reasoning about concurrent operations on data structures with *unspecified* sharing such as graphs.

5. When describing the specification of a program module, the $R; G$ conditions need to reflect the entire shared state even when the module accesses only parts of it. This limits the *modularity* of verification since the module specification becomes context-dependent and it may not always be reusable.

6. Since the $R; G$ conditions are defined statically and cannot evolve, in order to temporarily disable or enable certain operations by certain threads (e.g. allowing a lock to be released only by the thread who has acquired it) one must appeal to complex (unscalable) techniques such as auxiliary states.

7. As a program executes, its footprint grows and shrinks in tandem with the resources it accesses. It is thus valuable for the reasoning to mimic the programmer's intuition by reflecting the changes in the footprint. This calls for appropriate (de)composition of the shared state as well as its associated interference which cannot be achieved with a global view of the shared state.

Recent work on RGSep [57] has combined the compositionality of separation logic [40, 32, 39] with the concurrent techniques of RG reasoning.

In RGSep reasoning, the program state is split into private (thread-local) and shared parts ensuring that the private resources of each thread are untouched by others, with the $R; G$ conditions specified over the shared state only. However, since the shared state itself remains globally specified and is visible to all threads in its entirety, this separation only addresses the first problem outlined above.

Setting out to overcome the limitations of both RG and RGSep reasoning, Feng introduced *local rely-guarantee* (LRG) reasoning in [19]. As in RG and unlike RGSep, in LRG reasoning the program state is treated as a single shared resource accessible by all threads. Moreover, the compositionality afforded by the separating conjunction of separation logic $*$ is applied to both resources of the shared state *and* the $R; G$ conditions. This way, threads can hide (frame off) irrelevant parts of the shared state and their interference (resolving 1-3, 5 above) allowing for more local reasoning provided that they operate on completely *disjoint* resources. However, when reasoning about data structures with intricate and unspecified sharing (e.g. graphs), since decomposition of overlapping resources is not possible in a disjoint manner, LRG reasoning enforces a global treatment of the shared state, thus betraying its very objective of locality (issue 4). Furthermore, as with RG reasoning, the $R; G$ conditions are specified statically (albeit in a decomposable manner); hence temporary (un)blocking of certain actions by certain threads is not easy (issue 6). Finally, while LRG succeeds to capture the programmer's intuition of the program state by dynamically growing and shrinking the footprint when dealing with disjoint resources, it fails to achieve this level of fine-grained locality when dealing with overlapping (entangled) resources (issue 7).

Much like LRG reasoning, the reasoning framework of concurrent abstract predicates (CAP) [15] and its extended variants [54, 53, 10, 9] apply the compositionality of separation logic to concurrent reasoning. In these techniques, the state is split into private (exclusive to each thread) and shared parts where the shared state itself is divided into an arbitrary number of *regions* disjoint from one another. Each region is governed by an *interference* relation I that encompasses both the R and G conditions: the transitions in I are enabled by *capabilities* and a thread holding the relevant capability locally (in its private state) may perform the associated transition. As with LRG, this fine-grained division of the shared state into regions, allows for more local reasoning with the constraint that the regions are pairwise *disjoint* (addressing 1, 3 and 5 above). Dynamically allocated resources can be shared through creation of new regions (resolving 2). Moreover, since CAP is built on top of *deny-guarantee* reasoning [18] (an extension to rely-guarantee reasoning where *deny* permissions are used to block certain behaviour over a period of time) dynamic manipulation of interference transitions is straightforward (resolving 6). However, since the contents of regions must be disjoint from one another, when tackling data structures with complex and unspecified patterns of sharing that do not lend themselves to a clean decomposition, the entire data structure along with its interference must be confined to a single region, forgoing the notion of locality once again (issue 4). Furthermore, since the capabilities and interference associated with each region are defined upon its creation and remain unchanged throughout its lifetime, it is often necessary to foresee all possible extensions to the region (including dynamically allocated resources and their interference). This not only limits the locality of reasoning, but also gives way to unnatural specifications that contrast with the program footprint (issue 7).

In what follows, we present an overview of the key ideas behind the program logic of CoLoSL by means of a simple example. We then proceed by comparing CoLoSL to the logic of CAP [15] and other contemporary program logics, most notably Iris [36, 35].

8.1. CoLoSL: Overview

We introduce the program logic of CoLoSL, where threads access one global shared state and are verified with respect to their *subjective views* of this state. Each subjective view is an assertion which provides a thread-specific description of *parts* of the shared state. It describes the partial shared resources necessary for the thread to run and the thread-specific interference describing how the thread and the environment may affect these shared resources. Subjective views may arbitrarily overlap with each other, and subtly expand and contract to the resources and the interference required by the current thread. This flexibility provides compositional reasoning for shared-memory concurrency.

A subjective view $\boxed{P}_I$ comprises an assertion P which describes *parts* of the global shared state, and an interference assertion I which characterises how this partial shared state may be changed by the thread or the environment. Similar to interference assertions of CAP, the I declares transitions of the form $[\mathsf{a}]^{\mathrm{T}} : Q \rightsquigarrow R$, where a thread in possession of the $[\mathsf{a}]^{\mathrm{T}}$ capability (in its private state) may carry out its transition and update parts of the shared state described by Q to those of R. Assertions of subjective views must be *stable*; that is, robust with respect to the interference from the environment (as prescribed in I).

Subjective views can always be duplicated using the Copy principle:

$$\boxed{P}_I \Rightarrow \boxed{P}_I * \boxed{P}_I \qquad \text{(Copy)}$$

This is because subjective views describe parts of the global shared state and thus a thread may choose to duplicate this view of the shared state and pass it on to other threads. In other words, in combination with the usual law of parallel composition (Par) of separation logic [39], the Copy principle yields a mechanism to distribute the shared state between several threads:

$$\frac{\{P_1\}\ \mathsf{C}_1\ \{Q_1\} \quad \{P_2\}\ \mathsf{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ \{Q_1 * Q_2\}} \ \text{(Par)}$$

As we demonstrate shortly, subsequent threads may choose to customise the subjective views passed on to them in order to describe those parts of the shared state relevant to them. To do this, we introduce several novel reasoning principles which allow us to expand and contract subjective views. The subjective views of each thread will be typically too strong, describing resources that are not being used by the thread. However, as we demonstrate in §8.2 below, before weakening the view, it is sometimes useful to strengthen some of the interference transitions to preserve global knowledge. We achieve this using the Shift principle:

$$\begin{array}{ll} \text{if} & I \sqsubseteq^P I' \\ \text{then} & \boxed{P}_I \Rightarrow \boxed{P}_{I'} \end{array} \qquad \text{(Shift)}$$

This principle states that an interference assertion I can be exchanged for another interference assertion I' that has the same projected effect on the subjective state P. That is, as far as the resources described by P are concerned, the interference assertions I' adequately captures the interference prescribed by I and may thus replace it. When this is the case, we write $I \sqsubseteq^P I'$ and say that the actions of I have been *shifted* to I'. The SHIFT principle can be used to forget (frame off) actions which are irrelevant to P. For instance, when $P \triangleq \mathtt{x} \mapsto 1$ and $I \triangleq \{\mathtt{y} \mapsto 2 \rightsquigarrow \mathtt{y} \mapsto 3\}$, then I can be shifted to $\emptyset$ as $I \sqsubseteq^P \emptyset$ holds. The $I \sqsubseteq^P \emptyset$ states that the (only) action $\mathtt{y} \mapsto 2 \rightsquigarrow \mathtt{y} \mapsto 3$ in I is of no consequence to the resources of P and may be simply forgotten by replacing it with the empty interference $\emptyset$. This is because the *effect* of the $\mathtt{y} \mapsto 2 \rightsquigarrow \mathtt{y} \mapsto 3$ action lies outside the resources described by P in that its precondition $\mathtt{y} \mapsto 2$ does not overlap with P.

The SHIFT principle can also be used to to strengthen actions with knowledge of the global shared state arising form the combination of I and P. We present an example of this application of the SHIFT principle in the upcoming section (§8.2). The SHIFT principle provides a flexible mechanism for interference manipulation and is in marked contrast with most existing formalisms, where interference is fixed throughout the verification.

With a possibly strengthened interference assertion, we can then frame off parts of the shared state and zoom on resources required by the current thread using the FORGET principle:

$$\boxed{P \uplus Q}_I \Rightarrow \boxed{P}_I \qquad \text{(FORGET)}$$

At this point, the SHIFT principle may be applied again to forget those actions that are no longer relevant to the new subjective view captured by P. However, a stable subjective view may no longer be stable after forgetting parts of the shared state. This is often due to the combined knowledge of P and I being too weak and can be avoided by strengthening I (through SHIFT) prior to forgetting.

These reasoning principles enable us to provide subjective views for the threads which are just right. We can proceed to verify the threads, knowing that their subjective views describe personalised preconditions which only describe the resource relevant to the individual threads. The resulting postconditions naturally describe overlapping parts of the shared state, which are then joined together using the disjoint concurrency rule and the MERGE principle:

$$\boxed{P}_{I_1} * \boxed{Q}_{I_2} \Rightarrow \boxed{P \uplus Q}_{I_1 \cup I_2} \qquad \text{(MERGE)}$$

The $P \uplus Q$ assertion describes the overlapping of the states described by P and Q, using the overlapping conjunction $\uplus$ [21, 31]. The new interference is simply the union of previous interferences. Using the SHIFT principle, we can once again simplify the interference assertion with respect to this new larger subjective view.

Lastly, locally owned resources (described by P) can be shared using the EXTEND principle:

$$\begin{array}{ll} \text{if} & P \copyright I \quad \text{and} \quad \mathsf{sat}(\mathcal{C}_1, \mathcal{C}_2) \\ \text{then} & P \Rrightarrow \exists \mathrm{T}.\ [\mathcal{C}_1]^{\mathrm{T}} * \boxed{P * [\mathcal{C}_2]^{\mathrm{T}}}_I \end{array} \qquad \text{(EXTEND)}$$

where T denotes a logical variable distinct from the unbound variables in P, $\mathcal{C}_1$ and $\mathcal{C}_2$. That is, $\mathrm{T} \notin fv(P, \mathcal{C}_1, \mathcal{C}_2)$. The side condition $P \copyright I$ ensures that the actions of the new interference assertion I are *confined* to P (more in §9), so as not to invalidate other threads' existing views of the shared state. Upon extending the shared state with P, one may additionally generate fresh capabilities, as described by the user-defined capability assertions $\mathcal{C}_1$ and $\mathcal{C}_2$, and distribute them between its local state ($\mathcal{C}_1$) and the shared state ($\mathcal{C}_2$). To ensure the freshness of the new capabilities, they are associated with a fresh *ticket* (identifier) T that is distinct from existing tickets. As $\mathcal{C}_1$ and $\mathcal{C}_2$ denote user-specified capability assertions, we require that these assertions are satisfiable, as stipulated by $\mathsf{sat}(\mathcal{C}_1, \mathcal{C}_2)$. The $\Rrightarrow$ connective denotes state *repartitioning* [15] or *view shift* [14]. More concretely, the $P \Rrightarrow Q$ states that a logical state satisfying P may be changed to one satisfying Q, so long as the underlying machine state does not change. In particular, the $(P \Rightarrow Q)$ implies $(P \Rrightarrow Q)$.

As we demonstrate later in §8.3, the main novelty of the EXTEND rule is that the new actions in I may refer to existing shared resources, unlike CAP where all possible futures of the region must be accounted for upon its creation.

With these reasoning principles, we are able to expand and contract subjective views to provide just the resources required by a thread. In essence, we provide a framing mechanism both on shared resources as well as their interferences even in the presence of overlapping footprints and entangled resources. In what follows, we illustrate our CoLoSL reasoning principles by sketching a proof of a variation of Dijkstra's token ring mutual exclusion algorithm [11].

8.2. Dijkstra's Token Ring Algorithm

Consider the program $\mathbb{INC}$ defined in Fig. 8.1, ignoring the assertions for now, with the $\mathbb{P}_{\mathtt{x}}$, $\mathbb{P}_{\mathtt{y}}$, and $\mathbb{P}_{\mathtt{z}}$ programs given in Fig. 8.2. This program is written in pseudo-code resembling C with additional constructs for concurrency. The statements between angle brackets `<.>` denote atomic instructions that cannot be interrupted by other threads. We write `C1 || C2` (e.g. line 5) for the parallel computation of `C1` and `C2` . This corresponds to the standard fork-join parallelism. In this example, after initialisation of the variables to 0, three threads are spawned to increment each variable in a lock-step fashion by executing $\mathbb{P}_{\mathtt{x}}$, $\mathbb{P}_{\mathtt{y}}$ and $\mathbb{P}_{\mathtt{z}}$ (Fig. 8.2), respectively. The $\mathbb{P}_{\mathtt{x}}$ is the first thread to run its increment operation, followed by $\mathbb{P}_{\mathtt{y}}$ and finally $\mathbb{P}_{\mathtt{z}}$. This process repeats until $\mathtt{x} = \mathtt{y} = \mathtt{z} = 10$. This example code is interesting because the threads are intricately intertwined. In the case of the $\mathbb{P}_{\mathtt{y}}$ thread, the programmer knows that the code depends on the values of variables `x` and `y` and that it can increment `y` so long as its value is less than that of `x`. The $\mathbb{P}_{\mathtt{y}}$ thread is further aware of a much more complex behaviour: given the initial setting where all variables have value 0, the thread may only increase the value of `y` by 1 if `x` is one more than `y`, and the environment may only increase `x` by one if `x` and `z` (and in fact `y`) have the same value. Finally, the programmer knows that at the end all the variables will have value 10.

In CoLoSL we can simply specify this complex behaviour of the resources associated with thread $\mathbb{P}_{\mathtt{y}}$. Consider the CoLoSL assertions accompanying $\mathbb{INC}$. After initialisation, line 3 of $\mathbb{INC}$ provides a standard assertion from separation logic [40, 32] with the variables-as-resource model [5]. The assertion declares that the variable cells addressed by `x`, `y` and `z` all have value 0. This variable resource in the thread-local state is fully owned by the thread. Using the EXTEND principle, the thread is able to give up this local resource and transfer it to the global shared state. For example, line 4 demonstrates the creation of the subjec-

$\mathbb{INC}$:

```
1  //{x ↦ − ∗ y ↦ − ∗ z ↦ −}
2  x = 0; y = 0; z = 0;
3  //{x ↦ 0 ∗ y ↦ 0 ∗ z ↦ 0}
4  //{∃T. [x ↦ 0 ∗ y ↦ 0 ∗ z ↦ 0]_I ∗ [a_x]^T ∗ [a_y]^T ∗ [a_z]^T}
5  (ℙx || ℙy || ℙz)
6  //{∃T. [x ↦10 ∗ y ↦10 ∗ z ↦10]_I ∗ [a_x]^T ∗ [a_y]^T ∗ [a_z]^T}
```

$$I \triangleq \begin{cases} [\mathsf{a_x}]^{\mathrm{T}} : \exists \mathrm{V}{\in}\{0..9\}.\mathtt{z} \mapsto \mathrm{V} * \mathtt{x} \mapsto \mathrm{V} \\ \qquad\qquad \rightsquigarrow \ \mathtt{z} \mapsto \mathrm{V} * \mathtt{x} \mapsto \mathrm{V}{+}1 \\ [\mathsf{a_y}]^{\mathrm{T}} : \exists \mathrm{V}{\in}\{0..9\}.\mathtt{x} \mapsto \mathrm{V}{+}1 * \mathtt{y} \mapsto \mathrm{V} \\ \qquad\qquad \rightsquigarrow \ \mathtt{x} \mapsto \mathrm{V}{+}1 * \mathtt{y} \mapsto \mathrm{V}{+}1 \\ [\mathsf{a_z}]^{\mathrm{T}} : \exists \mathrm{V}{\in}\{0..9\}.\mathtt{y} \mapsto \mathrm{V}{+}1 * \mathtt{z} \mapsto \mathrm{V} \\ \qquad\qquad \rightsquigarrow \ \mathtt{y} \mapsto \mathrm{V}{+}1 * \mathtt{z} \mapsto \mathrm{V}{+}1 \end{cases}$$

Figure 8.1.: The concurrent increment program together with a CoLoSL proof sketch. Lines starting with // contain assertions that describe the local state and the subjective shared state at the relevant program point. The codes of $\mathbb{P}_\mathtt{x}$, $\mathbb{P}_\mathtt{y}$ and $\mathbb{P}_\mathtt{z}$ programs and their proof sketches are provided in Fig. 8.2.

tive view $\boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0}_I$, where part of the underlying global shared state now contains the three variable cells. In doing so, the thread also creates the fresh capabilities described by the $[\mathsf{a_x}]^{\mathrm{T}} * [\mathsf{a_y}]^{\mathrm{T}} * [\mathsf{a_z}]^{\mathrm{T}}$ assertion on line 4. Note that these newly generated capabilities are associated with a fresh ticket T which ensures their distinctness. The interference assertion I describes how this part of the shared state may be manipulated. For instance, the action below states that y may be incremented when its value is below 10 and is one less than that of x:

$$[\mathsf{a_y}]^{\mathrm{T}} : \exists \mathrm{V}{\in}\{0..9\}.\ \mathtt{x} \mapsto \mathrm{V}{+}1 * \mathtt{y} \mapsto \mathrm{V} \rightsquigarrow \mathtt{x} \mapsto \mathrm{V}{+}1 * \mathtt{y} \mapsto \mathrm{V}{+}1$$

This update is only possible when the local state of a thread owns the capability described by $[\mathsf{a_y}]^{\mathrm{T}}$ locally. In this example, all three capabilities are owned locally by the thread as described by the $[\mathsf{a_x}]^{\mathrm{T}} * [\mathsf{a_y}]^{\mathrm{T}} * [\mathsf{a_z}]^{\mathrm{T}}$ assertion (line 4). In general, capabilities can be buried inside boxes, only to emerge as a consequence of an action (see §10). Moreover, the actions in I may be associated with capabilities other than those newly generated, including pre-existing capabilities as well as those not existing at the time of extension. For instance, we may write an action of the form $[\mathsf{a_y}]^{-} : P \rightsquigarrow Q$, asserting that for *all* tickets T, the thread in possession of $[\mathsf{a_y}]^{\mathrm{T}}$ may update parts of the shared state described by P to those of Q. In general, an action of the form $P \rightsquigarrow Q$ may perform three types

$\mathbb{P}_{\mathtt{x}}$:

```
//{ $\boxed{\mathtt{z} \mapsto 0 * \mathtt{x} \mapsto 0}_{I_{\mathtt{x}}} * [\mathsf{a_x}]^{\mathrm{T}}$ }
while(x < 10){
//{ $\boxed{\exists \mathrm{v}{\in}\{0..9\}.\ \mathtt{z} \mapsto \mathrm{v} * \mathtt{x} \mapsto \mathrm{v} \vee \mathtt{z} \mapsto \mathrm{v}{-}1 * \mathtt{x} \mapsto \mathrm{v}}_{I_{\mathtt{x}}} * [\mathsf{a_x}]^{\mathrm{T}}$ }
  if (x == z) { <x++> }
}
//{ $\boxed{\mathtt{z} \mapsto 10 * \mathtt{x} \mapsto 10 \vee \mathtt{z} \mapsto 9 * \mathtt{x} \mapsto 10}_{I_{\mathtt{x}}} * [\mathsf{a_x}]^{\mathrm{T}}$ }
```

$$I_{\mathtt{x}} \triangleq \begin{cases} [\mathsf{a_x}]^{\mathrm{T}}: \exists \mathrm{v}{\in}\{0..9\}.\mathtt{z} \mapsto \mathrm{v} * \mathtt{x} \mapsto \mathrm{v} \\ \qquad\qquad \rightsquigarrow \ \mathtt{z} \mapsto \mathrm{v} * \mathtt{x} \mapsto \mathrm{v}{+}1 \\ [\mathsf{a_z}]^{\mathrm{T}}: \exists \mathrm{v}{\in}\{0..9\}.\mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v}{+}1 * \mathtt{z} \mapsto \mathrm{v} \\ \qquad\qquad \rightsquigarrow \ \mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v}{+}1 * \mathtt{z} \mapsto \mathrm{v}{+}1 \end{cases}$$

$\mathbb{P}_{\mathtt{y}}$:

```
//{ $\boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 \vee \mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0}_{I_{\mathtt{y}}} * [\mathsf{a_y}]^{\mathrm{T}}$ }
while(y < 10){
//{ $\boxed{\exists \mathrm{v}{\in}\{0..9\}.\ \mathtt{x} \mapsto \mathrm{v} * \mathtt{y} \mapsto \mathrm{v} \vee \mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v}}_{I_{\mathtt{y}}} * [\mathsf{a_y}]^{\mathrm{T}}$ }
  if (y < x) { <y++> }
}
//{ $\boxed{\mathtt{x} \mapsto 10 * \mathtt{y} \mapsto 10 \vee \mathtt{x} \mapsto 11 * \mathtt{y} \mapsto 10}_{I_{\mathtt{y}}} * [\mathsf{a_y}]^{\mathrm{T}}$ }
```

$$I_{\mathtt{y}} \triangleq \begin{cases} [\mathsf{a_x}]^{\mathrm{T}}: \exists \mathrm{v}{\in}\{0..9\}.\mathtt{x} \mapsto \mathrm{v} * \mathtt{y} \mapsto \mathrm{v} * \mathtt{z} \mapsto \mathrm{v} \\ \qquad\qquad \rightsquigarrow \ \mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v} * \mathtt{z} \mapsto \mathrm{v} \\ [\mathsf{a_y}]^{\mathrm{T}}: \exists \mathrm{v}{\in}\{0..9\}.\mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v} \\ \qquad\qquad \rightsquigarrow \ \mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v}{+}1 \end{cases}$$

$\mathbb{P}_{\mathtt{z}}$:

```
//{ $\boxed{\mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0 \vee \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 0}_{I_{\mathtt{z}}} * [\mathsf{a_z}]^{\mathrm{T}}$ }
while(z < 10){
//{ $\boxed{\exists \mathrm{v}{\in}\{0..9\}.\ \mathtt{y} \mapsto \mathrm{v} * \mathtt{z} \mapsto \mathrm{v} \vee \mathtt{y} \mapsto \mathrm{v}{+}1 * \mathtt{z} \mapsto \mathrm{v}}_{I_{\mathtt{z}}} * [\mathsf{a_z}]^{\mathrm{T}}$ }
  if (z < y) { <z++> }
}
//{ $\boxed{\mathtt{y} \mapsto 10 * \mathtt{z} \mapsto 10 \vee \mathtt{y} \mapsto 11 * \mathtt{z} \mapsto 10}_{I_{\mathtt{z}}} * [\mathsf{a_z}]^{\mathrm{T}}$ }
```

$$I_{\mathtt{z}} \triangleq \begin{cases} [\mathsf{a_y}]^{\mathrm{T}}: \exists \mathrm{v}{\in}\{0..9\}.\mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v} * \mathtt{z} \mapsto \mathrm{v} \\ \qquad\qquad \rightsquigarrow \ \mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v}{+}1 * \mathtt{z} \mapsto \mathrm{v} \\ [\mathsf{a_z}]^{\mathrm{T}}: \exists \mathrm{v}{\in}\{0..9\}.\mathtt{y} \mapsto \mathrm{v}{+}1 * \mathtt{z} \mapsto \mathrm{v} \\ \qquad\qquad \rightsquigarrow \ \mathtt{y} \mapsto \mathrm{v}{+}1 * \mathtt{z} \mapsto \mathrm{v}{+}1 \end{cases}$$

Figure 8.2.: The $\mathbb{P}_{\mathtt{x}}$, $\mathbb{P}_{\mathtt{y}}$ and $\mathbb{P}_{\mathtt{z}}$ and their CoLoSL proof sketches

of actions: i) *modify* resources: the P and Q contain the same amount of resource with different values (e.g. action of $[\mathsf{a_y}]^{\mathrm{T}}$ above); ii) *remove* resources from the shared state and transfer them into the local state of the thread performing the action (e.g. when $P \triangleq Q * R$ for some resource R to be removed); and iii) *add* resources from the local state of the thread into the shared state (e.g. when $Q \triangleq P * R$ for some resources R to be added). These three behaviours are not mutually exclusive and an action may exhibit any combination of them.

Using the Copy principle for subjective views and the disjoint concurrency rule (Par), we obtain the following precondition for thread $\mathbb{P}_{\mathsf{y}}$:

$$\boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0}_{I} * [\mathsf{a_y}]^{\mathrm{T}}$$

However, this precondition is more complicated than we need. Intuitively, the specification of each thread should only use the variable resource relevant to that thread and need only consider actions that affect that resource. In this example, the extraneous piece of state is the variable cell $\mathtt{z}$. This additional resource might seem an acceptable price to pay, but straightforward generalisations to n participants yields extra state of $n-2$ variable cells with their associated interferences which are of no interest to the particular thread. Fundamentally, for large systems, the burden of carrying the whole shared state around to analyse all threads, can lead to intractable proofs.[1]

As a first try at simplifying the precondition, consider the following implication using the Forget and Shift principles given earlier, where $I \backslash \mathsf{a_z}$ denotes the interference obtained by removing the action of $\mathsf{a_z}$ from I:

$$\begin{aligned}
& \boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0}_{I} * [\mathsf{a_y}]^{\mathrm{T}} \\
\overset{(\textsc{Forget})}{\Rightarrow}\; & \boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0}_{I} * [\mathsf{a_y}]^{\mathrm{T}} \\
\overset{(\textsc{Shift})}{\Rightarrow}\; & \boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0}_{I \backslash \mathsf{a_z}} * [\mathsf{a_y}]^{\mathrm{T}} \\
\overset{(\text{stabilise})}{\Rightarrow}\; & \boxed{\exists \mathrm{v}, \mathrm{v}'.(\mathtt{x} \mapsto \mathrm{v} * \mathtt{y} \mapsto \mathrm{v}') \land \mathrm{v} \geq \mathrm{v}'}_{I \backslash \mathsf{a_z}} * [\mathsf{a_y}]^{\mathrm{T}}
\end{aligned}$$

[1] We refer the reader to [47, 48], where we generalise the token ring algorithm of $\mathbb{INC}$ in Fig. 8.1 to n threads ($\mathbb{P}_1 \,||\, \mathbb{P}_2 \,||\, \cdots \,||\, \mathbb{P}_n$), demonstrating that the specification of each thread and its proof sketch remain unchanged.

The thread $\mathbb{P}_{\mathtt{y}}$ does not modify $\mathtt{z}$ and we can thus forget the variable assertion $\mathtt{z} \mapsto 0$. The variable cell $\mathtt{z}$ is no longer visible to $\mathbb{P}_{\mathtt{y}}$ and thus the action of $[\mathsf{a_z}]^{\mathrm{T}}$ does not affect the resources described by the assertion of the subjective view neither in its current state nor at any point during its lifetime. That is, the current subjective view is unaffected by this action, *and* after undergoing any number of actions from I, the resulting subjective view remains unaffected by it. Using the Shift principle, we can therefore forget the action of $[\mathsf{a_z}]^{\mathrm{T}}$.

Finally, we stabilise the resulting subjective view such that it is invariant under all possible actions by the environment. However, since we no longer know the value of $\mathtt{z}$, after stabilising against the action of $[\mathsf{a_x}]^{\mathrm{T}}$, the resulting assertion is too weak. Intuitively, we know that $\mathtt{x}$ can only be incremented when its value is equal to $\mathtt{z}$ and $\mathtt{y}$. However, this is not reflected in the action of $[\mathsf{a_x}]^{\mathrm{T}}$. Since we have forgotten $\mathtt{z}$, there is nothing to constrain the increment on $\mathtt{x}$. Hence, we can only stabilise in a general way as given, losing information about how the values of $\mathtt{x}$ and $\mathtt{y}$ are connected together through $\mathtt{z}$.

It is however possible to give a stronger specification as follows, with $I_{\mathtt{y}}$ as defined in Fig. 8.2 and $I'_{\mathtt{y}}$ defined shortly:

$$\begin{array}{rl}
& \boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0}_{I} * [\mathsf{a_y}]^{\mathrm{T}} \\
\overset{(\text{Shift})}{\Rightarrow} & \boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0}_{I'_{\mathtt{y}}} * [\mathsf{a_y}]^{\mathrm{T}} \\
\overset{(\text{Forget})}{\Rightarrow} & \boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0}_{I'_{\mathtt{y}}} * [\mathsf{a_y}]^{\mathrm{T}} \\
\overset{(\text{Shift})}{\Rightarrow} & \boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0}_{I_{\mathtt{y}}} * [\mathsf{a_y}]^{\mathrm{T}} \\
\overset{(\text{stabilise})}{\Rightarrow} & \boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 \vee \mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0}_{I_{\mathtt{y}}} * [\mathsf{a_y}]^{\mathrm{T}}
\end{array} \tag{8.1}$$

where $I_{\mathtt{y}}$ is as defined in Fig. 8.2 and $I'_{\mathtt{y}}$ is obtained by rewriting the action of $[\mathsf{a_x}]^{\mathrm{T}}$ in I as follows:

$$I'_{\mathtt{y}} \triangleq \begin{cases}
[\mathsf{a_x}]^{\mathrm{T}} : \exists \mathtt{v} {\in} \{0..9\}.\ \mathtt{x} \mapsto \mathtt{v} * \mathtt{y} \mapsto \mathtt{v} * \mathtt{z} \mapsto \mathtt{v} \rightsquigarrow \mathtt{x} \mapsto \mathtt{v}{+}1 * \mathtt{y} \mapsto \mathtt{v} * \mathtt{z} \mapsto \mathtt{v} \\
[\mathsf{a_y}]^{\mathrm{T}} : \quad \exists \mathtt{v} {\in} \{0..9\}.\ \mathtt{x} \mapsto \mathtt{v}{+}1 * \mathtt{y} \mapsto \mathtt{v} \rightsquigarrow \mathtt{x} \mapsto \mathtt{v}{+}1 * \mathtt{y} \mapsto \mathtt{v}{+}1 \\
[\mathsf{a_z}]^{\mathrm{T}} : \quad \exists \mathtt{v} {\in} \{0..9\}.\ \mathtt{y} \mapsto \mathtt{v}{+}1 * \mathtt{z} \mapsto \mathtt{v} \rightsquigarrow \mathtt{y} \mapsto \mathtt{v}{+}1 * \mathtt{z} \mapsto \mathtt{v}{+}1
\end{cases}$$

The derivation in (8.1) involves a subtle interaction between the resources of the subjective view and its interference relation. In order to capture the interaction between the resources of the shared state ($\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0$) and their interference relation I, we apply the SHIFT principle to strengthen the actions with relevant knowledge of the shared state as follows. Consider the action of $[\mathsf{a_x}]^{\mathrm{T}}$ in I and the initial state with value 0 in all the cells. This action can be replaced by:

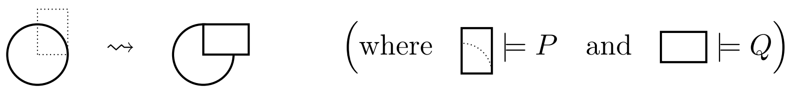
$$[\mathsf{a_x}]^{\mathrm{T}} : \exists \mathrm{v}{\in}\{0..9\}.\, \mathtt{x} \mapsto \mathrm{v} * \mathtt{y} \mapsto \mathrm{v} * \mathtt{z} \mapsto \mathrm{v} \rightsquigarrow \mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v} * \mathtt{z} \mapsto \mathrm{v}$$

This is possible because, as the programmer knows, whenever x and z have the same value, then y also has the same value which, under these conditions, is not changed by the actions in I. This amended action reflects stronger knowledge about when x can be incremented and how its value is related to those of y and z. As we justify formally in §9, by using the judgement $I \sqsubseteq^{P_0} I'_{\mathtt{y}}$ with $P_0 \triangleq \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0$, we can apply SHIFT to rewrite I as $I'_{\mathtt{y}}$. In order to establish the validity of the $I \sqsubseteq^{P_0} I'_{\mathtt{y}}$ shifting judgement, we appeal to a number of syntactic rules that reduce shifting judgements to standard separation logic entailments (see §9.5).

With the interference assertion rewritten to $I'_{\mathtt{y}}$, using the FORGET principle it is now safe to lose the z assertion to obtain the subjective view $\boxed{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0}_{I'_{\mathtt{y}}}$. This is because the new action of $[\mathsf{a_x}]^{\mathrm{T}}$ in $I'_{\mathtt{y}}$ retains enough information about how x, y and z are related. Since the action of $[\mathsf{a_z}]^{\mathrm{T}}$ only affects the z cell (now forgotten) leaving the cells x and y unaltered, we can use the SHIFT principle again to change the interference relation to $I_{\mathtt{y}} \triangleq I'_{\mathtt{y}} \backslash \mathsf{a_z}$, removing the action of $[\mathsf{a_z}]^{\mathrm{T}}$ from $I'_{\mathtt{y}}$. As we formally justify in §9, by using the judgement $I'_{\mathtt{y}} \sqsubseteq^{Q_0} I_{\mathtt{y}}$ with $Q_0 \triangleq \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0$, we can use the SHIFT principle to simplify $I'_{\mathtt{y}}$ to $I_{\mathtt{y}}$.

The interference assertion is now as simple as it can get, whilst retaining enough information about the connection between x, y and z. Finally, we stabilise the subjective view with respect to $I_{\mathtt{y}}$ and obtain our final precondition of $\mathbb{P}_{\mathtt{y}}$. To do this, we make critical use of the fact that the current thread *locally* holds the $\left[\mathsf{a_y}\right]^{\mathrm{T}}$ capability. That is, since the current thread *owns* $\left[\mathsf{a_y}\right]^{\mathrm{T}}$, no other thread in the environment may hold $\left[\mathsf{a_y}\right]^{\mathrm{T}}$ and thus the environment cannot perform the action of $\left[\mathsf{a_y}\right]^{\mathrm{T}}$. By contrast, the current thread does not hold the $[\mathsf{a_x}]^{\mathrm{T}}$ capability. As such, another thread in the environment may own $[\mathsf{a_x}]^{\mathrm{T}}$ and may perform its action.

Observe that the precondition of the $[\mathsf{a_y}]^{\mathrm{T}}$ action only requires requires the resources described by the subjective view. However, the precondition of the $[\mathsf{a_x}]^{\mathrm{T}}$ action depends on z, which is no longer included in the subjective view of the thread. If another thread in the environment owns the $[\mathsf{a_x}]^{\mathrm{T}}$ capability, it may perform its action whenever its subjective view is *compatible* with the precondition of the action. When that is the case, the piece of the state corresponding to the overlap between the state and the precondition of the action is removed, and the entire postcondition of the action is added in its place. Diagrammatically, a subjective state (represented by the circle) is affected as follows by an action $P \rightsquigarrow Q$:



This is because a subjective view describes a thread's partial knowledge about the shared state, while the environment may have additional knowledge to what the thread knows. In this case, while the thread does not have the capability to do the action of $[\mathsf{a_x}]^{\mathrm{T}}$, the environment might.

The proof of the specification of the thread $\mathbb{P}_{\mathsf{y}}$ is straightforward. By inspection, the invariant of the while loop is stable with respect to I_{y}. The atomic section allows safe manipulation of the contents of the subjective view. The final postcondition of $\mathbb{P}_{\mathsf{y}}$ follows from the invariant and the boolean expression of the while loop. We join up the postconditions of the threads using the Merge principle. Since $\vee$ distributes over $\uplus$, the subjective view simplifies to $\boxed{\mathsf{x} \mapsto 10 * \mathsf{y} \mapsto 10 * \mathsf{z} \mapsto 10}_{I_{\mathsf{x}} \cup I_{\mathsf{y}} \cup I_{\mathsf{z}}}$. Finally, since $I_{\mathsf{x}} \cup I_{\mathsf{y}} \cup I_{\mathsf{z}} \sqsubseteq^{P_{10}} I$ when $P_{10} \triangleq \mathsf{x} \mapsto 10 * \mathsf{y} \mapsto 10 * \mathsf{z} \mapsto 10$, using the Shift principle, we get the postcondition of $\mathbb{INC}$.

This concludes our CoLoSL proof of $\mathbb{INC}$. Our expansion and contraction of subjective views, in particular with shifting of interference assertions in key places, enables us to confine the specification and verification of each thread to just the resources they need. Such small specifications make proofs robust against changes to the environment of each thread, and thus provide more opportunities for proof reuse. We refer the reader to [47, 48] for two general variants of the $\mathbb{INC}$ program in Fig. 8.1. In the first instance, we generalise the $\mathbb{INC}$ program to a token ring comprising n threads. In the second instance, we consider a variant where the token ring may dynamically grow by spawning additional threads and appending them into the end of the ring. We then demonstrate that in contrast to existing CAP-like approaches [15, 10], the proof of the existing threads remain unchanged even in the face of such dynamic extension.

8.3. Comparison to CAP

CoLoSL is closely inspired by the program logic of CAP [15], improving on its formalism in several ways. To demonstrate this, we proceed with a pictorial description of our reasoning about a concurrent set module. We compare our CoLoSL reasoning with the original CAP reasoning in [15], demonstrating that our CoLoSL reasoning provides more concise proofs using our local reasoning about the shared state.

Consider the following illustration of the CAP set predicate in [15]:

$$\boxed{\overset{x}{v_1} \to \overset{y}{v_2} \to \overset{z}{v_3} \to \cdots \mathop{\circledast}_{(x,y)\notin S}\left(\mathop{\circledast}_{v}[\mathrm{U}(x,y,v)]\right) * \mathop{\circledast}_{(x,y)\in S}\left(\exists w.\ \mathop{\circledast}_{v\neq w}[\mathrm{U}(x,y,v)]\right)}^{s}_{I_x \cup I_y \cup I_z \cup \cdots}$$

The set is represented as a sorted singly-linked list with no duplicate elements. The list starts at address x with value v_1, points to the next element at address y with value v_2, and so forth. Hereafter, we write $\mathsf{node}(x, v, y)$ to denote a node at address x, with value v and successor y.

All nodes of the list reside in a single shared region labelled s and the interference on the list is the combined interference associated with each constituent node. Each node at a given address x is associated with a set of update capabilities of the form $[\mathrm{U}(x, y, v)]$ for *all* possible addresses y and *all* possible values v. This is to capture all potential successor addresses y and all potential values v that may be stored at address x. In order to modify a node, a thread can acquire the lock associated with the node and subsequently claim the relevant update capability.

Since in CAP the capabilities associated with a region can only be generated upon its creation, the shared region is required to keep track of all possible update capabilities $[\mathrm{U}(x, y, v)]$ associated with all addresses x (including those not currently in the domain of the list), all addresses y and all values v. At any one point, given $\mathsf{node}(x, v, y)$, the only update capability that can be claimed by a thread (through locking) is the one that reflects its current status, namely $[\mathrm{U}(x, y, v)]$. As a result, an auxiliary mathematical set S is used to track those nodes of the list that are currently locked and thus infer which $[\mathrm{U}]$ capabilities have been claimed. The distribution of update capabilities is captured by the two assertions written as the *infinite multiplicative star operator* $\circledast$. The first part of the assertion states that given any node at address x with successor y, if it is not locked (i.e. $(x, y) \notin S$), then all of its update capabilities of the form $[\mathrm{U}(x, y, v)]$ lie in the shared region for all values v. Dually, if it is locked (i.e. $(x, y) \in S$), then the update capabilities for all values v but one $(w \neq v)$ are in the shared region.

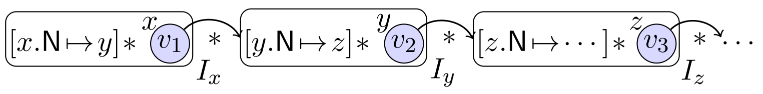
This CAP set predicate is unnecessarily complicated. It is counter-intuitive to have to account for the capabilities associated with addresses not in the domain of the list. Moreover, each thread observes all nodes in the list and thus needs to account for their associated interference.

The TaDA logic [10] took the first steps towards addressing the above shortcomings of CAP. TaDA regions are parametric in the PCM (partial commutative monoid) of capabilities (known as *guards* in TaDA). As such, one can choose a more suitable PCM to axiomatise the desired behaviour of capabilities. Later logics such as Iris [36] followed suit and allowed for the parameterisation of the capability (ghost resource) PCMs.

While the TaDA approach is much cleaner than that of CAP, it nevertheless requires the foresight of specifying all interference associated with the region upon its creation. As such, interference specifications are *static* and cannot be extended with new behaviour even when the existing resources are left untouched. By contrast, as well as being parametric in its capability PCM, interference specifications in CoLoSL are *dynamic* in that they may be extended with new behaviour using the EXTEND principle.

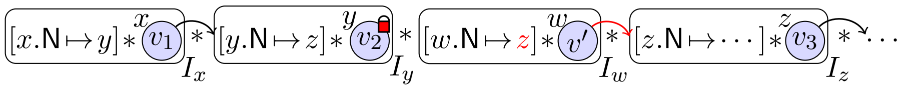
We proceed with the CoLoSL proof of the set implementation. As we demonstrate in §9, CoLoSL is parametric in the PCM of capabilities. We thus instantiate it with a heap-like capability separation algebra that is *stateful* and demonstrate that this allows for a more concise proof.

We specify the set predicate as the $*$-composition of the subjective views associated with each node in the singly-linked list as illustrated below:


$[x.\mathsf{N} \mapsto y] *$
x
v_1
$*$
I_x
$[y.\mathsf{N} \mapsto z] *$
y
v_2
$*$
I_y
$[z.\mathsf{N} \mapsto \cdots] *$
z
v_3
$* \cdots$
I_z

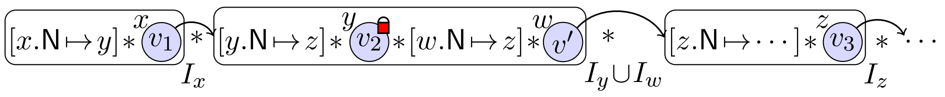
The interference on each subjective view is limited to the node in question. Associated with each node at address x is a "next" capability, $[x.\mathsf{N} \mapsto y]$, tracking its successor y. This is analogous to the $[\mathrm{U}(x, y, v)]$ capability of CAP and we shortly demonstrate how it is utilised in our reasoning.

Since CoLoSL allows for the *dynamic* extension of the shared state, we need not account for capabilities associated with all addresses. Instead, fresh capabilities are generated dynamically as needed. We demonstrate this by giving a reasoning outline of the `add`(v') method that adds value v' to the set by inserting it in the sorted list. Suppose $v_2 < v' < v_3$ and thus a new node w with value v' is to be inserted after node y. The operating thread proceeds by traversing the list by hand-over-hand locking until it reaches node y. It then locks y and claims its next pointer and moves it to its local state, as allowed by I_y. Subsequently, the shared state is *extended* by the resources associated with the new node and its associated capabilities ($[w.\mathsf{N} \mapsto z]$) are generated on the fly as illustrated below:



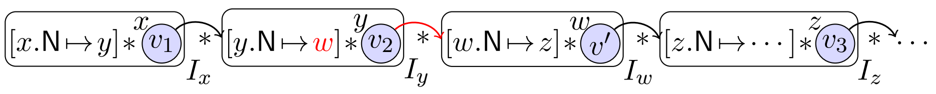
$[x.\mathsf{N} \mapsto y] * v_1$
x
I_x
$[y.\mathsf{N} \mapsto z] * v_2$
y
I_y
$[w.\mathsf{N} \mapsto z] * v'$
w
I_w
$[z.\mathsf{N} \mapsto \cdots] * v_3$
z
I_z
$* \cdots$

Since the locking thread holds the next pointer of y in its local state, it modifies it to point to the new node w. It then unlocks y and returns its next pointer to the shared state. When inserting a new node between y and z, the associated interference assertion I_y allows y to be unlocked only if it has been directed to a new node whose successor is z. As such, the unlocking thread must demonstrate that the new node w does indeed point to z. In order to establish this, we use the MERGE principle to combine the subjective views of y and w as follows:



$[x.\mathsf{N} \mapsto y] * v_1$
x
I_x
$[y.\mathsf{N} \mapsto z] * v_2 * [w.\mathsf{N} \mapsto z] * v'$
y
w
$I_y \cup I_w$
$[z.\mathsf{N} \mapsto \cdots] * v_3$
z
I_z
$* \cdots$

Finally, node y is unlocked; its next pointer is returned to the shared state and its next capability is modified to reflect its new successor. Using the COPY, FORGET and SHIFT principles in order, we obtain the set predicate with node w inserted into it:



$[x.\mathsf{N} \mapsto y] * v_1$
x
I_x
$[y.\mathsf{N} \mapsto w] * v_2$
y
I_y
$[w.\mathsf{N} \mapsto z] * v'$
w
I_w
$[z.\mathsf{N} \mapsto \cdots] * v_3$
z
I_z
$* \cdots$

We can reason about the set `remove` operation in a similar fashion. The dynamic extension afforded by the EXTEND principle allows us to generate new capabilities only when needed, resulting in a more concise specification and proof. Moreover, rather than having a distinct capability to modify the element at address x, for each possible successor address y (as with $[\mathrm{U}(x, y, v)]$ in CAP), we appeal to a single capability of the form $[x.\mathsf{N} \mapsto y]$ which is accordingly modified to $[x.\mathsf{N} \mapsto y']$ whenever the successor of x changes from y to y'. Lastly, using the reasoning principles of MERGE, FORGET, SHIFT and COPY, we can grow and shrink our subjective views as needed. Consequently, at any one point we only view the relevant parts of the shared state. We refer the reader to [47] for the full specification of the set operations as well as their respective proof sketches.

8.4. Comparison to Iris and Contemporary Logics

In comparison to contemporary program logics of [15, 54, 53, 38, 10, 36, 35], CoLoSL lacks several features such as abstract predicates [15, 54, 53, 38, 10, 36, 35], higher-order reasoning [54, 53, 36, 35] and abstract atomicity [10, 36]. These ideas suggest interesting directions and warrant further investigation.

Unlike CoLoSL however, the program logics of [19, 15, 54, 53, 10, 38] do not support the generalised interference manipulation afforded by the SHIFT principle for composing, framing and rewriting interference relations.

In the program logic of Iris [36, 35], one can use logical view shifts (analogous to that of the $\Rrightarrow$ repartitioning implication in CoLoSL) to manipulate the interference relations on a *per-example* basis. That is, using the view shifts in Iris, one can manipulate the interference relation for each program being verified (e.g. Dijkstra's token ring algorithm in §8.2). However, Iris does not feature a generalised view shift notion for interference manipulation as the necessary conditions for interference rewriting and framing have not been identified. By contrast, in CoLoSL we propose a *generalised* SHIFT principle where the conditions for rewriting and framing interference relations have been formalised via the $I \sqsubseteq^{P} I'$, providing an insight into the settings in which interference manipulation may be possible and beneficial.

9. Concurrent Local Subjective Logic

Concurrent local subjective logic (CoLoSL) is a general program logic for compositional reasoning about fine-grained concurrent algorithms. We present the general theory of CoLoSL and its various ingredients necessary for reasoning about concurrent algorithms.

In §9.1 we formally describe the underlying model of CoLoSL. We present the various ingredients necessary for defining the CoLoSL *worlds*: the building blocks of CoLoSL that track the resources held by each thread, the shared resources accessible to all threads, as well as the ways in which these shared resources may be manipulated by each thread.

In §9.2 we present the CoLoSL *assertions* and describe their semantics by relating them to sets of worlds. We then establish the validity of Copy, Forget and Merge principles introduced in the preceding chapters by establishing their truth for all possible worlds and interpretations.

In §9.3 we formally define the notion of *interference confinement* (P ⓒ I) necessary for extending the interference assertions via the Extend principle. Similarly, we formulate the definition of *interference shifting* ($I \sqsubseteq^{P} I'$) used for rewriting interference assertions. We then demonstrate the validity of the Shift principle by establishing its truth for all possible worlds and interpretations.

In §9.4 we present the CoLoSL *rely* and *guarantee* relations, describing how the shared state may be manipulated by the environment and the current thread, respectively. We then formulate the definition of the semantic implication $\Rrightarrow$ and subsequently demonstrate the validity of the Extend principle. We further formulate the definition of assertion *stability* defined in terms of the rely relation, namely, the possible actions taken by the environment.

In §9.5 we present several judgements that reduce the semantic checks needed for stability, interference confinement and interference shifting to separation logic entailments. These judgements do not contain subjective (boxed) assertions and solely include *local* assertions. As such, these judgements involve the familiar entailments of standard separation logic.

In §9.6 we present the programming language of CoLoSL. We build CoLoSL as an instance of the Views framework [14] and thus our programming language is that of Views, instantiated with an appropriate (parametric) choice of atomic operations. We then present the proof rules of CoLoSL, namely those of Views instantiated with the axiomatisation of the CoLoSL atomic operations.

In §9.7 we present the CoLoSL operational semantics via the small-step transition system of the Views framework [14], instantiated with the semantics of the atomic operations. We then demonstrate the soundness of the CoLoSL proof rules with respect to their operational semantics. As CoLoSL is built as an instance of Views, it suffices to demonstrate the soundness of its atomic operations with respect to their semantics.

The general theory of CoLoSL is parametric in several of its components which allows for its suitable instantiation depending on the concurrent program being verified. Following a similar style to that of preceding chapters, we delineate the parameters of CoLoSL enclosed in solid boxes labelled "CoLoSL Parameter".

9.1. CoLoSL Model

Worlds A CoLoSL *world* represents the underlying logical state tracking the resources held by each thread, the shared resources accessible to all threads, and the ways in which the shared resources may be manipulated by each thread. A world is a triple of the form $(l, g, \mathcal{I})$ where l and g are *instrumented states* and $\mathcal{I}$ is an *action model*. Let us explain the role of each component informally. The *local instrumented state* (or simply local state), l, represents the locally owned resources of a thread. The *shared instrumented state* (or simply shared state), g, represents the *entire* (global) shared state, accessible to all threads, subject to the interference described by the action model $\mathcal{I}$.

An action model is a partial function from *capabilities* to sets of *actions*. An action is a triple (p, q, c) of logical states where p and q are the *pre-* and *post-states* of the action, respectively, and c is the action *condition*. That is, the c acts as a mere catalyst for the action: it has to be present for the action to take effect, but is left unchanged by the action. Alternatively, the catalyst could be computed a posteriori for each action. However, we often need to isolate the part of the state that is modified by an action, hence our technical choice of recording the catalyst in the model. The *action model* $\mathcal{I}$ corresponds directly to the (semantic interpretation of) an interference assertion I. Although worlds do not put further constraints on the relationship between $\mathcal{I}$ and g, they are linked more tightly in the semantics of assertions (§9.2).

The composition of two worlds is defined whenever their local states are compatible and they agree on the other two components, hence have identical knowledge of the shared state and possible interferences.

We proceed by defining instrumented states, which constitute the notion of *resource* in CoLoSL, in the standard separation logic sense. Instrumented states have two components: one describes *logical states* (e.g. stacks and heaps); the other represents *capabilities.* The latter are inspired by the capabilities in deny-guarantee reasoning [18, 15]: a thread in possession of a given capability is allowed to perform its associated actions (as prescribed by the *action model* components of each world, defined below), whereas the actions of the capabilities *not* fully owned by a thread may be performed by the environment.

CoLoSL is parametric in the choice of the partial commutative monoid (hereafter simply monoid) representing the logical states and capabilities. This allows for suitable *instantiation* of CoLoSL depending on the programs being verified. For instance, in the token ring example $\mathbb{INC}$ of §8.2, the monoid of logical states is the standard variable stack of the variables-as-resource model [5], and the capabilities are captured by the $(\mathcal{P}(T), \uplus, \{\emptyset\})$ monoid where $T \triangleq \{\mathsf{a}_\mathtt{x}, \mathsf{a}_\mathtt{y}, \mathsf{a}_\mathtt{z}\}$ denotes a set of tokens. However, as we demonstrate in the examples of §10, our programs often call for a more complex model of logical states and capabilities. For instance, we may need our capabilities to be fractionally owned, where ownership of a *fraction* of a capability grants the right to perform the action to both the thread and the environment, whereas a fully-owned capability by the thread *denies* the environment the right to perform the associated action.

In general, logical states and capabilities can be instantiated as any partial commutative monoid [8] that satisfies the *cross-split* property [17]. We require the cross-split property to ensure the associativity of the overlapping conjunction connective $\circledast$. The cross-split property is a sufficient (but not necessary) condition for maintaining the associativity of $\circledast$.

Property 1 (Cross-split)**.** *A partial commutative monoid* $(\mathcal{M}, \bullet_{\mathcal{M}}, \textsc{Unit}_{\mathcal{M}})$ *satisfies the* cross-split property *if and only if:*

$$
\begin{aligned}
&\forall m_a, m_b, m_c, m_d \in \mathcal{M}.\ m_a \bullet_{\mathcal{M}} m_b = m_c \bullet_{\mathcal{M}} m_d \implies \\
&\quad \exists m_{ac}, m_{ad}, m_{bc}, m_{bd}.\ m_a = m_{ac} \bullet_{\mathcal{M}} m_{ad} \wedge m_b = m_{bc} \bullet_{\mathcal{M}} m_{bd} \\
&\qquad\qquad\qquad \wedge m_c = m_{ac} \bullet_{\mathcal{M}} m_{bc} \wedge m_d = m_{ad} \bullet_{\mathcal{M}} m_{bd}
\end{aligned}
$$

CoLoSL Parameter

Parameter 22 (Logical state partial commutative monoid)**.** Assume a set of *logical states* $h \in \textsc{LState}$.
Assume a *partial commutative monoid for logical states*, $\textsc{PCM}_{\textsc{l}} \triangleq (\textsc{LState}, \bullet_{\textsc{l}}, \textsc{Unit}_{\textsc{l}})$, satisfying the cross-split property (Prop. 1).

CoLoSL Parameter

Parameter 23 (Primitive capability partial commutative monoid)**.** Assume a set of *primitive capabilities* $\not c \in \textsc{Cap}$.
Assume a *partial commutative monoid for primitive capabilities*, $\textsc{PCM}_{\textsc{c}} \triangleq (\textsc{Cap}, \bullet_{\textsc{c}}, \textsc{Unit}_{\textsc{c}})$, satisfying the cross-split property (Prop. 1).

Recall that capabilities enable the manipulation of the shared state through their associated actions in the action models. At any one point a thread may unilaterally extend the shared state with some of its locally held resources (by the Extend principle). In doing so, it may also introduce new capabilities and actions that describe how the newly shared resources may be manipulated. In order to ensure the creation of *fresh* capabilities upon extension, the set of capabilities must be extensible. For instance, consider the following monoid where each element is a subset of the set $\mathcal{N} \triangleq \{1, 2, 3\}$, with composition defined as disjoint set union and a singleton unit set comprising the empty set.

$$\textsc{PCM}_{\mathcal{N}} \triangleq (\mathcal{P}(\{1,2,3\}), \uplus, \{\emptyset\})$$

The above monoid satisfies the cross-split property and thus meets the conditions stipulated in Par. 23. However, observe that since the carrier set $\mathcal{P}(\mathcal{N})$ is finite, it is possible to exhaust it to the point where no more fresh primitive capabilities can be allocated. For instance, when a thread holds the maximal primitive capability ȼ$=\{1, 2, 3\}$, it is not possible to allocate further non-unit capabilities as no non-unit element is compatible with ȼ. To remedy this, given the set of primitive capabilities $\textsc{Cap}$ in Par. 23, we define an extensible set of capabilities, $\kappa \in \textsc{Kap} \triangleq \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \textsc{Cap}$, defined as the set of finite functions from natural numbers to primitive capabilities in $\textsc{Cap}$. This way, since the domain of functions in $\textsc{Kap}$ are finite, we can always allocate fresh capabilities by picking a fresh *ticket* t not previously allocated. In the example above, we may initially generate the capability $\kappa{=}0 \mapsto \{1, 2, 3\}$ with ticket 0. Later when fresh capabilities are required, we can generate a fresh capability $\kappa'{=}1 \mapsto \{1, 2\}$ with ticket 1 and so forth. We define the composition of extensible capabilities $\bullet_{\text{K}}$ as function union with the values of tickets combined using the composition $\bullet_{\text{C}}$ on primitive capabilities. For instance, the $\kappa{=}0 \mapsto \{1, 2, 3\}$ in the example above can be split as $\kappa_1 \bullet_{\text{K}} \kappa_2$ with $\kappa_1{=}0 \mapsto \{1, 2\}$ and $\kappa_2{=}0 \mapsto \{3\}$.

Observe that this workaround is similar to the way in which allocation of fresh locations in a heap are ensured via the `alloc` operation. In the same way that heaps are modelled as finite functions from locations (normally modelled as natural numbers) to values, we model our extensible capabilities as finite functions from natural numbers to capabilities. However, while composition on heaps is defined as disjoint function union on the domain of heaps, we define our composition as function union on their domains with their ranges combined using the custom capability composition $\bullet_{\text{C}}$.

Note that in similar logics such as CAP [15] and iCAP [53], the freshness of new capabilities is ensured by generating a fresh region identifier r (generally modelled as natural numbers) upon region creation and associating the newly created capabilities with the fresh region identifier r.

Definition 90 (Capability monoid)**.** Given the partial commutative monoid of primitive capabilities $(\textsc{Cap}, \bullet_{\text{C}}, \textsc{Unit}_{\text{C}})$ in Par. 23, the set of *capabilities* is $\kappa \in \textsc{Kap} \triangleq \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \textsc{Cap}$.

The *capability composition*, $\bullet_{\text{K}} : \textsc{Kap} \times \textsc{Kap} \rightharpoonup \textsc{Kap}$, is defined as follows, for all $\kappa_1, \kappa_2 \in \textsc{Kap}$ and $t \in \mathbb{N}$:

$$(\kappa_1 \bullet_{\text{K}} \kappa_2)(t) \triangleq \begin{cases} \kappa_1(t) \bullet_{\text{C}} \kappa_2(t) & \text{if } t \in \mathit{dom}(\kappa_1) \text{ and } t \in \mathit{dom}(\kappa_2) \\ \kappa_1(t) & \text{if } t \in \mathit{dom}(\kappa_1) \text{ and } t \notin \mathit{dom}(\kappa_2) \\ \kappa_2(t) & \text{if } t \in \mathit{dom}(\kappa_2) \text{ and } t \notin \mathit{dom}(\kappa_1) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The *capability unit set* is $\textsc{Unit}_{\text{K}} \triangleq \{\mathbf{0}\}$ where $\mathbf{0}$ denotes a function with empty domain. The *partial commutative monoid of capabilities* is $\text{PCM}_{\text{K}} \triangleq (\textsc{Kap}, \bullet_{\text{K}}, \textsc{Unit}_{\text{K}})$.

Note that to distinguish the user-defined capabilities $\textsc{Cap}$ (supplied as a parameter in Par. 23) from the extensible capabilities $\textsc{Kap}$ (Def. 90), we refer to the former as *primitive capabilities* and the latter simply as *capabilities.*

We can now formalise the notion of *instrumented states.* As described above, an instrumented state is a pair comprising a logical state and a capability resource.

Definition 91 (Instrumented states)**.** Given the partial commutative monoid of logical states $(\textsc{LState}, \bullet_{\text{L}}, \textsc{Unit}_{\text{L}})$ in Par. 22 and the partial commutative monoid of capabilities $(\textsc{Kap}, \bullet_{\text{K}}, \textsc{Unit}_{\text{K}})$ in Def. 90, the set of *instrumented states* is: $l, g \in \textsc{IState} \triangleq \textsc{LState} \times \textsc{Kap}$.

Instrumented state composition, $\circ : \textsc{IState} \times \textsc{IState} \rightharpoonup \textsc{IState}$, is defined component-wise as $\circ \triangleq (\bullet_{\text{L}}, \bullet_{\text{K}})$ and is not defined when the composition on either component is undefined.

The *instrumented state unit set* is:

$$\textsc{Unit}_{\textsc{Ins}} \triangleq \{(h, \kappa) \mid h \in \textsc{Unit}_{\text{L}} \land \kappa \in \textsc{Unit}_{\text{K}}\}$$

The *partial commutative monoid of instrumented states* is: $\text{PCM}_{\textsc{Ins}} \triangleq (\textsc{IState}, \circ, \textsc{Unit}_{\textsc{Ins}})$.

Notationally, we write l (and its variants l', l_1, etc.) to range over either arbitrary instrumented states or those representing the local instrumented state. Similarly, we write g (and its variants g', g_1, etc.) to range over instrumented states when representing the shared (global) state. Given an instrumented state l, we write l_{L} and l_{K} for its first and second projections, respectively.

We often need to compare two instrumented states $l_1 \leq l_2$ (or their constituents: $h_1 \leq h_2$, $\kappa_1 \leq \kappa_2$) defined when there exists l such that $l \circ l_1 = l_2$. This is captured in the following definition.

Definition 92 (Ordering)**.** Given a partial commutative monoid $(\mathcal{M}, \bullet_{\mathcal{M}}, \textsc{Unit}_{\mathcal{M}})$, the *ordering relation*, $\leq\colon \mathcal{M} \times \mathcal{M}$, is defined as follows:

$$\leq \triangleq \{(m_1, m_2) \mid \exists m.\ m_1 \bullet_{\mathcal{M}} m = m_2\}$$

We write $m_1 \leq m_2$ for $(m_1, m_2) \in \leq$.

Given a monoid $(\mathcal{M}, \bullet_{\mathcal{M}}, \textsc{Unit}_{\mathcal{M}})$, in our formalisms we occasionally need to quantify over *compatible* elements of a monoid; that is, those elements that can be composed together by $\bullet_{\mathcal{M}}$. Similarly, we describe two elements of a monoid as *disjoint* when they do not overlap. We formulate these definitions below.

Definition 93 (Compatibility and disjointness)**.** Given a partial commutative monoid $(\mathcal{M}, \bullet_{\mathcal{M}}, \textsc{Unit}_{\mathcal{M}})$, the *compatibility relation*, $\sharp\colon \mathcal{M} \times \mathcal{M}$, is defined as follows:

$$\sharp \triangleq \{(m_1, m_2) \mid \exists m.\ m_1 \bullet_{\mathcal{M}} m_2 = m\}$$

Given the ordering relation $\leq$ (Def. 92), the *disjointness relation*, $\perp\colon \mathcal{M} \times \mathcal{M}$, is defined as follows:

$$\perp \triangleq \{(m_1, m_2) \mid \forall m \in \mathcal{M}.\ m \leq m_1 \land m \leq m_2 \Rightarrow m \in \textsc{Unit}_{\mathcal{M}}\}$$

We write $m_1 \sharp m_2$ for $(m_1, m_2) \in \sharp$ and write $m_1 \perp m_2$ for $(m_1, m_2) \in \perp$.

We proceed with the next ingredient of a CoLoSL world, *action models*. Recall that an action is simply a triple of instrumented states describing the pre- and post-states of the action, as well as the action condition (catalyst). Given an action (p, q, c) where p and q denote the action pre- and post-states and c denotes the catalyst, we require that c be maximal with respect to p and q in that p and q must not overlap: $p \perp q$. That is, the common parts between an action pre- and postcondition are captured by the catalyst c ensuring that p and q are disjoint.

An action model describes the set of actions associated with each capability. To ensure the generation of fresh capabilities upon extension, we additionally track the tickets allocated so far as part of the action model.

Definition 94 (Actions, action models)**.** Given the set of instrumented states IState (Def. 91), the set of *actions* is:

$$a \in \textsc{Action} \triangleq \left\{(p, q, c) \,\middle|\, p, q, c \in \textsc{IState} \land p \perp q\right\}$$

Given the set of capabilities Kap (Def. 90), the set of *action models* is:

$$\mathcal{I} \in \textsc{AMod} \triangleq \left(\textsc{Kap} \to \mathcal{P}(\textsc{Action})\right) \times (\mathbb{N} \overset{\text{fin}}{\rightharpoonup} \{1\})$$

Action model composition, $\cup : \textsc{AMod} \times \textsc{AMod} \rightharpoonup \textsc{AMod}$, is defined as follows, for all $\mathcal{I}, \mathcal{I}' \in \textsc{AMod}$, and $\kappa \in \textsc{Kap}$:

$$\mathcal{I} \cup \mathcal{I}' \triangleq \begin{cases} \left((\mathcal{I}_{\text{A}} \cup \mathcal{I}'_{\text{A}}), \mathcal{I}_{\text{T}}\right) & \text{if } \mathcal{I}_{\text{T}}{=}\mathcal{I}'_{\text{T}} \\ \text{undefined} & \text{otherwise} \end{cases} \quad \text{where} \;\; (\mathcal{I}_{\text{A}} \cup \mathcal{I}'_{\text{A}})(\kappa) \triangleq \mathcal{I}_{\text{A}}(\kappa) \cup \mathcal{I}'_{\text{A}}(\kappa)$$

The first component of an action model tracks the actions associated with capabilities; the second component tracks the tickets allocated so far. Given an action model $\mathcal{I}$, we write $\mathcal{I}_{\text{A}}$ and $\mathcal{I}_{\text{T}}$ for its first and second projections, respectively. For brevity, given a capability κ we write $\mathcal{I}(\kappa)$ for $\mathcal{I}_{\text{A}}(\kappa)$. We write $\mathbf{0}$ for an action model $\mathcal{I}$ with an empty domain in its first component (i.e. when $dom(\mathcal{I}_{\text{A}}) = \emptyset$).

The Effect of Actions Given a world $(l, g, \mathcal{I})$, since g represents the *entire* shared state, for $(l, g, \mathcal{I})$ to be well-formed, the actions in $\mathcal{I}$ must be *confined* to g. Let us elaborate on the necessity of the confinement condition.

Recall that a thread may unilaterally decide to introduce part of its local state into the shared state at any point (using the Extend principle). As such, confinement ensures that existing actions cannot affect future extensions of the shared state. Analogously, we require that the new actions introduced by the extension be confined to the extension in the same vein. This way, we ensure that extending the shared state cannot retroactively invalidate the views of other threads. However, as we demonstrate shortly, confinement does not prohibit *referring* to existing parts of the shared state in the new actions; rather, it only safeguards against *mutation* of the already shared resources through new actions.

Through confinement, we ensure that the *effect* of actions in the action models are contained to the shared state. In other words, given an action $a = (p, q, c)$ and a shared state g, whenever $p \circ c$ *agrees* with g, then p (the part of the state mutated by the action) is contained in g. Agreement of $p \circ c$ and g merely means that $p \circ c$ and g agree on the resources they have in common. Note that g only needs to contain p (and not $p \circ c$) for a to take effect. This relaxation is due to the fact that other threads may extend the shared state at a future point thus enabling the action. That is, the extension may provide the missing resources for $p \circ c$ to be contained in the shared state, thus allowing the extending thread to perform action a. Crucially, however, the part of the shared state mutated by the action, namely p, must be contained in g so that extensions of the shared state need not worry about existing actions interfering with new resources that were never shared beforehand.

Two elements of a monoid *agree* if they can be extended (via composition) to the same element. For instance the stack $\sigma_1{=}[\mathtt{x} \mapsto 0]$ agrees with both $\sigma_2{=}[\mathtt{y} \mapsto 1]$ and $\sigma_3{=}[\mathtt{x} \mapsto 0] \uplus [\mathtt{y} \mapsto 1]$ as they can all be extended to σ_3 (trivial extension in case of σ_3 itself). On the other hand, the σ_1 does not agree with $\sigma_4{=}[\mathtt{x} \mapsto 1]$ as no common super-stack can be found; that is, $\neg\exists\sigma.\ \sigma_1 \leq \sigma \wedge \sigma_4 \leq \sigma$.

Definition 95 (Agreement)**.** Given a partial commutative monoid $(\mathcal{M}, \bullet_{\mathcal{M}}, \textsc{Unit}_{\mathcal{M}})$ and the ordering relation $\leq$ (Def. 92), two elements $m_1, m_2 \in \mathcal{M}$ *agree*, written $\mathsf{agree}(m_1, m_2)$, if and only if:

$$\mathsf{agree}(m_1, m_2) \overset{\text{def}}{\iff} \exists m \in \mathcal{M}.\ m_1 \leq m \wedge m_2 \leq m$$

Definition 96 (Action confinement)**.** Given the set of actions $\textsc{Action}$ (Def. 94) and the set of instrumented states $\textsc{IState}$ (Def. 91), an action $a{=}(p, q, c) \in \textsc{Action}$ is *confined* to an instrumented state g, written $g \copyright\ a$, if and only if the following property is satisfied:

$$g \copyright\ a \overset{\text{def}}{\iff} \forall r.\ g \sharp r \wedge \mathsf{agree}(p \circ c, g) \Rightarrow p \leq g \wedge p \perp r$$

where $\sharp$ and $\perp$ respectively denote the compatibility and disjointness relations (Def. 93) and agree denotes the agreement relation (Def. 95).

As discussed earlier, only the action pre-state p (the part mutated by the action) is required to be contained in g and must be disjoint from all potential extensions (r) of the instrumented state g. That is, future extensions of g need not account for existing actions interfering with new resources.

Given a shared state g and an action model $\mathcal{I}$, we require that all actions of $\mathcal{I}$ be confined in all possible *futures* of g; that is, all shared states resulting from g after any number of applications of actions in $\mathcal{I}$. To this end, we define *action application* describing the effect of an action on an instrumented state. Observe that given an action a in $\mathcal{I}$, the g may not be affected by a when the intersection of g and the pre-state of a is an empty state in $\textsc{Unit}_{\textsc{Ins}}$. When this is the case, even though that action may potentially take place, it need not be accounted for as it leaves g unchanged. We thus introduce the notion of *visible actions* to describe those actions that affect (mutate) g.

Definition 97 (Action application)**.** Given the set of actions $\textsc{Action}$ (Def. 94) and the set of instrumented states $\textsc{IState}$ (Def. 91), the *application* of an action $a{=}(p, q, c) \in \textsc{Action}$ on an instrumented state $g \in \textsc{IState}$, written $a[g]$, is defined as follows:

$$a[g] \triangleq \left\{ q \circ l \,\middle|\, \mathsf{agree}(p \circ c, g) \wedge g{=}p \circ l \wedge q \sharp l \right\}$$

where $\sharp$ denotes the compatibility relation (Def. 93) and agree denotes the agreement relation (Def. 95).

An action $a \in \textsc{Action}$ is a *potential* action on an instrumented state g, written $\mathsf{potential}(a, g)$, if and only if $a[g]$ is not empty:

$$\mathsf{potential}(a, g) \overset{\text{def}}{\iff} a[g] \neq \emptyset$$

The $\mathsf{potential}(a, g)$ states that the action a may potentially take place on g (provided that g holds all the resources in the pre-state of a).

Definition 98 (Visible actions)**.** Given the set of actions $\textsc{Action}$ (Def. 94) and the set of instrumented states $\textsc{IState}$ (Def. 91), an action $a{=}(p, q, c) \in$

Action is *visible in g*, written $\mathsf{visible}(a, g)$, if and only if:

$$\mathsf{visible}(a, g) \overset{\text{def}}{\iff} \exists l.\, l \leq p \land l \leq g \land l \notin \textsc{Unit}_{\textsc{Ins}}$$

We are now ready to define our confinement condition on action models. Inspired by Local RG [19], we introduce the concept of locally fenced action models to capture all possible states reachable from the current state via a number of action applications. A set of states $\mathcal{F}$ *locally fences* an action model $\mathcal{I}$ if it is invariant under the interference perpetrated by the actions in $\mathcal{I}$. An action model is then confined to a logical state l if it can be fenced by a set of states that includes l. In the following we write $rng(f)$ to denote the *range* of a function f.

Definition 99 (Locally-fenced action model)**.** Given the set of action models AMod (Def. 94) and the set of instrumented states IState (Def. 91), an action model $\mathcal{I} \in$ AMod is *locally fenced* by $\mathcal{F} \in \mathcal{P}(\textsc{IState})$, written $\mathcal{F} \blacktriangleright \mathcal{I}$, if and only if:

$$\mathcal{F} \blacktriangleright \mathcal{I} \overset{\text{def}}{\iff} \forall g \in \mathcal{F}.\, \forall a \in rng(\mathcal{I}_{\text{A}}).\, g \copyright a \land (a[g] \subseteq \mathcal{F})$$

where © denotes action confinement (Def. 96) and $a[g]$ and $\mathsf{potential}(a, g)$ are as defined in Def. 97.

Definition 100 (Action model confinement)**.** Given the set of instrumented states IState (Def. 91) and the set of action models AMod (Def. 94), an action model $\mathcal{I} \in$ AMod is *confined* to an instrumented state $g \in$ IState, written $g \copyright \mathcal{I}$, if and only if:

$$g \copyright \mathcal{I} \overset{\text{def}}{\iff} \exists \mathcal{F} \in \mathcal{P}(\textsc{IState}).\, gl \in \mathcal{F} \land \mathcal{F} \blacktriangleright \mathcal{I}$$

where $\blacktriangleright$ denotes the local fencing relation (Def. 99).

Action model confinement is lifted to sets of instrumented states where given a set of instrumented states $S \in \mathcal{P}(\textsc{IState})$, an action model $\mathcal{I} \in$ AMod is *confined to the set of states S*, written $S \copyright \mathcal{I}$, if and only if:

$$\exists \mathcal{F} \in \mathcal{P}(\textsc{IState}).\, S \subseteq \mathcal{F} \land \mathcal{F} \blacktriangleright \mathcal{I}$$

An important property of the confinement relation is that it is preserved under composition. That is, whenever the confinement relation holds for two different states under two different action models (i.e. $g_1 \copyright \mathcal{I}_1$ and $g_2 \copyright \mathcal{I}_2$ hold), then the confinement relation also holds for the combined states under the combined action models (i.e. $g_1 \circ g_2 \copyright \mathcal{I}_1 \cup \mathcal{I}_2$ also hold). This is captured in the following lemma. Later, we appeal to this lemma to establish the validity of the Extend principle.

Lemma 3 (Confinement monotonicity)**.** *For all* $g_1, g_2 \in \text{IState}$ *(Def. 91) and* $\mathcal{I}_1, \mathcal{I}_2 \in \text{AMod}$ *(Def. 94):*

$$g_1 \copyright \mathcal{I}_1 \land g_2 \copyright \mathcal{I}_2 \implies g_1 \circ g_2 \copyright \mathcal{I}_1 \cup \mathcal{I}_2$$

Proof. The full proof is given in §C (Lemma 32) □

We are almost in a position to define *well-formedness* of worlds. Recall that a thread may independently extend the shared state with its locally held resources and introduce new actions and capabilities by claiming a *fresh* capability ticket t. In order to ensure the freshness of the ticket t, as part of the well-formedness of a world $w{=}(l, g, \mathcal{I})$, we require that the capability tickets found in the local state l and the shared state g be accounted for in $\mathcal{I}$. That is, $dom\big((l \circ g)_{\text{K}}\big) \subseteq dom(\mathcal{I}_{\text{T}})$.

We can now formalise the notion of well-formedness. A world $(l, g, \mathcal{I})$ is well-formed if l and g are compatible, the $\mathcal{I}$ is confined to g, and the capability tickets of l and g are contained in $\mathcal{I}$.

Definition 101 (Well-formedness)**.** Given the set of instrumented states IState (Def. 91) and the set of action models AMod (Def. 94), a triple $(l, g, \mathcal{I}) \in \text{IState} \times \text{IState} \times \text{AMod}$ is *well-formed*, written $\mathsf{wf}\,(l, g, \mathcal{I})$, if and only if:

$$\mathsf{wf}\,(l, g, \mathcal{I}) \overset{\text{def}}{\iff} l \sharp g \land g \copyright \mathcal{I} \land dom\big((l \circ g)_{\text{K}}\big) \subseteq dom(\mathcal{I}_{\text{T}})$$

where $\sharp$ denotes the compatibility relation (Def. 93) and $\copyright$ denotes action model confinement (Def. 100).

Definition 102 (Worlds)**.** Given the set of instrumented states IState (Def. 91) and the set of action models AMod (Def. 94), the set of *worlds*, $w \in \text{World}$, is defined as follows:

$$\text{World} \triangleq \{w \in \text{IState} \times \text{IState} \times \text{AMod} \mid \mathsf{wf}\,(w)\}$$

where $\mathsf{wf}\,(.)$ is as given in Def. 101.

World composition, $\bullet : \textsc{World} \times \textsc{World} \rightharpoonup \textsc{World}$, is defined as follows:

$$(l, g, \mathcal{I}) \bullet (l', g', \mathcal{I}') \triangleq \begin{cases} (l \circ l', g, \mathcal{I}) & \text{if } g = g', \text{ and } \mathcal{I} = \mathcal{I}' \\ & \text{and } \mathsf{wf}\,(l \circ l', g, \mathcal{I}) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

The *world unit set* is defined as follows:

$$\textsc{Unit}_{\text{W}} \triangleq \left\{ (l, g, \mathcal{I}) \,\middle|\, (l, g, \mathcal{I}) \in \textsc{World} \land l \in \textsc{Unit}_{\textsc{Ins}} \right\}$$

The *partial commutative monoid of worlds* is: $\text{PCM}_{\text{W}} \triangleq (\textsc{World}, \bullet, \textsc{Unit}_{\text{W}})$.

9.2. CoLoSL Assertions

Our assertions extend standard assertions from separation logic with *subjective views* and *capability assertions*. We assume an infinite set of *logical variables*, LVar, and a set of *logical environments*, LEnv, mapping logical variables onto their values.

Recall that CoLoSL is parametric with respect to the logical states (Par. 22) and primitive capabilities (Par. 23). As such, CoLoSL is also parametric in the logical states assertions and primitive capability assertions which may be instantiated with any assertion language interpreted over logical states LState and primitive capabilities Cap, respectively. We thus require that CoLoSL be supplied with the satisfiability relations for logical states assertions and primitive capability assertions. This is captured by the parameters below.

Recall from 8 that we write $\mathsf{sat}(\mathcal{C}_1, \mathcal{C}_2)$ to denote that the primitive capability assertions $\mathcal{C}_1$ and $\mathcal{C}_2$ are *always satisfiable.* That is, there exist primitive capabilities c_1 and c_2 such that c_1 satisfies $\mathcal{C}_1$, the c_2 satisfies $\mathcal{C}_2$ and the composition of c_1 and c_2 is defined (i.e. $\exists c.\ c_1 \bullet_{\text{C}} c_2 = c$).

CoLoSL Parameter

Parameter 24 (Logical state assertions)**.** Assume a set of *primitive logical state assertions* $\mathcal{H} \in$ LSAst.

Given the set of logical environments and the set of logical states LState (Par. 22), assume a satisfiability relation for the primitive logical state assertions:

$$\models_{\mathrm{L}} : (\mathrm{LEnv} \times \mathrm{LState}) \times \mathrm{LSAst}$$

CoLoSL Parameter

Parameter 25 (Primitive capability assertions)**.** Assume a set of *primitive capability assertions* $\mathcal{C} \in \mathrm{CAst}$.
Given the set of logical environments and the set of primitive capabilities Cap (Par. 23), assume a satisfiability relation for the primitive capability assertions:

$$\models_{\mathrm{C}} : (\mathrm{LEnv} \times \mathrm{Cap}) \times \mathrm{CAst}$$

Given the partial commutative monoid of primitive capabilities $(\mathrm{Cap}, \bullet_{\mathrm{C}}, \mathrm{Unit}_{\mathrm{C}})$ in Par. 23, the primitive capability assertions $\mathcal{C}_1$ and $\mathcal{C}_2$ are *always satisfiable*, written $\mathsf{sat}(\mathcal{C}_1, \mathcal{C}_2)$, if and only if:

$$\begin{aligned} \mathsf{sat}(\mathcal{C}_1, \mathcal{C}_2) \overset{\text{def}}{\iff} & \forall \Gamma \in \mathrm{LEnv}.\ \exists c, c_1, c_2 \in \mathrm{Cap}. \\ & c = c_1 \bullet_{\mathrm{C}} c_2 \land \Gamma, c_1 \models_{\mathrm{C}} \mathcal{C}_1 \land \Gamma, c_2 \models_{\mathrm{C}} \mathcal{C}_2 \end{aligned}$$

Definition 103 (CoLoSL assertions)**.** Given the logical state assertions LSAst (Par. 24) and the primitive capability assertions CAst (Par. 25), the *local CoLoSL assertions*, $p, q \in \mathrm{LAst}$, are defined by the following grammar, where $\mathcal{H} \in \mathrm{LSAst}$, $\mathcal{C} \in \mathrm{CAst}$ and $\mathrm{x}, \mathrm{T} \in \mathrm{LVar}$ denote logical variables:

$$\begin{aligned} p, q ::= & \ \mathsf{false} \mid p \Rightarrow q \mid \exists \mathrm{x}.\ p \mid \mathsf{emp} \mid \mathcal{H} \mid [\mathcal{C}]^{\mathrm{T}} \\ & \mid p * q \mid p \mathrel{-\!\!*} q \mid p \uplus q \mid p \mathrel{-\!\!\circledast} q \end{aligned}$$

The CoLoSL *assertions*, $P, Q \in \mathrm{Ast}$, and the *interference assertions*, $I \in \mathrm{IAst}$, are defined by the following grammars, where $p, q, r \in \mathrm{LAst}$ and $\mathrm{x}, \bar{\mathrm{Y}} \in \mathrm{LVar}$ denote logical variables:

$$\begin{aligned} P, Q &::= p \mid \exists \mathrm{x}.\ P \mid P \lor Q \mid P * Q \mid P \uplus Q \mid \boxed{P}_I \\ I &::= \emptyset \mid \{ r : \exists \bar{\mathrm{Y}}.\ p \rightsquigarrow q \} \cup I \end{aligned}$$

The syntax of local assertions ($p, q \in \textsc{LAst}$) is that of standard separation logic, extended with the $\uplus$ and $-\circledast$ connectives, described shortly. Local assertions are interpreted over the partial commutative monoid of instrumented states ($\textsc{IState}, \circ, \textsc{Unit}_{\textsc{Ins}}$) in Def. 91. The classical assertions are interpreted in the usual way. Other classical connectives (e.g. $\wedge, \vee, \neg, \forall$) can be derived in the standard way. The emp is true for the units of instrumented states in $\textsc{Unit}_{\textsc{Ins}}$. A logical state assertion $\mathcal{H}$ describes instrumented states of the form (h, κ) where h satisfies $\mathcal{H}$ (as described by the $\models_{\textsc{l}}$ relation in Par. 24) and $\kappa \in \textsc{Unit}_{\textsc{K}}$ (Def. 90). Analogously, a capability assertions $[\mathcal{C}]^{\textsc{t}}$ describes instrumented states of the form (h, κ) where $h \in \textsc{Unit}_{\textsc{L}}$ (Par. 22), and κ describes a capability where its ticket is denoted by $\textsc{t}$ and its value satisfies $\mathcal{C}$ (as described by the $\models_{\textsc{c}}$ relation in Par. 25). Often in our examples, when we do not extend the shared state and do not generate additional capabilities, we drop the ticket component of capability assertions for brevity and write e.g. $[\mathcal{C}]$ instead. This may be interpreted as the capability assertion $\mathcal{C}$ with the *default* (initial) ticket 0.

The $p * q$ describes an instrumented state that can be split (via the instrumented state composition operator $\circ$) into two substates satisfying p and q. The $-\!\!*$ connective is the right adjunct of $*$ and thus we have $p * (p -\!\!* q) \Rightarrow q$. An instrumented state l satisfies $p -\!\!* q$ if and only if for any state l' satisfying p, the combined state $l \circ l'$ satisfies q. The $\uplus$ connective is the *overlapping conjunction* or "sepish" [21, 50]. An instrumented state satisfies $p \uplus q$ if and only if it can be split into three substates (via the $\circ$ composition) such that the composition of the first two states satisfies p and the composition of the last two satisfies q. Lastly, the $-\circledast$ is the *existential magic wand* or "septraction" [57]. An instrumented state l satisfies $p -\circledast q$ if and only if there exists a state l' satisfying p, such that the combined state $l \circ l'$ satisfies q.

The syntax of assertions ($P, Q \in \textsc{Ast}$) is that of standard separation logic, with the exception of the subjective views $\boxed{P}_I$. Assertions are interpreted over the separation algebra of worlds ($\textsc{World}, \bullet, \textsc{Unit}_{\textsc{W}}$) in Def. 102. The local assertion p describes worlds of the form $(l, g, \mathcal{I})$ where l satisfies the local assertion p as described above. The classical assertions are interpreted in the usual way. The $P * Q$ and $P \uplus Q$ assertions are interpreted over worlds in an analogous manner to that described above. That is, a world w satisfies $P * Q$ if and only if it can be split into two worlds (via the $\bullet$ operator) satisfying P and Q. *Mutatis mutandis* for $P \uplus Q$. We have omitted the $-\!\!*$ and $-\!\!\circledast$ connectives from the syntax of assertions in $\textsc{Ast}$ (we have $-\!\!*$ and $-\!\!\circledast$ at the level of local assertions and not the top-level assertions) as we do not need them in our examples. It is however straightforward to extend the syntax of $\textsc{Ast}$ with these connectives, interpreted in an analogous manner to that of local assertions.

A subjective view $\boxed{P}_I$ describes worlds of the form $(l, g, \mathcal{I})$ where $l \in \textsc{Unit}_{\textsc{Ins}}$ and a state s can be found such that i) $g = s \circ r$ for some *context* (frame) r; ii) s satisfies P in the standard separation logic sense; and iii) I and $\mathcal{I}$ *agree* given the decomposition $s \circ r$, in the following sense:

(1) every action in I is empreflected in $\mathcal{I}$ (defined shortly);

(2) every action in $\mathcal{I}$ that is potentially enabled in g and has a visible effect on s is reflected in I;

(3) the above hold after any number of $\mathcal{I}$ action applications on g.

These conditions will be captured by the *action model closure* relation $\mathcal{I}\!\downarrow (s, r, \mathcal{I}')$ given by the upcoming Def. 107 (where $\mathcal{I}'$ denotes the interpretation of the interference assertion I). When the above conditions are met, we refer to s as a *subjective* state.

The semantics of CoLoSL assertions is given by a satisfiability relation $\Gamma, w \models P$ between a logical environment $\Gamma \in \textsc{LEnv}$, a world $w{=}(l, g, \mathcal{I})$ and an assertion P. We use two auxiliary satisfiability relations. The first one $\Gamma, l \vDash_{\text{SL}} P$ interprets assertions in the usual separation logic sense over an instrumented state l. We use this relation when we are only concerned with the local component of a world (l) and wish to ignore the shared component (g) and the action model ($\mathcal{I}$). The second one $\Gamma, s \models_{g,\mathcal{I}} P$ interprets assertions over a subjective state s that is part of the global shared state g, subject to action model $\mathcal{I}$. This third form of satisfaction is needed to deal with nesting of subjective views. We often write $\models_{\dagger}$ as a shorthand for $\models_{g,\mathcal{I}}$ when we do not need to refer to the individual components g and $\mathcal{I}$.

Note that this presentation with several satisfiability relations differs from the usual CAP presentation [15], where assertions are first interpreted over worlds that are not necessarily well-formed and are then cut down to well-formed ones. The CAP presentation strays from separation logic models in some respects. For instance, in CAP, the $-\!\!*$ is not the right adjunct of $*$. Although we have omitted $-\!\!*$ from the syntax of top-level assertions in Ast, its semantics in CoLoSL would be standard and would satisfy the adjunction with $*$.

Interference assertions are interpreted component-wise with each action interpreted as a set of actions of the form $(l_p, l_q, l_c) \in \textsc{Action}$ with pre- and post-states l_p and l_q and catalyst l_c. Note that the definition of actions (Def. 94) stipulates that l_c be maximal and that l_p and l_q not overlap. Lastly, observe that an interference assertion is of the form $\{r : \exists\bar{\textsc{y}}.\, p \rightsquigarrow q\}$ where $r \in \textsc{LAst}$. This is to allow for more expressive interference assertions such as when $r \triangleq [\mathcal{C}_1]^{\textsc{t}_1} * [\mathcal{C}_2]^{\textsc{t}_2}$ or $r \triangleq [\mathcal{C}_1]^{\textsc{t}_1} \vee [\mathcal{C}_2]^{\textsc{t}_2}$. However, as we stipulate in the following definition, an interference assertion $I \triangleq \{r : \exists\bar{\textsc{y}}.\, p \rightsquigarrow q\}$ is only meaningful when the interpretation of r describes a capability κ in Kap; otherwise the semantics of I is empty (equivalent to false).

Definition 104 (Assertion semantics)**.** Given the logical environments LEnv, the instrumented states IState (Def. 91) and the assertions Ast (Def. 103), the *local satisfiability relation*, $\vDash_{\textsc{sl}}$: $(\textsc{LEnv} \times \textsc{IState}) \times \textsc{Ast}$, is defined as follows, for all $\Gamma \in \textsc{LEnv}$ and $l \in \textsc{IState}$:

$\Gamma, l \vDash_{\textsc{sl}} \mathsf{false}$		never
$\Gamma, l \vDash_{\textsc{sl}} p \Rightarrow q$	iff	$\Gamma, l \vDash_{\textsc{sl}} p$ implies $\Gamma, l \vDash_{\textsc{sl}} q$
$\Gamma, l \vDash_{\textsc{sl}} \exists \textsc{x}.\, P$	iff	$\exists v.\, [\Gamma \mid \textsc{x} : v], l \models P$
$\Gamma, l \vDash_{\textsc{sl}} \mathsf{emp}$	iff	$l \in \textsc{Unit}_{\textsc{Ins}}$
$\Gamma, l \vDash_{\textsc{sl}} \mathcal{H}$	iff	$\exists h, \kappa.\, l{=}(h, \kappa)$ and $\Gamma, h \models_{\textsc{l}} \mathcal{H}$ and $\kappa \in \textsc{Unit}_{\textsc{k}}$
$\Gamma, l \vDash_{\textsc{sl}} [\mathcal{C}]^{\textsc{t}}$	iff	$\exists h, \kappa, t.\, l{=}(h, \kappa)$ and $h \in \textsc{Unit}_{\textsc{l}}$ and $\Gamma(\textsc{t}){=}t$ and $dom(\kappa){=}\{t\}$ and $\Gamma, \kappa(t) \models_{\textsc{c}} \mathcal{C}$
$\Gamma, l \vDash_{\textsc{sl}} p -\!\!* q$	iff	$\forall l'.\, \Gamma, l' \vDash_{\textsc{sl}} p$ and $l \sharp l'$ implies $\Gamma, l \circ l' \vDash_{\textsc{sl}} q$
$\Gamma, l \vDash_{\textsc{sl}} p -\!\!\circledast\, q$	iff	$\exists l'.\, \Gamma, l' \vDash_{\textsc{sl}} p$ and $\Gamma, l \circ l' \vDash_{\textsc{sl}} q$
$\Gamma, l \vDash_{\textsc{sl}} P_1 * P_2$	iff	$\exists l_1, l_2.\, l{=}l_1 \circ l_2$ and $\Gamma, l_1 \vDash_{\textsc{sl}} P_1$ and $\Gamma, l_2 \vDash_{\textsc{sl}} P_2$

$$
\begin{array}{lll}
\Gamma, l \vDash_{\mathrm{SL}} P_1 \mathbin{\ast\!\!\!\!\cup} P_2 & \text{iff} & \exists l', l_1, l_2.\ l{=}l' \circ l_1 \circ l_2 \text{ and} \\
 & & \Gamma, l' \circ l_1 \vDash_{\mathrm{SL}} P_1 \text{ and } \Gamma, l' \circ l_2 \vDash_{\mathrm{SL}} P_2 \\
\Gamma, l \vDash_{\mathrm{SL}} P \vee Q & \text{iff} & \Gamma, l \vDash_{\mathrm{SL}} P \text{ or } \Gamma, l \vDash_{\mathrm{SL}} Q \\
\Gamma, l \vDash_{\mathrm{SL}} \boxed{P}_I & \text{iff} & l \in \textsc{Unit}_{\textsc{Ins}}
\end{array}
$$

Given the set of worlds $\textsc{World}$ (Def. 102) and assertions $\textsc{Ast}$ (Def. 103), the *satisfiability relation*, $\models$: $(\textsc{LEnv} \times \textsc{World}) \times \textsc{Ast}$, is defined as follows, where $(\!| I |\!)_\Gamma$ denotes the interpretation of I defined below, and the definition of $\mathcal{I}{\downarrow}\,(s, r, \mathcal{I}')$ is given in Def. 107:

$$
\begin{array}{lll}
\Gamma, (l, g, \mathcal{I}) \models p & \text{iff} & \Gamma, l \vDash_{\mathrm{SL}} p \\
\Gamma, w \models \exists \mathrm{x}.\ P & \text{iff} & \exists v.\ [\Gamma \mid \mathrm{x} : v], w \models P \\
\Gamma, w \models P \vee Q & \text{iff} & \Gamma, w \models P \text{ or } \Gamma, w \models Q \\
\Gamma, w \models P_1 * P_2 & \text{iff} & \exists w_1, w_2.\ w{=}w_1 \bullet w_2 \text{ and} \\
 & & \Gamma, w_1 \models P_1 \text{ and } \Gamma, w_2 \models P_2 \\
\Gamma, w \models P_1 \mathbin{\ast\!\!\!\!\cup} P_2 & \text{iff} & \exists w', w_1, w_2.\ w{=}w' \bullet w_1 \bullet w_2 \text{ and} \\
 & & \Gamma, w' \bullet w_1 \models P_1 \text{ and } \Gamma, w' \bullet w_2 \models P_2 \\
\Gamma, (l, g, \mathcal{I}) \models \boxed{P}_I & \text{iff} & l \in \textsc{Unit}_{\mathrm{L}} \text{ and } \exists s, r, \mathcal{I}'.\ g{=}s \circ r \text{ and } \mathcal{I}'_{\mathrm{A}}{=}\,(\!| I |\!)_\Gamma \\
 & & \text{and } \Gamma, s \models_{g,\mathcal{I}} P \text{ and } \mathcal{I}{\downarrow}\,\big(s, r, \mathcal{I}'\big)
\end{array}
$$

where

$$
\begin{array}{lll}
\Gamma, s \models_{g,\mathcal{I}} p & \text{iff} & \Gamma, s \vDash_{\mathrm{SL}} p \\
\Gamma, s \models_{\dagger} \exists \mathrm{x}.\ P & \text{iff} & \exists v.\ [\Gamma \mid \mathrm{x} : v], s \models_{\dagger} P \\
\Gamma, s \models_{\dagger} P \vee Q & \text{iff} & \Gamma, s \models_{\dagger} P \text{ or } \Gamma, s \models_{\dagger} Q \\
\Gamma, s \models_{\dagger} P_1 * P_2 & \text{iff} & \exists s_1, s_2.\ s{=}s_1 \circ s_2 \text{ and} \\
 & & \Gamma, s_1 \models_{\dagger} P_1 \text{ and } \Gamma, s_2 \models_{\dagger} P_2 \\
\Gamma, s \models_{\dagger} P_1 \mathbin{\ast\!\!\!\!\cup} P_2 & \text{iff} & \exists s', s_1, s_2.\ s{=}s' \circ s_1 \circ s_2 \text{ and} \\
 & & \Gamma, s' \circ s_1 \models_{\dagger} P_1 \text{ and } \Gamma, s' \circ s_2 \models_{\dagger} P_2 \\
\Gamma, s \models_{g,\mathcal{I}} \boxed{P}_I & \text{iff} & \Gamma, (s, g, \mathcal{I}) \models \boxed{P}_I
\end{array}
$$

Given the set of interference assertions $\textsc{IAst}$ (Def. 103), the set of capabilities $\textsc{Kap}$ (Def. 90) and the set of actions $\textsc{Action}$ (Def. 94), the *interference* *interpretation function*, $\llparenthesis . \rrparenthesis_{(.)} : \textsc{IAst} \times \textsc{LEnv} \to (\textsc{Kap} \to \mathcal{P}(\textsc{Action}))$, is defined as follows, for all $\kappa \in \textsc{Kap}$:

$$\llparenthesis \emptyset \rrparenthesis_{\Gamma}(\kappa) \triangleq \emptyset$$

$$\llparenthesis \{r : \exists \bar{\mathrm{Y}}. p \rightsquigarrow q\} \cup I \rrparenthesis_{\Gamma}(\kappa) \triangleq \left\{ (l_p, l_q, l_c) \;\middle|\; \begin{array}{l} (l_p, l_q, l_c) \in \textsc{Action} \\ \wedge\, \Gamma, \kappa \vDash_{\mathrm{SL}} r \,\wedge\, \exists \bar{v}. \\ \quad [\Gamma \mid \bar{\mathrm{Y}}{:}\bar{v}], l_p \circ l_c \vDash_{\mathrm{SL}} p \\ \quad \wedge\; [\Gamma \mid \bar{\mathrm{Y}}{:}\bar{v}], l_q \circ l_c \vDash_{\mathrm{SL}} q \end{array} \right\} \cup \llparenthesis I \rrparenthesis_{\Gamma}(\kappa)$$

The $\vDash_{\mathrm{SL}}$ is the weakest of all three satisfiability relations. More concretely, if $\Gamma, (l, g, \mathfrak{I}) \models P$ then $\Gamma, l \vDash_{\mathrm{SL}} P$. Similarly, if $\Gamma, l \models_{g,\mathfrak{I}} P$ then $\Gamma, l \vDash_{\mathrm{SL}} P$. This is formalised in the following lemma.

Lemma 4 (satisfiability relations)**.** *Given the instrumented states* $\textsc{IState}$ *(Def. 91), action models* $\textsc{AMod}$ *(Def. 94), assertions* $\textsc{Ast}$ *(Def. 103) and the satisfiability relations* $\vDash_{\mathrm{SL}}$, $\models_{\dagger}$ *and* $\models$ *(Def. 104), for all* $P \in \textsc{Ast}$, $\Gamma \in \textsc{LEnv}$, $l, g \in \textsc{IState}$ *and* $\mathfrak{I} \in \textsc{AMod}$ *(Def. 94):*

$$\textit{if} \quad \Gamma, l \models_{g,\mathfrak{I}} P \quad \textit{then} \quad \Gamma, l \vDash_{\mathrm{SL}} P \tag{9.1}$$

$$\textit{if} \quad \Gamma, l \models P \qquad \textit{then} \quad \Gamma, l \vDash_{\mathrm{SL}} P \tag{9.2}$$

Proof. The full proof is given in §C (in Lemmata 24 and 25, respectively).

Action Model Closure Let us now turn to the definition of action model closure, as informally introduced at the beginning of this section. First, we need to revisit the effect of actions to take into account the splitting of the global shared state into a subjective state s and a context r.

Definition 105 (Combined action application)**.** Given the set of instrumented states $\textsc{IState}$ (Def. 91) and the set of actions $\textsc{Action}$ (Def. 94), the *combined application* of an action $a{=}(p, q, c) \in \textsc{Action}$ on an instrumented subjective state $s \in \textsc{IState}$ and context $r \in \textsc{IState}$, written $a[s, r]$, is defined as follows:

$$\left\{ (q \circ s', r') \;\middle|\; \begin{array}{l} \mathsf{agree}(s \circ r, p \circ c) \wedge \exists p_s, p_r. \\ \quad p{=}p_s \circ p_r \wedge p_s {\notin} \textsc{Unit}_{\textsc{Ins}} \wedge s{=}p_s \circ s' \wedge r{=}p_r \circ r' \wedge q \sharp s' \circ r' \end{array} \right\}$$
$$\cup \left\{ (s, q \circ r') \;\middle|\; \mathsf{agree}(s \circ r, p \circ c) \wedge r{=}p \circ r' \wedge q \sharp s \circ r' \right\}$$

where $\sharp$ denotes the compatibility relation (Def. 93) and agree denotes the agreement relation (Def. 95).

Observe that the combined action application $a[s, r]$ and action application $a[s \circ r]$ (Def. 97) are linked in the following way:

$$\forall (s', r') \in a[s, r].\ s' \circ r' \in a[s \circ r]$$

In our informal description of action model closure on p. 274 we stated that a set of actions must be *reflected* in an action model. Intuitively, an action is reflected in an action model if for every state in which the action can take place, the action model includes an action with a similar effect that can also occur in that state. In other words, an action $a{=}(p, q, c)$ is reflected in $\mathcal{I}$ from a state l if whenever a is enabled in an extension of l (i.e. $p \circ c \leq l \circ r$ for some extension r), then there exists an action $a'{=}(p, q, c') \in \mathcal{I}$ (with the same pre- and post states p and q) that is also enabled in the same extension of l (i.e. $p \circ c' \leq l \circ r$). We proceed with the definition of action reflection.

Definition 106 (Action reflection)**.** Given the set of instrumented states IState (Def. 91) and the set of actions Action (Def. 94), an action $a{=}(p, q, c) \in$ Action is *reflected* in a set of actions $A \in \mathcal{P}(\text{Action})$ from an instrumented state $l \in$ IState, written $\mathsf{reflected}(a, l, A)$, if and only if:

$$\forall r.\ p \circ c \leq l \circ r \Rightarrow \exists c'.\ (p, q, c') \in A \wedge p \circ c' \leq l \circ r$$

We now formally define action model closure. Action model closure constitutes the crux of the Shift principle. For each condition outlined on p. 274, we annotate which part of the definition implements them.

Definition 107 (Action model closure)**.** Given the set of instrumented states IState (Def. 91) and the set of action models AMod (Def. 94), an action model $\mathcal{I} \in$ AMod is *closed* under a subjective state $s \in$ IState, context $r \in$ IState, and action model $\mathcal{I}' \in$ AMod, written $\mathcal{I}{\downarrow}\,(s, r, \mathcal{I}')$, if and only if:

$$\forall n \in \mathbb{N}.\ \mathcal{I}{\downarrow_n}\,\left(s, r, \mathcal{I}'\right)$$

where

$$
\begin{aligned}
\mathcal{I}{\downarrow_0} \left(s, r, \mathcal{I}'\right) &\overset{\text{def}}{\iff} \mathsf{true} \\
\mathcal{I}{\downarrow_{n+1}} \left(s, r, \mathcal{I}'\right) &\overset{\text{def}}{\iff} \forall \kappa.\ \forall a \in \mathcal{I}'(\kappa).\ \mathsf{reflected}(a, s \circ r, \mathcal{I}(\kappa)) \wedge & (1)\\
&\qquad \forall \kappa.\ \forall a \in \mathcal{I}(\kappa).\ \mathsf{potential}(a, s \circ r) \Rightarrow \\
&\qquad\qquad \big(\mathsf{reflected}(a, s \circ r, \mathcal{I}'(\kappa)) \vee \neg \mathsf{visible}(a, s)\big) & (2)\\
&\qquad\qquad \wedge \forall (s', r') \in a[s, r].\ \mathcal{I}{\downarrow_n} \left(s', r', \mathcal{I}'\right) & (3)
\end{aligned}
$$

and the definitions of potential, visible and reflected are as given in Def. 97, Def. 98 and Def. 106, respectively.

Informally, given a global shared state $g \triangleq s \circ r$ comprising a subjective state s and a context r, the $\mathcal{I} \downarrow (s, r, \mathcal{I}')$ states that the actions of $\mathcal{I}'$ *simulate* those of $\mathcal{I}$ starting from the shared state g. More concretely, the $\mathcal{I}{\downarrow} (s, r, \mathcal{I}')$ states that the $\mathcal{I}$ is closed under $(s, r, \mathcal{I}')$ if the closure relation holds for any number of steps $n \in \mathbb{N}$ where

- s denotes the subjective view of the shared state;
- r denotes the context;
- $s \circ r$ captures the entire shared state;
- a step corresponds to the occurrence of an action as prescribed in $\mathcal{I}$ which may or may not be found in $\mathcal{I}'$.

The indexed closure relation is satisfied trivially for no steps (n=0). On the other hand, for an arbitrary $n \in \mathbb{N}$ the relation holds if and only if for any action a in $\mathcal{I}$, where a is potentially enabled in $s \circ r$, then:

1. every action in $\mathcal{I}'$ is reflected in $\mathcal{I}$ (*cf.* item (1) on p. 274); and
2. for every action a in $\mathcal{I}$, where a is potentially enabled in $s \circ r$, then
   a) *either* a is reflected in $\mathcal{I}'$ (a is known to the subjective view);
   b) *or* a does not affect the subjective state s; that is, a is not visible in s (*cf.* item (2) on p. 274); and
3. $\mathcal{I}$ is closed under any subjective state s' and context r' resulting from the application of a; that is, $(s', r') \in a[s, r]$ (*cf.* item (3) on p. 274)

Note that when a is not visible in s (2b), given any $(s', r') \in a[s, r]$, from the definition of action application we then know $s' = s$.

Recall from the semantics of assertions (Def. 104) that the actions in $\mathcal{I}'$ correspond to the interpretation of an interference assertion I on a subjective view (i.e. $\llparenthesis I \rrparenthesis_\Gamma$ given a logical environment Γ). The first condition ensures that the subjective actions in $\mathcal{I}'$ are contained in those of $\mathcal{I}$. As such, from the semantics of subjective assertions, the actions in $\mathcal{I}$ represent the superset of all interferences known to subjective views.

We make further observations about this definition. First, item (3) makes our assertions robust with respect to future extensions of the shared state, where potential actions may become enabled using additional catalyst that is not immediately present. Second, the $\mathcal{I}'$ (and thus interference assertions) need not reflect actions that have no visible effect on the subjective state.

Third, if an action model is closed under a subjective state $s_1 \circ s_2$ and context r, then it is also closed under the smaller subjective state s_1, and the larger context extended with the forgotten state $(s_2 \circ r)$. This is formalised in the lemma below (part a). We appeal to this lemma in establishing the validity of the Forget principle.

Fourth, whenever an action model closure relation holds for two different subjective states (that may overlap), subject to two different action models, then the closure relation also holds for the combined subjective states and their associated action models. This is captured in the following lemma (part b). We appeal to this lemma in establishing the validity of the Merge principle.

Lastly, given an existing action model $\mathcal{I}$ and an extension action model $\mathcal{I}_e$, whenever i) the actions of the existing $\mathcal{I}$ are confined to an existing global state g (i.e. g © $\mathcal{I}$ holds); and ii) the actions of the extension $\mathcal{I}_e$ are confined to an extension state s_e (i.e. s_e © $\mathcal{I}_e$ holds), then the combined action model $\mathcal{I} \cup \mathcal{I}_e$ is closed under the extended state s_e, the global state g and the extension action model $\mathcal{I}_e$ (i.e. $\mathcal{I} \cup \mathcal{I}_e \downarrow (s_e, g, \mathcal{I}_e)$ also holds). This is formalised in the lemma below (part c). We appeal to this lemma in establishing the validity of the Extend principle.

Lemma 5 (Forget, Merge and Extend closure)**.** *For all* $s_1, s_2, r \in$

IState *(Def. 91) and* $\mathcal{I}, \mathcal{I}' \in$ AMod *(Def. 94):*

$$\mathcal{I}{\downarrow}\left(s_1 \circ s_2, r, \mathcal{I}'\right) \implies \mathcal{I}{\downarrow}\left(s_1, s_2 \circ r, \mathcal{I}'\right) \tag{a}$$

For all $s_p, s_c, s_q, r \in$ IState *(Def. 91) and* $\mathcal{I}, \mathcal{I}_1, \mathcal{I}_2 \in$ AMod *(Def. 94):*

$$\begin{gathered}\mathcal{I}{\downarrow}\left(s_p \circ s_c, s_q \circ r, \mathcal{I}_1\right) \;\wedge\; \mathcal{I}{\downarrow}\left(s_q \circ s_c, s_p \circ r, \mathcal{I}_2\right) \implies \\ \mathcal{I}{\downarrow}\left(s_p \circ s_c \circ s_q, r, \mathcal{I}_1 \cup \mathcal{I}_2\right)\end{gathered} \tag{b}$$

For all $g, s_e \in$ IState *(Def. 91) and* $\mathcal{I}, \mathcal{I}, \mathcal{I}_e \in$ AMod *(Def. 94):*

$$g \textcircled{c}\, \mathcal{I} \wedge s_e \textcircled{c}\, \mathcal{I}_e \implies \mathcal{I} \cup \mathcal{I}_e{\downarrow}\left(s_e, g, \mathcal{I}_e\right) \tag{c}$$

Proof. The proof of all three parts are given in §C (Lemma 28, Lemma 29 and Lemma 33, respectively).

This completes the definition of assertion semantics. We can now show that the logical principles of CoLoSL are valid. The proofs of the Shift and Extend principles are delayed until §9.3 and §9.4, where we formalise the definitions of $\sqsubseteq$ and $\Rrightarrow$.

Lemma 6 (Copy, Forget and Merge validity)**.** *For all* $P, Q \in$ Ast *and* $I, I' \in$ IAst *(Def. 103):*

$$\vdash \boxed{P}_I \overset{\textsc{Copy}}{\Longrightarrow} \boxed{P}_I * \boxed{P}_I \tag{9.3}$$

$$\vdash \boxed{P * Q}_I \overset{\textsc{Forget}}{\Longrightarrow} \boxed{P}_I \tag{9.4}$$

$$\vdash \boxed{P}_{I_1} * \boxed{Q}_{I_2} \overset{\textsc{Merge}}{\Longrightarrow} \boxed{P \mathbin{\cup\!\!\!\!\ast} Q}_{I_1 \cup I_2} \tag{9.5}$$

Proof (9.3). This is immediate from the semantics of CoLoSL assertions (Def. 104).

Proof (9.4). It suffices to show that for all $\Gamma \in$ LEnv:

$$\left\{w \mid \Gamma, w \models \boxed{P \mathbin{\cup\!\!\!\!\ast} Q}_I\right\} \subseteq \left\{w \mid \Gamma, w \models \boxed{P}_I\right\}$$

We proceed as follows:

$$\left\{w \,\middle|\, \Gamma, w \models \boxed{P \mathbin{\cup\!\!\!\!\ast} Q}_I\right\}$$

$$
\begin{aligned}
&= \left\{ (l, (s \circ r), \mathcal{I}) \;\middle|\; \begin{array}{l} l \in \textsc{Unit}_{\textsc{Ins}} \wedge \exists \mathcal{I}'.\ \mathcal{I}'{=}((\!|I|\!)_\Gamma, \mathcal{I}_{\mathrm{T}}) \\ \wedge\, \exists s_p, s_c, s_q.\ s{=}s_p \circ s_c \circ s_q \\ \qquad \wedge\, \Gamma, (s_p \circ s_c) \models_{(s \circ r), \mathcal{I}} P \\ \qquad \wedge\, \Gamma, (s_q \circ s_c) \models_{(s \circ r), \mathcal{I}} Q \\ \qquad \wedge\, \mathcal{I}\!\downarrow (s_p \circ s_c \circ s_q, r, \mathcal{I}') \end{array} \right\} \\
\text{(Lemma 5(a))} &\subseteq \left\{ (l, (s \circ r), \mathcal{I}) \;\middle|\; \begin{array}{l} l \in \textsc{Unit}_{\textsc{Ins}} \wedge \exists \mathcal{I}'.\ \mathcal{I}'{=}((\!|I|\!)_\Gamma, \mathcal{I}_{\mathrm{T}}) \\ \wedge\, \exists s_p, s_c, s_q.\ s{=}s_p \circ s_c \circ s_q \\ \qquad \wedge\, \Gamma, (s_p \circ s_c) \models_{(s \circ r), \mathcal{I}} P \\ \qquad \wedge\, \Gamma, (s_q \circ s_c) \models_{(s \circ r), \mathcal{I}} Q \\ \qquad \wedge\, \mathcal{I}\!\downarrow (s_p \circ s_c, s_q \circ r, \mathcal{I}') \end{array} \right\} \\
&\subseteq \left\{ (l, (s_p \circ r), \mathcal{I}) \;\middle|\; \begin{array}{l} l \in \textsc{Unit}_{\textsc{Ins}} \wedge \exists \mathcal{I}'.\ \mathcal{I}'{=}((\!|I|\!)_\Gamma, \mathcal{I}_{\mathrm{T}}) \\ \quad \wedge\, \Gamma, s_p \models_{(s_p \circ r), \mathcal{I}} P \wedge \mathcal{I}\!\downarrow (s_p, r, \mathcal{I}') \end{array} \right\} \\
&= \left\{ w \;\middle|\; \Gamma, w \models \boxed{P}_I \right\}
\end{aligned}
$$

as required. □

Proof (9.5). It suffices to show that for all $\Gamma \in \text{LE}\textsc{nv}$:

$$
\left\{ w \mid \Gamma, w \models \boxed{P}_{I_1} * \boxed{Q}_{I_2} \right\} \subseteq \left\{ w \mid \Gamma, w \models \boxed{P \uplus Q}_{I_1 \cup I_2} \right\}
$$

We then proceed as follows:

$$
\begin{aligned}
& \left\{ w \mid \Gamma, w \models \boxed{P}_{I_1} * \boxed{Q}_{I_2} \right\} \\
&= \left\{ (l, (s \circ r), \mathcal{I}) \;\middle|\; \begin{array}{l} l \in \textsc{Unit}_{\textsc{Ins}} \wedge \exists\, \mathcal{I}_1, \mathcal{I}_2, s_p, s_c, s_q. \\ \mathcal{I}_1{=}((\!|I_1|\!)_\Gamma, \mathcal{I}_{\mathrm{T}}) \wedge \mathcal{I}_2{=}((\!|I_2|\!)_\Gamma, \mathcal{I}_{\mathrm{T}}) \\ \wedge\, s{=}s_p \circ s_c \circ s_q \\ \wedge\, \Gamma, (s_p \circ s_c) \models_{(s \circ r), \mathcal{I}} P \\ \wedge\, \Gamma, (s_q \circ s_c) \models_{(s \circ r), \mathcal{I}} Q \\ \wedge\, \mathcal{I}\!\downarrow (s_p \circ s_c, s_q \circ r, \mathcal{I}_1) \\ \wedge\, \mathcal{I}\!\downarrow (s_c \circ s_q, s_p \circ r, \mathcal{I}_2) \end{array} \right\} \\
\text{(Lemma 5(b))} &\subseteq \left\{ (l, (s \circ r), \mathcal{I}) \;\middle|\; \begin{array}{l} l \in \textsc{Unit}_{\textsc{Ins}} \wedge \exists\, \mathcal{I}_1, \mathcal{I}_2, s_p, s_c, s_q. \\ \mathcal{I}_1{=}((\!|I_1|\!)_\Gamma, \mathcal{I}_{\mathrm{T}}) \wedge \mathcal{I}_2{=}((\!|I_2|\!)_\Gamma, \mathcal{I}_{\mathrm{T}}) \\ \wedge\, s{=}s_p \circ s_c \circ s_q \\ \wedge\, \Gamma, (s_p \circ s_c) \models_{(s \circ r), \mathcal{I}} P \\ \wedge\, \Gamma, (s_q \circ s_c) \models_{(s \circ r), \mathcal{I}} Q \\ \wedge\, \mathcal{I}\!\downarrow (s_p \circ s_c \circ s_q, r, \mathcal{I}_1 \cup \mathcal{I}_2) \end{array} \right\}
\end{aligned}
$$

$$=\left\{ (l, (s \circ r), \mathfrak{I}) \;\middle|\; \begin{array}{l} l \in \textsc{Unit}_{\textsc{Ins}} \wedge \exists \mathfrak{I}', s_p, s_c, s_q. \\ \mathfrak{I}' = (\langle\!| I_1 \cup I_2 |\!\rangle_\Gamma, \mathfrak{I}_\top) \\ \wedge\, s = s_p \circ s_c \circ s_q \\ \wedge\, \Gamma, (s_p \circ s_c) \models_{(s \circ r), \mathfrak{I}} P \\ \wedge\, \Gamma, (s_q \circ s_c) \models_{(s \circ r), \mathfrak{I}} Q \\ \wedge\, \mathfrak{I}\!\downarrow (s_p \circ s_c \circ s_q, r, \mathfrak{I}') \end{array} \right\}$$

$$= \left\{ w \mid \Gamma, w \models \boxed{P \uplus Q}_{I_1 \cup I_2} \right\}$$

as required. □

Note that as shown below, the version of Forget where P and Q predicates are conjoined using $\uplus$ (rather than $*$) is also valid for all P, Q, and I, where the first implication follows from the semantics of $\uplus$.

$$\boxed{P \uplus Q}_I \Rightarrow \boxed{P * \mathsf{true}}_I \overset{\textsc{Forget}}{\Rightarrow} \boxed{P}_I$$

9.3. Interference Manipulations

We now proceed with formalising the requirements of the Extend and Shift principles.

Shared State Extension When extending the shared state with locally owned resources, one may specify a new interference assertion over these newly shared resources. While in CoLoSL the new interference may refer to parts of the shared state beyond the newly added resources (in particular the existing shared state), they must not allow visible updates to those parts, so as not to invalidate other threads' views of existing resources. We thus require that the actions of the newly introduced interference be confined (Def. 96) to the extension and not interfere with the existing shared resources. We first motivate this constraint with an example.

Example 11. Let P defined below describe the view of the current thread. Since the current thread owns the location addressed by x locally, it may extend the shared state with x as described by Q. In extending the shared state, the current thread also extended the interference allowed on the shared state by adding a new action associated with the newly generated capability resource [a], as given in I', which updates the value of location x.

$$P \triangleq \text{x} \mapsto 1 * \boxed{\text{y} \mapsto 1 \vee \text{y} \mapsto 2}_{I} \qquad I \triangleq ([\mathsf{b}]^{-} : \text{y} \mapsto 1 \rightsquigarrow \text{y} \mapsto 2)$$
$$Q \triangleq \exists \text{T}.\ [\mathsf{a}]^{\text{T}} * \boxed{(\text{y} \mapsto 1 \vee \text{y} \mapsto 2) * \text{x} \mapsto 1}_{I \cup I'} \qquad I' \triangleq ([\mathsf{a}]^{\text{T}} : \text{x} \mapsto 1 \rightsquigarrow \text{x} \mapsto 2)$$

Since location x was previously held locally by the current thread and was hence not visible to other threads, this new action will not invalidate their view of the shared state and thus this extension is valid. On the other hand, if I' is replaced with $I'' \triangleq ([\mathsf{a}]^{\text{T}} : \text{y} \mapsto 1 \rightsquigarrow \text{y} \mapsto 3)$, allowing for the mutation of location y, this would potentially invalidate the views of other threads. Indeed, other threads may rely on the fact that the only updates allowed on location y are done through some $[\mathsf{b}]^{-}$ capability as specified in I. As such, this new behaviour would invalidate their view of the shared state.

In order to ensure sound extension of the shared state, we require in Extend that the newly introduced interferences be confined to the locally owned resources (e.g. in Example 11 above we require $\text{x} \mapsto 1 \circledcirc I'$).

Definition 108 (Interference confinement)**.** Given the action model confinement relation ⓒ (Def. 96), the set of logical environments LEnv, the sets of assertions Ast and interference assertions IAst (Def. 103), the $\vDash_{\text{SL}}$ satisfiability relation and the $\langle\!|.|\!\rangle_{(.)}$ interpretation function (Def. 104), an interference assertion $I \in$ IAst is *confined* to an assertion $P \in$ Ast, written $P \circledcirc I$, if and only if:

$$\forall \Gamma \in \text{LEnv}.\ \{l \mid \Gamma, l \vDash_{\text{SL}} P\} \circledcirc \langle\!|I|\!\rangle_{\Gamma}$$

The $P \circledcirc I$ states that the actions in the interference assertion I are confined to the states described by P, defined as a straightforward lift of action model confinement (Def. 96) to assertions. For instance, consider the interference assertion I from Fig. 8.1 in §8, repeated below:

$$I \triangleq \begin{cases} [\mathsf{a_x}]^{\text{T}} : \quad \exists \text{v} \in \{0 \cdots 9\}.\ \mathsf{x} \mapsto \text{v} * \mathsf{z} \mapsto \text{v} \rightsquigarrow \mathsf{x} \mapsto \text{v}+1 * \mathsf{z} \mapsto \text{v} \\ [\mathsf{a_y}]^{\text{T}} : \exists \text{v} \in \{0 \cdots 9\}.\ \mathsf{x} \mapsto \text{v}+1 * \mathsf{y} \mapsto \text{v} \rightsquigarrow \mathsf{x} \mapsto \text{v}+1 * \mathsf{y} \mapsto \text{v}+1 \\ [\mathsf{a_z}]^{\text{T}} : \exists \text{v} \in \{0 \cdots 9\}.\ \mathsf{y} \mapsto \text{v}+1 * \mathsf{z} \mapsto \text{v} \rightsquigarrow \mathsf{y} \mapsto \text{v}+1 * \mathsf{z} \mapsto \text{v}+1 \end{cases}$$

We then have $P_0 \circledcirc I$ with $P_0 \triangleq \mathsf{x} \mapsto 0 * \mathsf{y} \mapsto 0 * \mathsf{z} \mapsto 0$. Observe that establishing $P_0 \circledcirc I$ requires a semantic (model-level) check. However, as we demonstrate later in §9.5, rather than checking interference confinement semantically, we present a number of syntactic judgements that reduce interference confinement to logical entailments. Using these judgements, we then demonstrate that $P_0 \copyright I$ holds (see Example 12 on p. 299).

We delay the proof of the EXTEND principle until §9.4 where we formalise the definition of $\Rrightarrow$.

Action Shifting Recall from §8 that interference assertions may shrink over time by forgetting actions that are either redundant or not relevant to the current subjective view via *shifting.* In order to ensure the soundness of interference shifting, we must ensure that the forgotten actions are irrelevant not only for the current subjective view, but also for all possible futures of the subjective view under all potential actions, both from the thread and the environment. To capture this set of possible futures, we refine our notion of local fences (Def. 99), which is defined in the context of the global shared state, to consider a subjective state within the global shared state instead. To do this, we also need to refine our notion of action application from Def. 105 to ignore the context of a subjective state, which as far as a subjective view is concerned could be anything.

Definition 109 (Subjective action application)**.** Given the set of instrumented states ISTATE (Def. 91) and the set of actions ACTION (Def. 94), the *subjective application* of an action $a \in$ ACTION on an instrumented state $s \in$ ISTATE, written $a(s)$, is defined as follows:

$$a(s) \triangleq \left\{(s', r') \,\middle|\, \exists r \in \text{ISTATE}.\ s \sharp r \land (s', r') \in a[s, r]\right\}$$

where $\sharp$ denotes the compatibility relation (Def. 93) and $a[s, r]$ denotes the combined action application (Def. 105).

Note that in contrast with global action application $a[g]$ in Def. 97, only *parts* of the action pre-state has to intersect with the subjective view s for $a(s)$ to apply. Thus, we fabricate a context r that is compatible with the subjective view and satisfies the rest of the pre-state.

Definition 110 (Fenced action model)**.** Given the set of instrumented states ISTATE (Def. 91) and the set of action models AMOD (Def. 94), an action model $\mathcal{I} \in$ AMOD is *fenced* by $\mathcal{F} \in \mathcal{P}(\text{ISTATE})$, written $\mathcal{F} \rhd \mathcal{I}$, if and only if:

$$\forall l \in \mathcal{F}.\ \forall a \in rng(\mathcal{I}).\ \forall (s, r) \in a(l).\ \exists r' \leq r.\ s \circ r' \in \mathcal{F}$$

Given the set of logical environments LEnv, the sets of assertions Ast and interference assertions IAst (Def. 103), the $\models_{\text{SL}}$ satisfiability relation and the $\langle\!|.|\!\rangle_{(.)}$ interpretation function (Def. 104), an interference assertion $I \in$ IAst is *fenced* by an assertion $P \in$ Ast, written $P \rhd I$, if and only if:

$$\forall \Gamma \in \text{LEnv}.\ \{l \mid \Gamma, l \models_{\text{SL}} P\} \rhd \langle\!| I |\!\rangle_{\Gamma}$$

Observe that the definition of fencing is subsumed by that of local fencing (Def. 99). That is, for all fences $\mathcal{F}$ and action models $\mathcal{I}$, the $\mathcal{F} \blacktriangleright \mathcal{I}$ implies $\mathcal{F} \rhd \mathcal{I}$. In particular, note that in contrast with local fences, fences do not require that actions be confined inside the subjective state. The $P \rhd I$ states that the interference assertion I is fenced by assertion P, defined simply as the lifting of the fencing relation $\rhd$ to assertions. For instance, consider the interference assertion I'_{y} from §8, repeated below:

$$I'_{\mathsf{y}} \triangleq \begin{cases} [\mathsf{a_x}]^{\mathrm{T}} : \exists \mathrm{v} {\in} \{0 \cdots 9\}.\ \mathsf{x} \mapsto \mathrm{v} * \mathsf{y} \mapsto \mathrm{v} * \mathsf{z} \mapsto \mathrm{v} \ \rightsquigarrow\ \mathsf{x} \mapsto \mathrm{v}{+}1 * \mathsf{y} \mapsto \mathrm{v} * \mathsf{z} \mapsto \mathrm{v} \\ [\mathsf{a_y}]^{\mathrm{T}} : \qquad \exists \mathrm{v} {\in} \{0 \cdots 9\}.\ \mathsf{x} \mapsto \mathrm{v}{+}1 * \mathsf{y} \mapsto \mathrm{v} \ \rightsquigarrow\ \mathsf{x} \mapsto \mathrm{v}{+}1 * \mathsf{y} \mapsto \mathrm{v}{+}1 \\ [\mathsf{a_z}]^{\mathrm{T}} : \qquad \exists \mathrm{v} {\in} \{0 \cdots 9\}.\ \mathsf{y} \mapsto \mathrm{v}{+}1 * \mathsf{z} \mapsto \mathrm{v} \ \rightsquigarrow\ \mathsf{y} \mapsto \mathrm{v}{+}1 * \mathsf{z} \mapsto \mathrm{v}{+}1 \end{cases}$$

A possible fence for I'_{y} is denoted by the local assertion F_{y} below ($F_{\mathsf{y}} \rhd I'_{\mathsf{y}}$):

$$F_{\mathsf{y}} \triangleq \bigvee_{\mathrm{w}=0}^{10} (\mathrm{x} \mapsto \mathrm{w} * \mathrm{y} \mapsto \mathrm{w}) \vee (\mathrm{x} \mapsto \mathrm{w}{+}1 * \mathrm{y} \mapsto \mathrm{w})$$

Note that establishing $F_{\mathsf{y}} \rhd I'_{\mathsf{y}}$ requires a semantic check. Later in §9.5 we present a number of syntactic judgements that reduce fencing checks to logical entailments. Using these judgements, we then demonstrate that $F_{\mathsf{y}} \rhd I'_{\mathsf{y}}$ holds (see Example 14 on p. 307).

Fencing vs. Stability The keen-eyed reader may have noticed that the definition of fencing is rather similar to the well-known notion of *stability* under an interference. However, as we demonstrate later, the definition of stability in CoLoSL is slightly weaker than that of fencing.That is, given a set of states $\mathcal{F}$ and an action model $\mathcal{I}$, if $\mathcal{I}$ is fenced by $\mathcal{F}$ (i.e. $\mathcal{F} \rhd \mathcal{I}$ holds), then $\mathcal{F}$ is also stable with respect to the actions of $\mathcal{I}$. However, the other direction does not generally hold. We refer the reader to §9.5 where we illustrate this with an example on page 312.

Definition 111 (Action model shifting)**.** Given the set of instrumented states IState (Def. 91) and the set of action models AMod (Def. 94), an action model $\mathcal{I}' \in$ AMod is a *shifting* of $\mathcal{I} \in$ AMod with respect to $S \in \mathcal{P}(\text{IState})$, written $\mathcal{I} \sqsubseteq^{S} \mathcal{I}'$, if and only if there exists a fence $\mathcal{F} \in \mathcal{P}(\text{IState})$ such that:

$$\begin{aligned}&\mathcal{I}_{\text{T}}{=}\mathcal{I}'_{\text{T}} \land S \subseteq \mathcal{F} \land \mathcal{F} \rhd \mathcal{I} \land \forall l \in \mathcal{F}.\ \forall \kappa.\\ &\quad\quad \forall a \in \mathcal{I}'(\kappa).\ \mathsf{reflected}(a, l, \mathcal{I}(\kappa))\\ &\quad\quad\ \land\ \forall a \in \mathcal{I}(\kappa).\ a(l) \neq \emptyset \Rightarrow (\neg\mathsf{visible}(a, l) \lor \mathsf{reflected}(a, l, \mathcal{I}'(\kappa)))\end{aligned}$$

where $\mathsf{reflected}$ and $\mathsf{visible}$ are as defined in Defs. 106 and 98, the $a(l)$ denotes subjective action application (Def. 109) and $\rhd$ is the fencing relation (Def. 110).

An important property of the action model shifting is that it preserves action model closure (Def. 107). That is, if an action model $\mathcal{I}$ is closed under a subjective state s, context r and action model $\mathcal{I}_1$ (i.e. $\mathcal{I}{\downarrow}(s, r, \mathcal{I}_1)$ holds), and $\mathcal{I}_1$ is shifted to $\mathcal{I}_2$ with respect to a set of states containing s (i.e. $\mathcal{I}_1 \sqsubseteq^{\{s\}} \mathcal{I}_2$ holds), then $\mathcal{I}$ is also closed under s, r and $\mathcal{I}_2$ (i.e. $\mathcal{I}{\downarrow}(s, r, \mathcal{I}_2)$ also holds). This is formalised in the following lemma. Later, we appeal to this lemma to establish the validity of the Shift principle.

Lemma 7 (Shift closure)**.** *For all $s, r \in$ IState (Def. 91) and $\mathcal{I}, \mathcal{I}_1, \mathcal{I}_2 \in$ AMod (Def. 94):*

$$\mathcal{I}{\downarrow}(s, r, \mathcal{I}_1) \land \mathcal{I}_1 \sqsubseteq^{\{s\}} \mathcal{I}_2 \Rightarrow \mathcal{I}{\downarrow}(s, r, \mathcal{I}_2)$$

Proof. The full proof is given §C (Lemma 31).

Given the set of logical environments LEnv, the sets of assertions Ast and interference assertions IAst (Def. 103), the $\models_{\text{SL}}$ satisfiability relation and the $\langle\!|.|\!\rangle_{(.)}$ interpretation function (Def. 104), an interference assertion $I' \in$ IAst is a *shifting* of $I \in$ IAst with respect to $P \in$ Ast, written $I \sqsubseteq^{P} I'$, if and only if:

$$\forall \Gamma \in \text{LEnv}, n \in \mathbb{N}.\ (\langle\!|I|\!\rangle_{\Gamma}, n) \sqsubseteq^{\{l \mid \Gamma, l \models_{\text{SL}} P\}} (\langle\!|I'|\!\rangle_{\Gamma}, n)$$

The second line of the $\mathcal{I} \sqsubseteq^S \mathcal{I}'$ definition requires that the new action model $\mathcal{I}'$ not introduce new actions (i.e. actions not present in $\mathcal{I}$). The last line of the definition stipulates that the new action model $\mathcal{I}'$ include all visible potential actions of $\mathcal{I}$. The $I \sqsubseteq^P I'$ states that the interference assertion I' is the shifting of I under assertion P, defined simply as the lifting of the shifting relation $\sqsubseteq$ to assertions.

For instance, consider the interference assertion I'_{y} from §8, repeated above (p. 286), and the interference assertion I_{y} from Fig. 8.2, repeated below:

$$I_{\mathsf{y}} \triangleq \begin{cases} [\mathsf{a_x}]^{\mathrm{T}} : \exists \mathsf{v}.\ \mathsf{x} \mapsto \mathsf{v} * \mathsf{y} \mapsto \mathsf{v} * \mathsf{z} \mapsto \mathsf{v} * \mathsf{v} \dot{<} 10 \rightsquigarrow \mathsf{x} \mapsto \mathsf{v}{+}1 * \mathsf{y} \mapsto \mathsf{v} * \mathsf{z} \mapsto \mathsf{v} \\ [\mathsf{a_y}]^{\mathrm{T}} : \qquad \exists \mathsf{v}.\ \mathsf{x} \mapsto \mathsf{v}{+}1 * \mathsf{y} \mapsto \mathsf{v} * \mathsf{v} \dot{<} 10 \rightsquigarrow \mathsf{x} \mapsto \mathsf{v}{+}1 * \mathsf{y} \mapsto \mathsf{v}{+}1 \end{cases}$$

We then have $I'_{\mathsf{y}} \sqsubseteq^{Q_0} I_{\mathsf{y}}$ with $Q_0 \triangleq \mathsf{x} \mapsto 0 * \mathsf{y} \mapsto 0$. Observe that establishing $I'_{\mathsf{y}} \sqsubseteq^{Q_0} I_{\mathsf{y}}$ requires a semantic (model-level) check. However, as we demonstrate later in §9.5, rather than checking interference shifting semantically, we present a number of syntactic judgements that reduce interference shifting to logical entailments. Using these judgements, we then demonstrate that $I'_{\mathsf{y}} \sqsubseteq^{Q_0} I_{\mathsf{y}}$ holds (see Example 14 on p. 307).

We can now show that the SHIFT principle is valid.

Lemma 8 (SHIFT validity)**.** *For all $P \in$ AST and $I, I' \in$ IAST (Def. 103):*

$$\vdash \boxed{P}_I \land I \sqsubseteq^P I' \overset{\textsc{Shift}}{\Longrightarrow} \boxed{P}_{I'}$$

Proof. Pick arbitrary $P \in$ AST, $I, I' \in$ IAST, $\Gamma \in$ LENV, and $w{=}(l, g, \mathcal{I}) \in$ WORLD such that:

$$\Gamma, w \models \boxed{P}_I \tag{9.6}$$

$$I \sqsubseteq^P I' \tag{9.7}$$

It then suffices to show that:

$$\Gamma, w \models \boxed{P}_{I'} \tag{9.8}$$

From (9.6) and definition of $\models$ we know that there exist $s, r \in$ ISTATE such that:

$$l \in \textsc{Unit}_{\textsc{Ins}} \land g{=}s \circ r \land \Gamma, s \models_{g,\mathcal{I}} P \land \mathcal{I}{\downarrow} (s, r, (\langle\!| I |\!\rangle_\Gamma, \mathcal{I}_{\mathrm{T}})) \tag{9.9}$$

From Lemma 4 (9.1) above and (9.9) we know $\Gamma, s \vDash_{\mathrm{SL}} P$ and consequently from (9.9) and the definition of $\sqsubseteq^{P}$ we have:

$$(\langle\!\lvert I \rvert\!\rangle_{\Gamma}, \mathcal{I}_{\mathrm{T}}) \sqsubseteq^{\{s\}} (\langle\!\lvert I' \rvert\!\rangle_{\Gamma}, \mathcal{I}_{\mathrm{T}}) \tag{9.10}$$

and thus from (9.9), (9.10) and Lemma 7 we have:

$$\mathcal{I}{\downarrow} \left(s, r, (\langle\!\lvert I' \rvert\!\rangle_{\Gamma}, \mathcal{I}_{\mathrm{T}})\right) \tag{9.11}$$

Consequently, from (9.9), (9.11) and the definition of $\models$ we have:

$$\Gamma, w \models \boxed{P}_{I'}$$

and can discharge the obligation in (9.8) as required. □

9.4. Rely and Guarantee

We define the rely and guarantee conditions of each thread in terms of their action models. This allows us to define *stability* against rely conditions, *repartitioning*, which logically represents a thread's atomic actions (and have to be in the guarantee condition), and *semantic implication*. Equipped with these notions, we can justify the EXTEND principle.

Rely The rely relation represents the potential interference from the environment. Although the rely will be different for every program (and indeed, every thread), it is always defined in the same way, which can be distilled into two relations.

The first relation, R^{e}, transforms a world $(l, g, \mathcal{I})$ to $(l, g \circ g', \mathcal{I}'' \cup \mathcal{I}')$, extending the shared state g with the new resources in g', modifying the action model $\mathcal{I}$ to $\mathcal{I}''$ (by extending its ticket component with a fresh ticket), while simultaneously extending it with $\mathcal{I}'$ describing how the new resources in g' may be manipulated. In order to ensure that the newly introduced actions do not interfere with existing shared resources and thus do not invalidate other threads' views of the shared state, these new actions must be confined to the extension (i.e. g' © $\mathcal{I}'$). Recall that when extending the shared state, a thread may also introduce *fresh* capabilities to enable the new actions. To ensure the freshness of the newly generated capabilities, the ticket component of the action model $\mathcal{I}$ is extended with a fresh ticket t not previously claimed ($t \notin dom(\mathcal{I}_{\mathrm{T}})$). That is, $\mathcal{I}$ is transformed to $\mathcal{I}''$ by extending its ticket component with a fresh ticket t, while leaving its action component unchanged (i.e. $\mathcal{I}''_{\mathrm{A}}{=}\mathcal{I}_{\mathrm{A}}$).

The second relation, R^{u}, transforms a world $(l, g, \mathcal{I})$ to $(l, g', \mathcal{I})$ by updating the shared state g to g' according to the actions in the action model $\mathcal{I}$. That is, at any one point the environment may update the shared state g by performing an action for which it has the sufficient capability (κ). For this to be possible, the thread performing the action must have the κ capability in its own local state and thus κ must be compatible with the capabilities contained in both the shared state g and the local states l (i.e. $\kappa \sharp (l \circ g)_{\mathrm{K}}$).

Definition 112 (Rely)**.** Given the set of worlds $\textsc{World}$ (Def. 102), the *extension rely* relation, $R^{\mathsf{e}} : \mathcal{P}(\textsc{World} \times \textsc{World})$, is defined as follows:

$$R^{\mathsf{e}} \triangleq \left\{ \begin{pmatrix} (l, g, \mathcal{I}), \\ (l, g \circ g', \mathcal{I}'' \cup \mathcal{I}') \end{pmatrix} \middle| \begin{array}{l} (\mathcal{I}''{=}\mathcal{I} \lor \exists t.\, t \notin dom(\mathcal{I}_{\mathrm{T}}) \land \mathcal{I}''{=}(\mathcal{I}_{\mathrm{A}}, \mathcal{I}_{\mathrm{T}} \uplus [t \mapsto 1])) \\ \land\, g' \circledc \mathcal{I}' \end{array} \right\}$$

where $\circledc$ denotes the confinement relation (Def. 100) and $\uplus$ denotes the standard disjoint function union.
The *update rely* relation, $R^{\mathsf{u}} : \mathcal{P}(\textsc{World} \times \textsc{World})$, is defined as follows:

$$R^{\mathsf{u}} \triangleq \left\{ \Big((l, g, \mathcal{I}),\ (l, g', \mathcal{I}) \Big) \,\middle|\, \exists \kappa.\ \kappa \sharp (l \circ g)_{\mathrm{K}} \land (g, g') \in \lceil \mathcal{I} \rceil (\kappa) \right\}$$

where $\sharp$ denotes the compatibility relation (Def. 93), and

$$\lceil \mathcal{I} \rceil (\kappa) \triangleq \{ (p \circ c \circ r, q \circ c \circ r) \mid (p, q, c) \in \mathcal{I}(\kappa) \land r \in \textsc{IState} \}$$

The *rely* relation, $\mathcal{R} : \mathcal{P}(\textsc{World} \times \textsc{World})$ is defined as follows, where $*$ denotes the reflexive transitive closure of the relation:

$$\mathcal{R} \triangleq (R^{\mathsf{u}} \cup R^{\mathsf{e}})^*$$

The rely relation enables us to define the stability of assertions with respect to the environment actions.

Definition 113 (Stability)**.** Given the set of logical environments $\textsc{LEnv}$, the set of worlds $\textsc{World}$ (Def. 102), the set of assertions $\textsc{Ast}$ (Def. 103), the assertion satisfiability relation $\models$ (Def. 104) and the rely relation $\mathcal{R}$ (Def. 112), an assertion $P \in \textsc{Ast}$ is *stable*, written $\mathsf{stable}\,(P)$, if and only if:

$$\mathsf{stable}\,(P) \overset{\text{def}}{\iff} \mathsf{stable}\,(P, \mathcal{R})$$

where for all relations $R \in \mathcal{P}\,(\textsc{World} \times \textsc{World})$, the $\mathsf{stable}\,(P, R)$ is defined as follows:

$$\begin{aligned}\mathsf{stable}\,(P, R) \overset{\text{def}}{\iff} & \; \forall \Gamma \in \text{LE}\textsc{nv}.\, \forall w, w' \in \textsc{World}. \\ & \qquad \Gamma, w \models P \wedge (w, w') \in R \Rightarrow \Gamma, w' \models P\end{aligned}$$

Proving that an assertion is stable is not always obvious, in particular when there are numerous transitions to consider (all those in the reflexive transitive closure of R^{e} and R^{u}). As we demonstrate in the following lemma, it suffices to check stability against the update actions in R^{u}.

Lemma 9 (Assertion stability)**.** *Given the extend rely* R^{e} *(Def. 112), for all* $\Gamma \in \text{LE}\textsc{nv}$*,* $w, w' \in \textsc{World}$ *(Def. 102) and* $P \in \textsc{Ast}$ *(Def. 103),* $\mathsf{stable}\,(P, R^{\mathsf{e}})$ *holds:*

$$\Gamma, w \models P \wedge (w, w') \in R^{\mathsf{e}} \implies \Gamma, w' \models P \qquad \text{(a)}$$

Given the update rely R^{u} *(Def. 112), for all* $P \in \textsc{Ast}$ *(Def. 103), if* P *is stable with respect to the actions in* R^{u} *(Def. 112), then it is stable:*

$$\mathsf{stable}\,(P, R^{u}) \Rightarrow \mathsf{stable}\,(P) \qquad \text{(b)}$$

Proof. The full proof of both parts is given in §C (Lemma 37 and Lemma 38, respectively).

Observe that establishing $\mathsf{stable}\,(P, R^{\mathsf{u}})$ requires a semantic (model-level) check. However, as we demonstrate later in §9.5, rather than checking stability semantically, we present a number of syntactic judgements that reduce stability to logical entailments.

Guarantee We now define the guarantee relation that describes all possible updates the current thread may perform. The guarantee relation is the dual of the rely relation: the actions in the guarantee of one thread are included in the rely of concurrently running threads. As such, it should come as no surprise that transitions in the guarantee can be similarly categorised into two relations resonating with those of the rely.

The *extension guarantee* relation, G^{e}, transforms a world $(l \circ l', g, \mathcal{I})$ to $(l \circ (h_1, \kappa_1), g \circ g', \mathcal{I}'' \cup \mathcal{I}')$, where $g' = l' \circ (h_2, \kappa_2)$ and $h_1, h_2 \in \textsc{Unit}_{\mathrm{L}}$. As with the rely extension R^{e}, the action model $\mathcal{I}$ is updated to $\mathcal{I}''$ (by extending it with a fresh ticket t), and is similarly extended with $\mathcal{I}'$ describing how the new resources in g' may be manipulated provided that g' © $\mathcal{I}'$. Moreover, the extension g' contains the locally held resources in l', as well as the freshly generated capability κ_2 (with ticket t). Similarly, the local state l is extended with the freshly generated capability κ_1 (with ticket t).

Similar to its rely counterpart, the *update guarantee* relation, G^{u}, transforms a world $(l, g, \mathcal{I})$ to $(l, g', \mathcal{I})$ by updating the shared state g to g' according to the actions in the action model $\mathcal{I}$, provided that the associated capability is held locally (i.e. contained in l). The guarantee extension is more involved than its R^{u} counterpart, because updates in the guarantee may move resources from the local state into the shared state while simultaneously mutating the shared state as prescribed by the action model. When updating the shared state, threads are not allowed to introduce new capabilities, as this can only be achieved when extending the shared state (through G^{e}). Intuitively, we must ensure that resources are not created "out of thin air" in the process. This can be expressed as preserving the *orthogonal* set of the combined local and shared states; that is, the set of states compatible with that combination. Given a partial commutative monoid $(\mathcal{M}, \bullet_{\mathcal{M}}, \textsc{Unit}_{\mathcal{M}})$, the *orthogonal* set of an element m is defined as the set of all elements in $\mathcal{M}$ that are compatible with it.

Definition 114 (Orthogonal set)**.** Given a partial commutative monoid $(\mathcal{M}, \bullet_{\mathcal{M}}, \textsc{Unit}_{\mathcal{M}})$, the *orthogonal set* function $(.)^{\sharp} : \mathcal{M} \to \mathcal{P}(\mathcal{M})$, is defined as follows, for all $m \in \mathcal{M}$:

$$(m)^{\sharp} \triangleq \left\{ m' \mid m \sharp m' \right\}$$

Definition 115 (Guarantee)**.** Given the set of worlds $\textsc{World}$ (Def. 102), the *extension guarantee* relation, $G^{\mathsf{e}} : \mathcal{P}(\textsc{World} \times \textsc{World})$, is defined as follows:

$$G^{\mathsf{e}} \triangleq \left\{ \begin{pmatrix} (l \circ l', g, \mathcal{I}), \\ (l \circ (h_1, \kappa_1), g \circ g', \mathcal{I}'' \cup \mathcal{I}') \end{pmatrix} \middle| \begin{array}{l} \exists t, h_2, \kappa_2.\ h_1, h_2 \in \textsc{Unit}_{\textsc{l}} \\ \quad \wedge\, dom(\kappa_1) \cup dom(\kappa_2) \subseteq \{t\} \\ \quad \wedge\, \mathcal{I}'' = (\mathcal{I}_{\textsc{a}}, \mathcal{I}_{\textsc{t}} \uplus [t \mapsto 1]) \\ \quad \wedge\, g' = l' \circ (h_2, \kappa_2) \wedge g' \circledcirc \mathcal{I}' \end{array} \right\}$$

where $\circledcirc$ denotes the confinement relation (Def. 100) and $\uplus$ denotes the standard disjoint function union.

The *update guarantee* relation, $G^{\mathsf{u}} : \mathcal{P}(\textsc{World} \times \textsc{World})$, is defined as follows:

$$G^{\mathsf{u}} \triangleq \left\{ ((l, g, \mathcal{I}), (l', g', \mathcal{I})) \middle| \begin{array}{l} ((l' \circ g')_{\textsc{k}})^{\sharp} = ((l \circ g)_{\textsc{k}})^{\sharp} \wedge \\ \begin{bmatrix} g = g' \vee \exists \kappa \leq l_{\textsc{k}}. \\ \quad (g, g') \in \lceil \mathcal{I} \rceil (\kappa) \\ \quad \wedge\, ((l \circ g)_{\textsc{l}})^{\sharp} = ((l' \circ g')_{\textsc{l}})^{\sharp} \end{bmatrix} \end{array} \right\}$$

where $(.)^{\sharp}$ denotes the orthogonal function (Def. 114), the $\leq$ denotes the ordering relation (Def. 92) and $\lceil \mathcal{I} \rceil$ is as given in Def. 112.

The *guarantee* relation, $\mathcal{G} : \mathcal{P}(\textsc{World} \times \textsc{World})$ is defined as follows, where $*$ denotes the reflexive transitive closure of the relation:

$$\mathcal{G} \triangleq (G^{\mathsf{u}} \cup G^{\mathsf{e}})^*$$

Using the guarantee relation, we introduce the notion of *repartitioning* $P \Rrightarrow^{\{p\}\{q\}} Q$. This relation holds whenever, from any world satisfying P, if whenever parts of the composition of its local and shared states that satisfies p is exchanged for one satisfying q, it is possible to split the resulting logical state into a local and shared part again, in such a way that the resulting transition is in $\mathcal{G}$, and the resulting world satisfies Q.

Definition 116 (Repartitioning)**.** Given the sets of logical states $\textsc{LState}$ (Par. 22), worlds $\textsc{World}$ (Def. 102), capabilities $\textsc{Kap}$ (Def. 90) and the guarantee relation $\mathcal{G}$ (Def. 115), the *repartitioning* relation, $\Rrightarrow$: $(\mathcal{P}(\textsc{World}) \times \mathcal{P}(\textsc{LState})) \times (\mathcal{P}(\textsc{World}) \times \mathcal{P}(\textsc{LState}))$, contains $((W_1, L_1), (W_2, L_2))$ if and only if:

for all worlds $w_1 = (l_1, g_1, \mathcal{I}_1) \in W_1$, there exist $h_1, h' \in \textsc{LState}$ such that:

1. $h_1 \in L_1$; and

2. $h_1 \bullet_{\mathrm{L}} h' = (l_1 \circ g_1)_{\mathrm{L}}$; and

3. for all $h_2 \in L_2$, there exists a world $w_2{=}(l_2, g_2, \mathcal{I}_2)$ such that:

   a) $w_2 \in W_2$; and

   b) $h_2 \bullet_{\mathrm{L}} h' = (l_2 \circ g_2)_{\mathrm{L}}$; and

   c) $(w_1, w_2) \in \mathcal{G}$

Given the set of logical environments LEnv, assertions Ast and local assertions LAst (Def. 103) and the $\vDash_{\mathrm{SL}}$ and $\models$ satisfiability relations (Def. 104), the repartitioning relation $\Rrightarrow$ is lifted to assertions as follows, for all $P, Q \in$ Ast and $p, q \in$ LAst:

$$\big((P,p),(Q,q)\big) \in \Rrightarrow \overset{\text{def}}{\iff} \forall \Gamma \in \text{LEnv}.\ \big((W_1, L_1),(W_2, L_2)\big) \in \Rrightarrow$$

where

$$\begin{array}{ll} W_1 \triangleq \{w \mid \Gamma, w \models P\} & W_2 \triangleq \big\{w \mid \Gamma, w \models Q\big\} \\ L_1 \triangleq \big\{h \mid \exists \kappa \in \text{Unit}_{\mathrm{K}}.\ \Gamma,(h,\kappa) \vDash_{\mathrm{SL}} p\big\} & \\ L_2 \triangleq \big\{h \mid \exists \kappa \in \text{Unit}_{\mathrm{K}}.\ \Gamma,(h,\kappa) \vDash_{\mathrm{SL}} q\big\} & \end{array}$$

We write $P \Rrightarrow^{\{p\}\{q\}} Q$ for $((P,p),(Q,q)) \in\Rrightarrow$, and write $P \Rrightarrow Q$ for $P \Rrightarrow^{\{\mathsf{emp}\}\{\mathsf{emp}\}} Q$, in which case the repartitioning has no "side effect" and simply shuffles resources around between the local and shared state or modifies the action models (i.e. via transitions in G^{e}).

We can now show that the Extend principle is valid.

Lemma 10 (Extend validity)**.** *For all $P \in$ Ast and $I \in$ IAst (Def. 103):*

$$\begin{array}{ll} \textit{if} & P * [\mathcal{C}_2]^{\mathrm{T}} \circledcirc I \quad \textit{and} \quad \mathrm{T} \notin \mathit{fv}(P, \mathcal{C}_1, \mathcal{C}_2) \quad \textit{and} \quad \mathsf{sat}(\mathcal{C}_1, \mathcal{C}_2) \\ \textit{then} & \vdash P \overset{\text{Extend}}{\Rrightarrow} \exists \mathrm{T}.\ [\mathcal{C}_1]^{\mathrm{T}} * \boxed{P * [\mathcal{C}_2]^{\mathrm{T}}}_{I} \end{array}$$

Proof. Pick an arbitrary $\Gamma \in$ LEnv and $w_1 = (l, g, \mathcal{I}) \in$ World such that

$$P * [\mathcal{C}_2]^{\mathrm{T}} \circledcirc I \tag{9.12}$$

$$\mathrm{T} \notin \mathit{fv}(P, \mathcal{C}_1, \mathcal{C}_2) \quad \text{and} \quad \mathsf{sat}(\mathcal{C}_1, \mathcal{C}_2) \tag{9.13}$$

$$\Gamma, w_1 \models P \tag{9.14}$$

From (9.13) and the definition of sat (Par. 25) we know that there exist

$\cancel{c}, \cancel{c}_1, \cancel{c}_2 \in \textsc{Cap}$ such that:

$$\cancel{c} = \cancel{c}_1 \bullet_{\mathrm{C}} \cancel{c}_2 \land \Gamma, \cancel{c}_1 \models_{\mathrm{C}} \mathcal{C}_1 \land \Gamma, \cancel{c}_2 \models_{\mathrm{C}} \mathcal{C}_2 \tag{9.15}$$

Pick a ticket $t \in \mathbb{N}$ such that $t \notin \mathit{dom}(\mathcal{I}_{\mathrm{T}})$. It is always possible to pick such t since the domain of $\mathcal{I}_{\mathrm{T}}$ is finite. Pick $\mathcal{I}', \mathcal{I}'', \kappa_1, \kappa_2, h_1, h_2, l', g', w_2$ such that:

$$\mathcal{I}'' = (\mathcal{I}_{\mathrm{A}}, \mathcal{I}_{\mathrm{T}} \uplus [t \mapsto 1]) \tag{9.16}$$

$$\kappa_1 = [t \mapsto \cancel{c}_1] \text{ and } \kappa_2 = [t \mapsto \cancel{c}_2] \tag{9.17}$$

$$h_1, h_2 \in \textsc{Unit}_{\mathrm{L}} \text{ and } l' = (h_1, \kappa_1) \text{ and } g' = g \circ l \circ (h_2, \kappa_2) \tag{9.18}$$

$$\mathcal{I}' = (\llparenthesis I \rrparenthesis_{[\Gamma \mid \mathrm{T}:t]}, \mathcal{I}''_{\mathrm{T}}) \text{ and } w_2 = (l', g', \mathcal{I}'' \cup \mathcal{I}') \tag{9.19}$$

From the definition of $\Rrightarrow$ it then suffices to show:

$$(l \circ g)_{\mathrm{L}} = (l' \circ g')_{\mathrm{L}} \tag{9.20}$$

$$(w_1, w_2) \in \mathcal{G} \tag{9.21}$$

$$\mathsf{wf}\,(w_2) \tag{9.22}$$

$$\Gamma, w_2 \models \exists \mathrm{T}.\ [\mathcal{C}_1]^{\mathrm{T}} * \boxed{P * [\mathcal{C}_2]^{\mathrm{T}}}_{I} \tag{9.23}$$

RTS. (9.20)
This follows immediately from the definitions of l' and g' in (9.18).

RTS. (9.21)
From (9.14) and Lemma 4 (9.2) we have $\Gamma, l \vDash_{\mathrm{SL}} P$. Since $\mathrm{T} \notin \mathit{fv}(P)$, we then have $[\Gamma \mid \mathrm{T}:t], l \vDash_{\mathrm{SL}} P$. Similarly, since $\mathrm{T} \notin \mathit{fv}(\mathcal{C}_2)$, from (9.15), (9.17), (9.18) and the definition of $\vDash_{\mathrm{SL}}$ we have $[\Gamma \mid \mathrm{T}:t], (h_2, \kappa_2) \vDash_{\mathrm{SL}} [\mathcal{C}_2]^{\mathrm{T}}$. Consequently we have:

$$[\Gamma \mid \mathrm{T}:t], l \circ (h_2, \kappa_2) \vDash_{\mathrm{SL}} P * [\mathcal{C}_2]^{\mathrm{T}} \tag{9.24}$$

From (9.12), (9.19) and the definition of $\copyright$ we know there exist $\mathcal{F}'$ such that:

$$\{l \mid [\Gamma \mid \mathrm{T}:t], l \vDash_{\mathrm{SL}} P * [\mathcal{C}_2]^{\mathrm{T}}\} \subseteq \mathcal{F}' \land \mathcal{F}' \blacktriangleright \mathcal{I}' \tag{9.25}$$

From (9.24), (9.25) and the definition of $\blacktriangleright$ we have:

$$l \circ (h_2, \kappa_2) \circledcirc \mathcal{I}' \tag{9.26}$$

Consequently, from the definition of w_2 (9.19) we have $(w_1, w_2) \in G^{\mathsf{e}}$ and thus $(w_1, w_2) \in \mathcal{G}$, as required.

RTS. (9.22)
From the well-formedness of w_1 we know $\mathit{dom}((l \circ g)_{\mathrm{K}}) \subseteq \mathit{dom}(\mathcal{I}_{\mathrm{T}})$. From the definition of $\mathcal{I}'' \cup \mathcal{I}'$ we know $\mathit{dom}((\mathcal{I}'' \cup \mathcal{I}')_{\mathrm{T}}) = \mathit{dom}(\mathcal{I}_{\mathrm{T}}) \uplus \{t\}$. From the definition of l' and g' we know $\mathit{dom}((l' \circ g')_{\mathrm{K}}) = \mathit{dom}((l \circ g)_{\mathrm{K}}) \uplus \{t\}$. As such we have:

$$\mathit{dom}((l' \circ g')_{\mathrm{K}}) \subseteq \mathit{dom}((\mathcal{I}'' \cup \mathcal{I}')_{\mathrm{T}}) \tag{9.27}$$

From well-formedness of w_1 we know $g \circledcirc \mathcal{I}$. From the definition of $\mathcal{I}''$ and the definition of $\circledcirc$ we then have $g \circledcirc \mathcal{I}''$. Consequently, from the definition of g' in (9.18), (9.26) and Lemma 3 we have:

$$g' \circledcirc \mathcal{I}'' \cup \mathcal{I}' \tag{9.28}$$

From the well-formedness of w_1 we know $l \sharp g$ and $\mathit{dom}((l \circ g)_{\mathrm{K}}) \subseteq \mathit{dom}(\mathcal{I}_{\mathrm{T}})$. Since t is a fresh ticket (not in $\mathit{dom}(\mathcal{I}_{\mathrm{T}})$), from the definition of l' and g' we then trivially have $l' \sharp g'$. As such, from (9.27), (9.28) and the definitions of wf and w_2 we have $\mathsf{wf}\,(w_2)$ as required.

RTS. (9.23)
Since $\mathrm{T} \notin \mathit{fv}(\mathcal{C}_1)$, from (9.15), (9.17), (9.18) and the definition of $\vDash_{\mathrm{SL}}$ we have $[\Gamma \mid \mathrm{T}{:}t], (h_1, \kappa_1) \vDash_{\mathrm{SL}} [\mathcal{C}_1]^{\mathrm{T}}$. Consequently, from the definition of $\models$ we have:

$$[\Gamma \mid \mathrm{T}{:}t], \big((h_1, \kappa_1), g', \mathcal{I}'' \cup \mathcal{I}'\big) \models [\mathcal{C}_1]^{\mathrm{T}} \tag{9.29}$$

Since $\mathrm{T} \notin \mathit{fv}(P)$, from (9.14) and the definitions of $\models$ and w_1 we have $[\Gamma \mid \mathrm{T}{:}t], \big(l, g, \mathcal{I}\big) \models P$. Consequently, from the semantics of assertions we have $[\Gamma \mid \mathrm{T}{:}t], \big(l, g, \mathcal{I}''\big) \models P$. On the other hand, since $\mathrm{T} \notin \mathit{fv}(\mathcal{C}_2)$, from (9.15), (9.17), (9.18) and the definition of $\models$ we have $[\Gamma \mid \mathrm{T}{:}t], \big((h_2, \kappa_2), g, \mathcal{I}''\big) \models$

$[\mathcal{C}_2]^{\mathrm{T}}$. Consequently we have:

$$[\Gamma \,|\, \mathrm{T}{:}t], \big(l \circ (h_2, \kappa_2), g, \mathcal{I}''\big) \models P * [\mathcal{C}_2]^{\mathrm{T}} \tag{9.30}$$

Thus from (9.26), (9.30), the definition of g' (9.18) and Lemma 36 (in §C) we have:

$$[\Gamma \,|\, \mathrm{T}{:}t], l \circ (h_2, \kappa_2) \models_{g', \mathcal{I}'' \cup \mathcal{I}'} P * [\mathcal{C}_2]^{\mathrm{T}} \tag{9.31}$$

From well-formedness of w_1 we know g © $\mathcal{I}$. From the definition of $\mathcal{I}''$ and the definition of © we then have g © $\mathcal{I}''$. As such, from (9.26) and Lemma 5(c) we have:

$$\mathcal{I}'' \cup \mathcal{I}' \downarrow \big(l \circ (h_2, \kappa_2), g, \mathcal{I}'\big) \tag{9.32}$$

From (9.29), (9.31), (9.32), the definition of g' (9.18) and the definitions of w_2 and $\models$ we have:

$$[\Gamma \,|\, \mathrm{T}{:}t], w_2 \models [\mathcal{C}_1]^{\mathrm{T}} * \boxed{P * [\mathcal{C}_2]^{\mathrm{T}}}_I$$

and thus from the definition of $\models$ we have:

$$\Gamma, w_2 \models \exists \mathrm{T}.\ [\mathcal{C}_1]^{\mathrm{T}} * \boxed{P * [\mathcal{C}_2]^{\mathrm{T}}}_I$$

as required. □

9.5. CoLoSL Judgements as SL Entailments

Observe that the semantic definitions of interference confinement P © I (Def. 108) and interference shifting $I \sqsubseteq^P I'$ (Def. 111) are defined by interpreting P via the (standard) separation logic satisfiability relation $\vDash_{\mathrm{SL}}$ (Def. 104) which ignores the subjective boxed assertions (that is, any empty instrumented state in $\textsc{Unit}_{\textsc{Ins}}$ satisfies a boxed assertion via $\vDash_{\mathrm{SL}}$). In other words, in order to check P © I, it suffices to check the confinement of I (interpreted) with respect to the *local* states described by P, ignoring the shared states. As such, we define a simple mechanism for *weakening an assertion* P, written $\rfloor P$, by erasing the subjective views. That is, for a local assertion p we have $\rfloor p = p$, whereas $\rfloor \boxed{P}_I = \mathsf{emp}$. The weakening of other assertions is defined inductively following the assertion syntax. As we demonstrate shortly, for all CoLoSL assertions we have $P \vdash \rfloor P$. Intuitively, this is because $\boxed{P}_I \vdash \mathsf{emp}$. That is, both $\boxed{P}_I$ and emp describe worlds of the form $(l, g, \mathcal{I})$ where l is in the unit set. Moreover, whilst emp places no further constraints on g and $\mathcal{I}$, the semantics of $\boxed{P}_I$ stipulates certain conditions on g and $\mathcal{I}$.

Definition 117 (Weakening)**.** Given the set of assertions $\textsc{Ast}$ and local assertions $\textsc{LAst}$ (Def. 103), the *assertion weakening* function, $\rfloor(.) : \textsc{Ast} \to \textsc{LAst}$, is defined inductively over the structure of assertions as follows, for all $p \in \textsc{LAst}$, $P, Q \in \textsc{Ast}$ with $\odot \in \{\vee, *, \uplus\}$:

$$\rfloor p \triangleq p \qquad \rfloor(\exists \mathrm{x}.\ P) \triangleq \exists \mathrm{x}.\ \rfloor P \qquad \rfloor(P \odot Q) \triangleq \rfloor P \odot \rfloor Q \qquad \rfloor \boxed{P}_I \triangleq \mathsf{emp}$$

Lemma 11 (Weakening)**.** *For all $P \in \textsc{Ast}$ (Def. 103):*

$$P \vdash \rfloor P$$

Proof. By induction on the structure of assertions. The full proof is given in §C (Lemma 39).

Interference confinement and local fencing judgements We present a number of judgements that reduce the interference confinement condition P Ⓒ I and local fencing conditions to separation logic entailments in Fig. 9.1. In the judgements presented we assume that the logical variables $\bar{\mathrm{x}}$ do not appear free in f and f_j for all j. Furthermore, for the judgements presented in Fig. 9.1 we assume that the underlying partial commutative monoid of logical states (Par. 22) satisfies the disjointness property [17][1].

As expressed by the top-left rule (Ⓒ-Intro), the P Ⓒ I holds if there is a weaker assertion P' ($P \vdash P'$) that acts as a *local fence* for I ($P' \blacktriangleright I$). The remaining judgements are those of local fencing of the form $f \blacktriangleright I$ where $f \in \textsc{LAst}$ (Def. 103) is a local assertion without subjective views. As such, since for all assertions P we have $P \vdash \rfloor P$ (Lemma 11) and $\rfloor P \in \textsc{LAst}$, in order to ascertain P Ⓒ I, applying the (Ⓒ-Intro) rule it suffices to check the validity of $\rfloor P \blacktriangleright I$ using the local fencing judgements of Fig. 9.1. The premises of the local fencing judgements involve entailments of the

[1] $\forall h, h'.\ h \bullet_{\mathrm{L}} h = h' \implies h = h' \wedge h \in \textsc{Unit}_{\mathrm{L}}$

form $p \vdash_{\mathsf{SL}} p'$ where $p, p' \in$ LAST are local assertions. Pleasantly, since local assertions do not contain subjective (boxed) assertions, entailments of the form $p \vdash_{\mathsf{SL}} p'$ are the familiar entailments of standard separation logic.

Recall that the local fencing judgement $f \blacktriangleright I$ states that f must be invariant under all actions of I and must confine the actions in f. The fencing condition is checked for each action in I (the $\blacktriangleright$-EMPTY and $\blacktriangleright$-COMP rules). For each action of the form $r : \exists \bar{\mathrm{x}}.\ p \rightsquigarrow q$ in I, the five remaining rules of Fig. 9.1 may apply. The ($\blacktriangleright$-AWEAK) rule simply allows us to check local fencing for the weaker actions denoted by $r : \exists \bar{\mathrm{x}}.\ p' \rightsquigarrow q'$ which contain the actions of $r : \exists \bar{\mathrm{x}}.\ p \rightsquigarrow q$. Similarly, the ($\blacktriangleright$-EQUIV) rule allows us to check local fencing for an equivalent set of actions. In the ($\blacktriangleright$-FALSE) rule, the action of $r : \exists \bar{\mathrm{x}}.\ p \rightsquigarrow q$ cannot possibly fire as its precondition does not *agree* with f (see Def. 95): no state satisfying f may be extended such that a subpart satisfies p. The ($\blacktriangleright$-EXACT) rule allows us to trim a *neutral* part r' of the action (corresponding to a part of the catalyst in the interpretation of the action) appearing both in p and q. This may only be applied when r' is *exact*; that is, satisfied by at most one instrumented state.[2] When this is the case, the part of the state denoted by r' is then uniquely determined and left unchanged by the action. The last rule ($\blacktriangleright$-INV) reduces local fencing to entailment checking, provided that the fence f can be expressed as a disjunction of *precise* assertions (second premise). An assertion is precise if its is satisfied by at most one substate of each instrumented state.[3] The first premise states that f is invariant under the action $r : \exists \bar{\mathrm{x}}.\ p \rightsquigarrow q$, akin to the RGSep encoding of stability checks as separation logic entailments [57]. Informally, it asserts that for any state in f, when a part satisfying p is removed and a state satisfying q is added instead, the result must still be in f. The third premise checks the confinement condition: given a state $l_1 \circ l_2$ in the local assertion f_j ($l_1 \circ l_2 \vDash_{\mathrm{SL}} f_j$), and a state $l_2 \circ l_3$ in p ($l_2 \circ l_3 \models p$ where $l_1 \circ l_2 \circ l_3$ is defined), then the combined state $l_1 \circ l_2 \circ l_3$ must also be in f_j. Hence, by the precision of f_j we have $l_1 \circ l_2 \circ l_3 = l_1 \circ l_2$. That is, $l_1 \circ l_3 \leq l_1 \circ l_2$ as required by the definition of local fencing $\blacktriangleright$.

[2] $\mathsf{exact}\,(p) \triangleq \forall \Gamma, l_1, l_2.\ \Gamma, l_1 \vDash_{\mathrm{SL}} p \land \Gamma, l_2 \vDash_{\mathrm{SL}} p \implies l_1 = l_2$

[3] $\mathsf{precise}\,(p) \triangleq \forall \Gamma, l, l_1, l_2.\ l_1 \leq l \land l_2 \leq l \land \Gamma, l_1 \vDash_{\mathrm{SL}} p \land \Gamma, l_2 \vDash_{\mathrm{SL}} p \implies l_1 = l_2$

$$\frac{P \vdash P' \quad P' \blacktriangleright I}{P \copyright I}\ \copyright\text{-INTRO} \qquad \frac{}{f \blacktriangleright \emptyset}\ \blacktriangleright\text{-EMPTY} \qquad \frac{f \blacktriangleright I_1 \quad f \blacktriangleright I_2}{f \blacktriangleright I_1 \cup I_2}\ \blacktriangleright\text{-COMP}$$

$$\frac{\begin{array}{c} p \vdash_{\mathsf{SL}} p' \qquad q \vdash_{\mathsf{SL}} q' \\ f \blacktriangleright \{r : \exists \bar{\mathrm{X}}.\ p' \rightsquigarrow q'\} \end{array}}{f \blacktriangleright \{r : \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\}}\ \blacktriangleright\text{-AWEAK} \qquad \frac{f \blacktriangleright \bigcup\limits_{\mathrm{X} \in J} \{r : \exists \bar{\mathrm{Y}}.\ p \rightsquigarrow q\}}{f \blacktriangleright \{r : \exists \mathrm{X} \in J.\ \exists \bar{\mathrm{Y}}.\ p \rightsquigarrow q\}}\ \blacktriangleright\text{-AEQUIV}$$

$$\frac{f \uplus p \vdash_{\mathsf{SL}} \mathsf{false}}{f \blacktriangleright \{r : \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\}}\ \blacktriangleright\text{-FALSE} \qquad \frac{\mathsf{exact}\,(r') \quad f \blacktriangleright \{r : \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\}}{f \blacktriangleright \{r : \exists \bar{\mathrm{X}}.\ p * r' \rightsquigarrow q * r'\}}\ \blacktriangleright\text{-EXACT}$$

$$\frac{\begin{array}{cc} (p \mathrel{-\!\!\circledast} f) * q \vdash_{\mathsf{SL}} f & f \Leftrightarrow \bigvee\limits_{j \in J} f_j \\ \mathsf{precise}\,(f_j) \quad \text{and} \quad f_j \uplus p \vdash_{\mathsf{SL}} f_j & \text{for all } j \in J \end{array}}{f \blacktriangleright \{r : \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\}}\ \blacktriangleright\text{-INV}$$

Figure 9.1.: Interference confinement and local fencing judgements

Example 12 (Confinement and local fencing judgements)**.** Using the confinement and local fencing rules in Fig. 9.1, we can now establish the $P_0 \copyright I$ judgement used in §8, with P_0, and I repeated below as defined in §8:

$$P_0 \triangleq \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{x} \mapsto 0 \qquad I \triangleq I_1 \cup I_2 \cup I_3$$

where

$$I_1 \triangleq \left\{ [\mathsf{a}_{\mathtt{x}}]^{\mathrm{T}} : \exists \mathrm{V} \in \{0 \cdots 9\}.\ \mathtt{x} \mapsto \mathrm{V} * \mathtt{z} \mapsto \mathrm{V} \rightsquigarrow \mathtt{x} \mapsto \mathrm{V}{+}1 * \mathtt{z} \mapsto \mathrm{V} \right.$$
$$I_2 \triangleq \left\{ [\mathsf{a}_{\mathtt{y}}]^{\mathrm{T}} : \exists \mathrm{V} \in \{0 \cdots 9\}.\ \mathtt{x} \mapsto \mathrm{V}{+}1 * \mathtt{y} \mapsto \mathrm{V} \rightsquigarrow \mathtt{x} \mapsto \mathrm{V}{+}1 * \mathtt{y} \mapsto \mathrm{V}{+}1 \right.$$
$$I_3 \triangleq \left\{ [\mathsf{a}_{\mathtt{z}}]^{\mathrm{T}} : \exists \mathrm{V} \in \{0 \cdots 9\}.\ \mathtt{y} \mapsto \mathrm{V}{+}1 * \mathtt{z} \mapsto \mathrm{V} \rightsquigarrow \mathtt{y} \mapsto \mathrm{V}{+}1 * \mathtt{z} \mapsto \mathrm{V}{+}1 \right.$$

Let us define the local fence assertion F and the interference assertions I_1', I_2', I_3' as follows:

$$F \triangleq \Big(\bigvee_{\mathrm{W}=0}^{10} F^1_{\mathrm{W}} \Big) \vee \Big(\bigvee_{\mathrm{W}=0}^{9} F^2_{\mathrm{W}} \Big) \vee \Big(\bigvee_{\mathrm{W}=0}^{9} F^3_{\mathrm{W}} \Big)$$

$$F^1_{\mathrm{W}} \triangleq \mathrm{X} \mapsto \mathrm{W} * \mathrm{Y} \mapsto \mathrm{W} * \mathrm{Z} \mapsto \mathrm{W}$$
$$F^2_{\mathrm{W}} \triangleq \mathrm{X} \mapsto \mathrm{W}{+}1 * \mathrm{Y} \mapsto \mathrm{W} * \mathrm{Z} \mapsto \mathrm{W}$$
$$F^3_{\mathrm{W}} \triangleq \mathrm{X} \mapsto \mathrm{W}{+}1 * \mathrm{Y} \mapsto \mathrm{W}{+}1 * \mathrm{Z} \mapsto \mathrm{W}$$

$$I'_1 \triangleq \bigcup_{j=0}^{9} I^{\mathtt{x}}_j \qquad I^{\mathtt{x}}_j \triangleq \left\{ [\mathsf{a}_{\mathtt{x}}]^{\mathrm{T}} : \mathtt{x} \mapsto j * \mathtt{z} \mapsto j \rightsquigarrow \mathtt{x} \mapsto j{+}1 * \mathtt{z} \mapsto j \right\}$$

$$I'_2 \triangleq \bigcup_{j=0}^{9} I^{\mathtt{y}}_j \qquad I^{\mathtt{y}}_j \triangleq \left\{ [\mathsf{a}_{\mathtt{y}}]^{\mathrm{T}} : \mathtt{x} \mapsto j{+}1 * \mathtt{y} \mapsto j \rightsquigarrow \mathtt{x} \mapsto j{+}1 * \mathtt{y} \mapsto j{+}1 \right\}$$

$$I'_3 \triangleq \bigcup_{j=0}^{9} I^{\mathtt{z}}_j \qquad I^{\mathtt{z}}_j \triangleq \left\{ [\mathsf{a}_{\mathtt{z}}]^{\mathrm{T}} : \mathtt{y} \mapsto j{+}1 * \mathtt{z} \mapsto j \rightsquigarrow \mathtt{y} \mapsto j{+}1 * \mathtt{z} \mapsto j{+}1 \right\}$$

We can then establish the validity of the $P_0 \copyright\, I$ judgement as shown in the derivation below:

$$\dfrac{\dfrac{}{P_0 \vdash F} \qquad \dfrac{\dfrac{\dfrac{(\dagger)}{F \blacktriangleright I'_1}}{F \blacktriangleright I_1}\;\blacktriangleright\text{-}\textsc{Equiv} \quad \dfrac{\dfrac{(\ddagger)}{F \blacktriangleright I'_2}}{F \blacktriangleright I_2}\;\blacktriangleright\text{-}\textsc{Equiv} \quad \dfrac{\dfrac{(\dagger\dagger)}{F \blacktriangleright I'_3}}{F \blacktriangleright I_3}\;\blacktriangleright\text{-}\textsc{Equiv}}{F \blacktriangleright I}\;\blacktriangleright\text{-}\textsc{Comp}}{P_0 \copyright\, I}\;\copyright\text{-}\textsc{Intro}$$

with

$$\dfrac{(*)\quad \dfrac{i\in\{1..3\}\quad j\in\{0..9\}}{F^i_j \uplus (\mathtt{x}\mapsto j * \mathtt{z}\mapsto j) \vdash_{\mathsf{SL}} F^i_j} \qquad \dfrac{j\in\{0..9\}}{\begin{pmatrix}((\mathtt{x}\mapsto j * \mathtt{z}\mapsto j) \mathrel{-\!\circledast} F)\\ *\ \mathtt{x}\mapsto j{+}1 * \mathtt{z}\mapsto j\end{pmatrix} \vdash_{\mathsf{SL}} F}}{\dfrac{F \blacktriangleright I^{\mathtt{x}}_j \quad \text{for all } j\in\{0..9\}}{(\dagger)}\;\triangleright\text{-}\textsc{Comp}}\;\triangleright\text{-}\textsc{Inv}$$

$$\dfrac{(*)\quad \dfrac{i\in\{1..3\}\quad j\in\{0..9\}}{F^i_j \uplus (\mathtt{x}\mapsto j{+}1 * \mathtt{y}\mapsto j) \vdash_{\mathsf{SL}} F^i_j} \qquad \dfrac{j\in\{0..9\}}{\begin{pmatrix}((\mathtt{x}\mapsto j{+}1 * \mathtt{y}\mapsto j) \mathrel{-\!\circledast} F)\\ *\ \mathtt{x}\mapsto j{+}1 * \mathtt{y}\mapsto j{+}1\end{pmatrix} \vdash_{\mathsf{SL}} F}}{\dfrac{F \blacktriangleright I^{\mathtt{y}}_j \quad \text{for all } j\in\{0..9\}}{(\ddagger)}\;\triangleright\text{-}\textsc{Comp}}\;\triangleright\text{-}\textsc{Inv}$$

$$\dfrac{(*)\quad \dfrac{i\in\{1..3\}\quad j\in\{0..9\}}{F^i_j \uplus (\mathtt{y}\mapsto j{+}1 * \mathtt{z}\mapsto j) \vdash_{\mathsf{SL}} F^i_j} \qquad \dfrac{j\in\{0..9\}}{\begin{pmatrix}((\mathtt{y}\mapsto j{+}1 * \mathtt{z}\mapsto j) \mathrel{-\!\circledast} F)\\ *\ \mathtt{y}\mapsto j{+}1 * \mathtt{z}\mapsto j{+}1\end{pmatrix} \vdash_{\mathsf{SL}} F}}{\dfrac{F \blacktriangleright I^{\mathtt{z}}_j \quad \text{for all } j\in\{0..9\}}{(\dagger\dagger)}\;\triangleright\text{-}\textsc{Comp}}\;\triangleright\text{-}\textsc{Inv}$$

and

$$\frac{\dfrac{i\in\{1..3\} \quad j\in\{0..9\}}{\mathsf{precise}\left(F^i_j\right)}}{(*)}$$

Interference shifting and fencing judgements We present several judgements for the interference shifting condition $I \sqsubseteq^P I'$ required by the Shift principle in the top part of Fig. 9.2. As before, we assume that the logical variables $\bar{\mathrm{x}}$ do not appear free in f and r_j for all j. Given an assertion P, we write $I \equiv^P I'$ as a shorthand for $I \sqsubseteq^P I' \land I' \sqsubseteq^P I$.

Recall that intuitively, the $I \sqsubseteq^P I'$ states that the interference assertion I may be replaced by I', as I and I' both describe the same interference with respect to the states described by P. The ($\sqsubseteq$-Weak) rule weakens the shifting assertion P to P'. The remaining shifting judgements are all of the form $I \sqsubseteq^f I'$ where $f \in$ LAst is a local assertion (Def. 103) without boxed assertions. As before, since for all assertions P we have $P \vdash \downharpoonleft P$ (Lemma 11) and $\downharpoonleft P \in$ LAst, in order to ascertain $I \sqsubseteq^P I$, applying the ($\sqsubseteq$-Weak) rule it suffices to check the validity of $I \sqsubseteq^{\downharpoonleft P} I'$ using the remaining shifting judgements. Once again, the premises of the remaining judgements involve entailments of the form $p \vdash_{\mathsf{SL}} p'$ where $p, p' \in$ LAst are local assertions. As such, these entailments are the familiar entailments of standard separation logic.

The next two rules ($\sqsubseteq$-Refl and $\sqsubseteq$-Tran) state that the shifting relation is $\sqsubseteq^f$ is reflexive and transitive, respectively. The ($\sqsubseteq$-Comp) rule checks the shifting judgement component-wise, provided that f is invariant with respect to $I_1 \cup I_2$; that is, $I_1 \cup I_2$ is fenced by f as stated by the $f \rhd I_1 \cup I_2$ premise (see Def. 111). This bigger fencing condition is necessary: the $I_1 \sqsubseteq^f I_1'$ only states that I_1 and I_1' have the same effect with respect to f. Similarly for $I_2 \sqsubseteq^f I_2'$. We thus need $f \rhd I_1 \cup I_2$ ensuring that f is an invariant of the shared state under the combined interferences of I_1 and I_2.

The ($\sqsubseteq$-AWeak) rule simply states that if the weaker actions denoted by $r : \exists \bar{\mathrm{x}}.\, p' \rightsquigarrow q'$ may be ignored under f, then the stronger actions of $r : \exists \bar{\mathrm{x}}.\, p \rightsquigarrow q$ (contained in those of $r : \exists \bar{\mathrm{x}}.\, p' \rightsquigarrow q'$) may also be ignored.

The next three rules capture situations where it is not possible to apply the action to f. The ($\sqsubseteq$-LFalse) rule describes the case when the

$$\frac{P \vdash P' \quad I \sqsubseteq^{P'} I'}{I \sqsubseteq^{P} I'}\ \sqsubseteq\text{-Weak} \qquad \frac{}{I \sqsubseteq^{f} I}\ \sqsubseteq\text{-Refl} \qquad \frac{I \sqsubseteq^{f} I'' \quad I'' \sqsubseteq^{f} I'}{I \sqsubseteq^{f} I'}\ \sqsubseteq\text{-Tran}$$

$$\frac{f \rhd I_1 \cup I_2 \quad I_1 \sqsubseteq^{f} I_1' \quad I_2 \sqsubseteq^{f} I_2'}{I_1 \cup I_2 \sqsubseteq^{f} I_1' \cup I_2'}\ \sqsubseteq\text{-Comp} \qquad \frac{\begin{array}{c} p \vdash_{\mathsf{SL}} p' \qquad q \vdash_{\mathsf{SL}} q' \\ \{r : \exists \bar{\mathrm{X}}.\ p' \rightsquigarrow q'\} \sqsubseteq^{f} \emptyset \end{array}}{\{r : \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\} \sqsubseteq^{f} \emptyset}\ \sqsubseteq\text{-AWeak}$$

$$\frac{f \uplus p \vdash_{\mathsf{SL}} \mathsf{false}}{\{r\colon \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\} \sqsubseteq^{f} \emptyset}\ \sqsubseteq\text{-LFalse} \qquad \frac{(p \mathbin{-\!\!\circledast} (p \uplus f)) * q \vdash_{\mathsf{SL}} \mathsf{false}}{\{r : \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\} \sqsubseteq^{f} \emptyset}\ \sqsubseteq\text{-RFalse}$$

$$\frac{\mathsf{exact}\,(r') \quad f \perp p}{\{r : \exists \bar{\mathrm{X}}.\ p * r' \rightsquigarrow q * r'\} \sqsubseteq^{f} \emptyset}\ \sqsubseteq\text{-Exact}$$

$$\frac{f \uplus p \vdash_{\mathsf{SL}} \bigvee_{j \in J} f \uplus (p * r_j) \qquad \mathsf{exact}\,(r_j) \ \text{ for all } j \in J}{\{r : \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\} \ \equiv^{f} \ \bigcup_{j \in J} \{r : \exists \bar{\mathrm{X}}.\ p * r_j \rightsquigarrow q * r_j\}}\ \sqsubseteq\text{-Catalyst}$$

$$\frac{\mathrm{X} \notin fv(r)}{\{r : \exists \mathrm{X} {\in} J.\ \exists \bar{\mathrm{Y}}.\ p \rightsquigarrow q\} \equiv^{\mathsf{true}} \bigcup_{\mathrm{X} \in J} \{r : \exists \bar{\mathrm{Y}}.\ p \rightsquigarrow q\}}\ \sqsubseteq\text{-AEquiv}$$

$$\frac{}{\bigcup_{j \in J,\, k \in K} \{r : \exists \bar{\mathrm{X}}.\ p_j \rightsquigarrow q_k\} \equiv^{\mathsf{true}} \left\{r : \exists \bar{\mathrm{X}}.\ \bigvee_{j \in J} p_j \rightsquigarrow \bigvee_{k \in K} q_k\right\}}\ \sqsubseteq\text{-ADisj}$$

$$\frac{}{\mathsf{true} \rhd I}\ \rhd\text{-True} \qquad \frac{f \rhd I_1 \quad f \rhd I_2}{f \rhd I_1 \cup I_2}\ \rhd\text{-Comp} \qquad \frac{f \blacktriangleright I}{f \rhd I}\ \rhd\text{-}\blacktriangleright$$

$$\frac{f \rhd I' \quad I' \sqsubseteq^{f} I}{f \rhd I}\ \rhd\text{-Mon} \qquad \frac{p \vdash_{\mathsf{SL}} p' \quad q \vdash_{\mathsf{SL}} q' \quad f \rhd \{r\colon \exists \bar{\mathrm{X}}. p' \rightsquigarrow q'\}}{f \rhd \{r\colon \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\}}\ \rhd\text{-AWeak}$$

$$\frac{f \rhd \bigcup_{\mathrm{X} \in J} \{r\colon \exists \bar{\mathrm{Y}}. p \rightsquigarrow q\} \quad \mathrm{X} \notin fv(r)}{f \rhd \{r\colon \exists \mathrm{X} {\in} J.\ \exists \bar{\mathrm{Y}}.\ p \rightsquigarrow q\}}\ \rhd\text{-AEquiv} \qquad \frac{f \perp p}{f \rhd \{r\colon \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\}}\ \rhd\text{-Disjoint}$$

$$\frac{\mathsf{exact}\,(r') \quad f \rhd \{r\colon \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\}}{f \rhd \{r\colon \exists \bar{\mathrm{X}}.\ p * r' \rightsquigarrow q * r'\}}\ \rhd\text{-Exact} \qquad \frac{(p \mathbin{-\!\!\circledast} (f \uplus p)) * q \vdash_{\mathsf{SL}} f}{f \rhd \{r\colon \exists \bar{\mathrm{X}}.\ p \rightsquigarrow q\}}\ \rhd\text{-Inv}$$

Figure 9.2.: Interference shifting (above) and fencing judgements (below)

precondition of the action is not compatible with f. Analogously, the ($\sqsubseteq$-RFalse) rule describes the case when the postcondition is incompatible with f. Lastly, the ($\sqsubseteq$-Exact) rule describes the case when the precondition of the action is entirely outside f. The notation $p \perp q$ is the assertion counterpart of the disjointness relation (Def. 93) and ensures that the states described by p and q are disjoint. That is, whenever $\Gamma, l_1 \vDash_{\text{SL}} p$ and $\Gamma, l_2 \vDash_{\text{SL}} q$, then $l_1 \perp l_2$ as defined in Def. 93. The $p \perp q$ can be expressed in separation logic as follows:

$$p \perp q \quad \Leftrightarrow \quad p \vdash_{\mathsf{SL}} \neg\left(\mathsf{true} * (\neg\mathsf{emp} \wedge (\mathsf{true} \mathrel{-\!\!\circledast} q))\right)$$

Recall that the states described by a fence assertion f captures the set of all possible states the subjective shared state may be in and thus denotes an invariant of the subjective view. The ($\sqsubseteq$-Catalyst) rule is a shifting equivalence that uses the knowledge embodied by the invariant f to rewrite actions into equivalent ones. More precisely, if whenever the precondition p of the action agrees with f, then a catalyst r_j is also true, then adding r_j as a neutral part of the action does not alter the action behaviour. We can use this rule (with the single $r_0 = \mathrm{X} \mapsto \mathrm{V}$) to justify the shiftings of the token ring algorithm in §8. The exactness of each r_j ensures that no piece of the state in r_j is mutated by the action. In general, it may not be the case that a single exact assertion can be added, but it may be the case that a disjunction of exact facts holds. The last two shifting equivalences ($\sqsubseteq$-AEquiv and $\sqsubseteq$-ADisj) are straightforward.

The bottom part of Fig. 9.2 presents judgements that partially axiomatise the *fencing* relation $f \rhd I$. Most of these rules are similar to those for local fencing $f \blacktriangleright I$ in Fig. 9.1. The ($\rhd$-True) rule states that true fences any interference assertion. The ($\rhd$-Comp) rule states that fencing can be checked per action. The next two judgements allow for proof reuse. Recall that the semantic definition of the fencing relation $\rhd$ is subsumed by that of local fencing $\blacktriangleright$. The ($\rhd$-$\blacktriangleright$) rule thus states that once the local fencing judgement $f \blacktriangleright I$ is established, it immediately establishes the weaker fencing judgement $f \rhd I$. Note that the other direction is not valid in general as fencing lacks the confinement condition required by local fencing. The ($\rhd$-Mon) rule states that the fencing relation is preserved by interference shifting. Similarly, the other direction of this judgement is not valid in general since a fence f for a smaller interference assertion need not be a fence for a larger interference assertion.

As with the analogous local fencing judgement in Fig. 9.1, the ($\triangleright$-AWeak) rule simply allows us to check fencing for the weaker action of $r : \exists\bar{\mathrm{x}}.\, p' \rightsquigarrow q'$ which contain the action of $r : \exists\bar{\mathrm{x}}.\, p \rightsquigarrow q$. Similarly, the ($\triangleright$-AEquiv) rule allows us to check fencing for an equivalent set of actions. The ($\triangleright$-Disjoint) rule states that contrarily to local fencing, actions may have effects outside of the fence. If the action precondition does not intersect with the fence (i.e. $f \perp p$), then its effect is entirely outside the fence and the action may be ignored. The ($\triangleright$-Exact) judgement states that, as for local fencing, neutral parts of actions may be ignored.

Let us now focus on the ($\triangleright$-Inv) judgement which states that whenever p and q are disjoint in an action $r : \exists\bar{\mathrm{x}}.\, p \rightsquigarrow q$ (e.g. when their common parts have been removed using the previous rule), then applying the action must preserve the fence f. Contrarily to the case of local fencing, the action is allowed to act partly outside of f, hence the state on which the action is applied is $f \uplus\!\!\!*\, p$. However, the whole of the postcondition q is then added, and the resulting state must still be in f. One might instead have expected that only parts of the resulting state need to be represented in f, to mimic the relationship between f and p (and indeed, this is all that is required for stability, as we shall see shortly). However, we do need the full q. Recall from Def. 111 that the shifting judgement $I \sqsubseteq^{f} I'$ requires that the two interference assertions I and I' have the same effect even after an arbitrary number of steps. Ignoring parts of q would be unsound as there would be no guarantee that I accounts for all possible actions on those discarded parts of q, since it would not be a part of f. Hence, we may end up with a new interference assertion I' that breaks the original action model closure property.

Example 13 (Shifting and fencing judgements)**.** Using the shifting and fencing rules in Fig. 9.2, we can now establish the $I \sqsubseteq^{P_0} I'_{\mathtt{y}}$ judgement used in §8, with P_0, I and $I'_{\mathtt{y}}$ defined below (repeated from §8):

$$P_0 \triangleq \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0 \qquad I \triangleq I_0 \cup I_2 \cup I_3 \qquad I'_{\mathtt{y}} \triangleq I_1 \cup I_2 \cup I_3$$

$$I_0 \triangleq \Big\{ [\mathsf{a}_{\mathtt{x}}]^{\mathrm{T}} : \exists \mathrm{v} {\in} \{0 \cdots 9\}.\, \mathtt{x} \mapsto \mathrm{v} * \mathtt{z} \mapsto \mathrm{v} \rightsquigarrow \mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{z} \mapsto \mathrm{v}$$

$$I_1 \triangleq \left\{ [\mathsf{a}_{\mathtt{x}}]^{\mathrm{T}} : \exists \mathrm{v}{\in}\{0\cdots 9\}.\, \mathtt{x} \mapsto \mathrm{v} * \mathtt{y} \mapsto \mathrm{v} * \mathtt{z} \mapsto \mathrm{v} \leadsto \mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v} * \mathtt{z} \mapsto \mathrm{v} \right.$$

$$I_2 \triangleq \left\{ \left[\mathsf{a}_{\mathtt{y}}\right]^{\mathrm{T}} : \exists \mathrm{v}{\in}\{0\cdots 9\}.\, \mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v} \leadsto \mathtt{x} \mapsto \mathrm{v}{+}1 * \mathtt{y} \mapsto \mathrm{v}{+}1 \right.$$

$$I_3 \triangleq \left\{ [\mathsf{a}_{\mathtt{z}}]^{\mathrm{T}} : \exists \mathrm{v}{\in}\{0\cdots 9\}.\, \mathtt{y} \mapsto \mathrm{v}{+}1 * \mathtt{z} \mapsto \mathrm{v} \leadsto \mathtt{y} \mapsto \mathrm{v}{+}1 * \mathtt{z} \mapsto \mathrm{v}{+}1 \right.$$

Let us define the fence assertion F and interference assertions I'_1, I'_2, I'_3, K:

$$F \triangleq \bigvee_{\mathrm{w}=0}^{10} \begin{pmatrix} (\mathrm{X} \mapsto \mathrm{W} * \mathrm{Y} \mapsto \mathrm{W} * \mathrm{Z} \mapsto \mathrm{W}) \\ \vee\, (\mathrm{X} \mapsto \mathrm{W}{+}1 * \mathrm{Y} \mapsto \mathrm{W} * \mathrm{Z} \mapsto \mathrm{W}) \\ \vee\, (\mathrm{X} \mapsto \mathrm{W}{+}1 * \mathrm{Y} \mapsto \mathrm{W}{+}1 * \mathrm{Z} \mapsto \mathrm{W}) \end{pmatrix}$$

$$I'_0 \triangleq \bigcup_{j=0}^{9} I^{\mathtt{x}}_j \qquad I^{\mathtt{x}}_j \triangleq \left\{ [\mathsf{a}_{\mathtt{x}}]^{\mathrm{T}} : \mathtt{x} \mapsto j * \mathtt{z} \mapsto j \leadsto \mathtt{x} \mapsto j{+}1 * \mathtt{z} \mapsto j \right\}$$

$$I'_2 \triangleq \bigcup_{j=0}^{9} I^{\mathtt{y}}_j \qquad I^{\mathtt{y}}_j \triangleq \left\{ \left[\mathsf{a}_{\mathtt{y}}\right]^{\mathrm{T}} : \mathtt{x} \mapsto j{+}1 * \mathtt{y} \mapsto j \leadsto \mathtt{x} \mapsto j{+}1 * \mathtt{y} \mapsto j{+}1 \right\}$$

$$I'_3 \triangleq \bigcup_{j=0}^{9} I^{\mathtt{z}}_j \qquad I^{\mathtt{z}}_j \triangleq \left\{ [\mathsf{a}_{\mathtt{z}}]^{\mathrm{T}} : \mathtt{y} \mapsto j{+}1 * \mathtt{z} \mapsto j \leadsto \mathtt{y} \mapsto j{+}1 * \mathtt{z} \mapsto j{+}1 \right\}$$

$$K \triangleq \bigcup_{j=0}^{9} K_j \qquad K_j \triangleq \left\{ [\mathsf{a}_{\mathtt{x}}]^{\mathrm{T}} : \mathtt{x} \mapsto j * \mathtt{y} \mapsto j * \mathtt{z} \mapsto j \leadsto \mathtt{x} \mapsto j{+}1 * \mathtt{y} \mapsto j * \mathtt{z} \mapsto j \right\}$$

We can then establish the validity of $I \sqsubseteq^{P_0} I'_{\mathtt{y}}$ as shown below:

$$\dfrac{\dfrac{}{P_0 \vdash F} \qquad \dfrac{(\dagger) \qquad \dfrac{\dfrac{}{I_0 \sqsubseteq^F I'_0}\,\sqsubseteq\text{-AEQUIV} \qquad \dfrac{(\dagger\dagger)}{I'_0 \sqsubseteq^F I_1}\,\sqsubseteq\text{-TRAN}}{I_0 \sqsubseteq^F I_1}\,\sqsubseteq\text{-TRAN} \qquad (\ddagger)}{I \sqsubseteq^F I'_{\mathtt{y}}}\,\sqsubseteq\text{-COMP}}{I \sqsubseteq^{P_0} I'_{\mathtt{y}}}\,\sqsubseteq\text{-WEAK}$$

with

$$\dfrac{\dfrac{(1) \quad (2) \quad (3)}{F \rhd I}\,\rhd\text{-COMP}}{(\dagger)} \qquad\qquad \dfrac{\dfrac{}{I_2 \cup I_3 \sqsubseteq^F I_2 \cup I_3}\,\sqsubseteq\text{-REFL}}{(\ddagger)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{j{\in}\{0..9\}}{\left[\mathtt{x}{\mapsto} j * \mathtt{z}{\mapsto} j \mathrel{-\!\!\circledast} \left(F \uplus (\mathtt{x}{\mapsto} j * \mathtt{z}{\mapsto} j)\right)\right] * \mathtt{x}{\mapsto} j{+}1 * \mathtt{z}{\mapsto} j \;\vdash_{\mathsf{SL}} F}}{F \rhd I^{\mathtt{x}}_j \quad \text{for all } j{\in}\{0..9\}}\,\rhd\text{-INV}}{F \rhd I'_0}\,\rhd\text{-COMP}}{\dfrac{F \rhd I_0}{(1)}}\,\rhd\text{-EQUIV}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{j\in\{0..9\}}{\left(\begin{array}{l}\big[\mathtt{x}\mapsto j{+}1 * \mathtt{y}\mapsto j \mathrel{-\!\!\circledast} \big(F \uplus (\mathtt{x}\mapsto j{+}1 * \mathtt{y}\mapsto j)\big)\big]\\ * \ \mathtt{x}\mapsto j{+}1 * \mathtt{y}\mapsto j{+}1\end{array}\right) \vdash_{\mathsf{SL}} F}}{F \rhd I^{\mathtt{y}}_j \quad \text{for all } j\in\{0..9\}}\ \rhd\text{-}\textsc{Inv}}{\dfrac{F \rhd I'_2}{F \rhd I_2}\ \rhd\text{-}\textsc{Equiv}}\ \rhd\text{-}\textsc{Comp}}{(2)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{j\in\{0..9\}}{\left(\begin{array}{l}\big[\mathtt{y}\mapsto j{+}1 * \mathtt{z}\mapsto j \mathrel{-\!\!\circledast} \big(F \uplus (\mathtt{y}\mapsto j{+}1 * \mathtt{z}\mapsto j)\big)\big]\\ * \ \mathtt{y}\mapsto j{+}1 * \mathtt{z}\mapsto j{+}1\end{array}\right) \vdash_{\mathsf{SL}} F}}{F \rhd I^{\mathtt{z}}_j \quad \text{for all } j\in\{0..9\}}\ \rhd\text{-}\textsc{Inv}}{\dfrac{F \rhd I'_3}{F \rhd I_3}\ \rhd\text{-}\textsc{Equiv}}\ \rhd\text{-}\textsc{Comp}}{(3)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\dagger)}{F \rhd I}}{F \rhd I_0}\ \rhd\text{-}\textsc{Comp} \qquad \dfrac{}{I_0 \sqsubseteq^F I'_0}\ \sqsubseteq\text{-}\mathrm{A}\textsc{Equiv}}{F \rhd I'_0}\ \rhd\text{-}\textsc{Mon} \qquad (\ddagger\ddagger)}{I'_0 \sqsubseteq^F K}\ \sqsubseteq\text{-}\textsc{Comp} \qquad \dfrac{}{K \sqsubseteq^F I_1}\ \sqsubseteq\text{-}\mathrm{A}\textsc{Equiv}}{(\dagger\dagger)}$$

and

$$\dfrac{\dfrac{\dfrac{j\in\{0..9\}}{F\uplus(\mathtt{x}\mapsto j * \mathtt{z}\mapsto j) \ \vdash_{\mathsf{SL}} F\uplus(\mathtt{x}\mapsto j * \mathtt{z}\mapsto j * \mathtt{y}\mapsto j)} \qquad \dfrac{j\in\{0..9\}}{\mathsf{exact}\,(\mathtt{y}\mapsto j)}}{I^{\mathtt{x}}_j \sqsubseteq^F K_j \quad \text{for all } j\in\{0..9\}}\ \sqsubseteq\text{-}\textsc{Catalyst}}{(\ddagger\ddagger)}$$

Example 14 (Shifting and fencing judgements (continued))**.** Using the shifting and fencing rules in Fig. 9.2, we can now establish the $I'_{\mathtt{y}} \sqsubseteq^{Q_0} I_{\mathtt{y}}$ judgement used in §8, where Q_0 and $I_{\mathtt{y}}$ are as defined below (repeated from §8) and $I'_{\mathtt{y}}$, I_1, I_2, I_3 are as defined above in Example 13:

$$Q_0 \triangleq \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 \qquad\qquad I_{\mathtt{y}} \triangleq I_1 \cup I_2$$

Let I'_2 and I'_3 be as defined above in Example 13 and let us define the fence assertion F_{y} and the interference assertion I'_1 as follows:

$$F_{\mathsf{y}} \triangleq \bigvee_{\mathrm{w}=0}^{10} (\mathrm{x} \mapsto \mathrm{w} * \mathrm{y} \mapsto \mathrm{w}) \vee (\mathrm{x} \mapsto \mathrm{w}{+}1 * \mathrm{y} \mapsto \mathrm{w})$$

$$I'_1 \triangleq \bigcup_{j=0}^{9} I^{\mathsf{x}}_j \qquad I^{\mathsf{x}}_j \triangleq \left\{ [\mathsf{a}_{\mathrm{x}}]^{\mathrm{T}} : \mathsf{x} \mapsto j * \mathsf{y} \mapsto j * \mathsf{z} \mapsto j \rightsquigarrow \mathsf{x} \mapsto j{+}1 * \mathsf{y} \mapsto j * \mathsf{z} \mapsto j \right\}$$

We can then establish the validity of the $I'_{\mathsf{y}} \sqsubseteq^{Q_0} I_{\mathsf{y}}$ judgement as shown in the derivation below:

$$\dfrac{\overline{Q_0 \vdash F_{\mathsf{y}}} \qquad \dfrac{(\dagger) \qquad \dfrac{\overline{\mathsf{exact}\,(\mathsf{y} \mapsto \mathrm{v}{+}1)} \qquad \overline{F_{\mathsf{y}} \perp \mathsf{z} \mapsto \mathrm{v}}}{I_3 \sqsubseteq^{F_{\mathsf{y}}} \emptyset}\ \sqsubseteq\text{-Exact} \qquad (\ddagger)}{I'_{\mathsf{y}} \sqsubseteq^{F_{\mathsf{y}}} I_{\mathsf{y}}}\ \sqsubseteq\text{-Comp}}{I'_{\mathsf{y}} \sqsubseteq^{Q_0} I_{\mathsf{y}}}\ \sqsubseteq\text{-Weak}$$

with

$$\dfrac{\dfrac{(1) \quad (2) \quad (3)}{F_{\mathsf{y}} \rhd I}\ \sqsubseteq\text{-Comp}}{(\dagger)} \qquad\qquad \dfrac{\dfrac{}{I_1 \cup I_2 \sqsubseteq^{F_{\mathsf{y}}} I_1 \cup I_2}\ \sqsubseteq\text{-Refl}}{(\ddagger)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{j{\in}\{0..9\}}{\mathsf{exact}\,(\mathsf{z}{\mapsto}j)} \qquad \dfrac{\dfrac{j{\in}\{0..9\}}{\begin{pmatrix}[\mathsf{x}{\mapsto}j * \mathsf{y}{\mapsto}j \mathrel{-\!\!\circledast} (F_{\mathsf{y}} \uplus (\mathsf{x}{\mapsto}j * \mathsf{y}{\mapsto}j))] \\ * \ \mathsf{x}{\mapsto}j{+}1 * \mathsf{y}{\mapsto}j\end{pmatrix} \vdash_{\mathsf{SL}} F_{\mathsf{y}}}}{F_{\mathsf{y}} \rhd \left\{[\mathsf{a}_{\mathrm{x}}]^{\mathrm{T}} : \mathsf{x}{\mapsto}j * \mathsf{y}{\mapsto}j \rightsquigarrow \mathsf{x}{\mapsto}j{+}1 * \mathsf{y}{\mapsto}j\right\}}\ \rhd\text{-Inv}}{F_{\mathsf{y}} \rhd I^{\mathsf{x}}_j \quad \text{for all } j{\in}\{0..9\}}\ \rhd\text{-Exact}}{F_{\mathsf{y}} \rhd I'_1}\ \rhd\text{-Comp}}{F_{\mathsf{y}} \rhd I_1}\ \rhd\text{-Equiv}}{(1)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{j{\in}\{0..9\}}{\begin{pmatrix}[\mathsf{x}{\mapsto}j{+}1 * \mathsf{y}{\mapsto}j \mathrel{-\!\!\circledast} (F_{\mathsf{y}} \uplus (\mathsf{x}{\mapsto}j{+}1 * \mathsf{y}{\mapsto}j))] \\ * \ \mathsf{x}{\mapsto}j{+}1 * \mathsf{y}{\mapsto}j{+}1\end{pmatrix} \vdash_{\mathsf{SL}} F_{\mathsf{y}}}}{F_{\mathsf{y}} \rhd I^{\mathsf{y}}_j \quad \text{for all } j{\in}\{0..9\}}\ \rhd\text{-Inv}}{F_{\mathsf{y}} \rhd I'_2}\ \rhd\text{-Comp}}{F_{\mathsf{y}} \rhd I_2}\ \rhd\text{-Equiv}}{(2)}$$

and

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{j\in\{0..9\}}{\mathsf{exact}\,(\mathsf{y}\mapsto j{+}1)} \qquad \dfrac{\dfrac{j\in\{0..9\}}{F_{\mathsf{y}} \perp \mathsf{z}\mapsto j}}{F_{\mathsf{y}} \rhd \left\{ [\mathsf{a_z}]^{\mathrm{T}} : \mathsf{z}\mapsto j \rightsquigarrow \mathsf{z}\mapsto j{+}1 \right\}}\ \rhd\text{-}\textsc{Disjoint}}{F_{\mathsf{y}} \rhd I^{\mathsf{z}}_{j} \quad \text{for all } j\in\{0..9\}}\ \rhd\text{-}\textsc{Exact}}{F_{\mathsf{y}} \rhd I'_3}\ \rhd\text{-}\textsc{Comp}}{\overline{F_{\mathsf{y}} \rhd I_3}}\ \rhd\text{-}\textsc{Equiv}}{(3)}$$

Stability We shortly present a number of judgements that reduce stability checks to standard separation logic entailments similar to those of interference confinement, shifting and fencing. Unlike the previous judgements studied so far where the boxed assertions were ignored, the boxed assertions are at the core of stability checks as they are the only source of interference from the environment and local assertions are always stable. Our stability judgements must thus account for subjective assertions which may include nested boxed assertions such as $\boxed{P * \boxed{Q}_{I'}}_{I}$. Although CoLoSL allows for such nested subjective assertions, it is often simpler to *flatten* them into equivalent assertions with no nested boxes. We thus introduce *normalised flat assertions*, $\textsc{NFAst} \subset \textsc{Ast}$, to capture the subset of CoLoSL assertions that i) contain no nested subjected views; and ii) exclude the overlapping conjunction connective $\uplus$ from the top-level assertions. More precisely, the set of flat assertions may include the $\uplus$ connective at the level of local assertions $\textsc{LAst}$, but not at the top-level assertion language. This is because while $\uplus$ between local assertions describes overlapping states, its behaviour at the top-level may be equivalently captured by $*$. More concretely, an assertion of the form $p \uplus \boxed{P}_I$ (where p is a local assertion) is equivalent to $p * \boxed{P}_I$. Similarly, an assertion of the form $\boxed{P}_I \uplus \boxed{Q}_{I'}$ is equivalent to $\boxed{P}_I * \boxed{Q}_{I'}$.

An assertion is in a (flat) normalised form if it comprises a disjunction of $*$-separated local and boxed assertions, where the boxed assertions contain local assertions only. More concretely, a normalised assertion is of the form $\bigvee\limits_{j\in J} \left(p_j * \boxed{q^1_j}_{I^1_j} * \cdots * \boxed{q^n_j}_{I^n_j} \right)$ where $p_j, q^1_j, \ldots, q^n_j \in \textsc{LAst}$ are local assertions. We provide a *flattening mechanism* in order to rewrite a given CoLoSL assertion as an equivalent normalised flat assertion. Our flattening mechanism assumes that the given assertion is in *prenex normal form* [28]. An assertion is in the prenex normal form if it is written as a string of quantifiers (referred to as the prefix) followed by a quantifier-free part (referred to as the matrix). With the exception of the parametric assertions provided by the user (i.e. the logical state assertions LSAST in Par. 24 and user-defined capability assertions CAST in Par. 25), it is straightforward to transform CoLoSL assertions to equivalent assertions in the prenex normal form. In particular, as the CoLoSL assertion language includes the existential quantifier $\exists$ only (and thus the reordering of quantifiers does not alter the assertion semantics), a CoLoSL assertion may be transformed into the prenex formal form by i) renaming the repeated logical variables by capture-avoiding substitution; and ii) promoting all existential quantifiers to the beginning of the assertion using the standard prenex conversion rules [28]. We write $\circledQ P$ for an assertion in the prenex normal form with prefix $\circledQ$ and matrix P.

Definition 118 (Normalised flat assertions)**.** Given the CoLoSL assertions AST, local assertions LAST and interference assertions IAST (Def. 103), the set of *normalised flat assertions*, NFAST $\subset$ AST, is defined by the following grammar, where $p^{\not\emptyset} \in$ LAST denotes a quantifier-free local assertion, $I \in$ IAST, and $\ddagger$ denotes a parametric quantifier included in the grammar of logical state assertions (Par. 24) or user-defined capability assertions (Par. 25):

$$\begin{aligned}
\text{NFAST} \ni P^{\mathsf{n}}, Q^{\mathsf{n}} &::= P^{\not\emptyset} \mid \exists \mathrm{x}.\ P^{\mathsf{n}} \mid \ddagger P^{\mathsf{n}} \\
\text{QFAST} \ni P^{\not\emptyset}, Q^{\not\emptyset} &::= p^{\not\emptyset} * P^{\mathsf{b}} \mid P^{\not\emptyset} \vee Q^{\not\emptyset} \\
\text{BAST} \ni P^{\mathsf{b}}, Q^{\mathsf{b}} &::= \mathsf{emp} \mid \boxed{p^{\not\emptyset}}_{I} \mid P^{\mathsf{b}} * Q^{\mathsf{b}}
\end{aligned}$$

The *flattening normalisation* function, $\mathsf{nf}\,(.) :$ AST $\to$ NFAST, for assertions in the prenex normal form is defined as follows:

$$\mathsf{nf}\,(\circledQ P) \triangleq \circledQ\, \mathsf{nf}\,(P)$$

with $\mathsf{nf}\,(.)$ defined inductively over the structure of quantifier-free CoLoSL assertions in AST as follows:

$$\mathsf{nf}\left(p^{\not\emptyset}\right) \triangleq p^{\not\emptyset} \qquad\qquad \mathsf{nf}\,(P \vee Q) \triangleq \mathsf{nf}\,(P) \vee \mathsf{nf}\,(Q)$$

$$\mathsf{nf}\,(P * Q) \triangleq \bigvee_{j\in J,\, k\in K} \left(p_j^{\emptyset} * q_k^{\emptyset} * P_j^{\mathsf{b}} * Q_k^{\mathsf{b}}\right) \quad \text{where} \quad \mathsf{nf}\,(P) \triangleq \bigvee_{j\in J} \left(p_j^{\emptyset} * P_j^{\mathsf{b}}\right)$$
$$\text{and} \quad \mathsf{nf}\,(Q) \triangleq \bigvee_{k\in K} \left(q_k^{\emptyset} * Q_k^{\mathsf{b}}\right)$$

$$\mathsf{nf}\,(P \uplus Q) \triangleq \bigvee_{j\in J,\, k\in K} \left((p_j^{\emptyset} \uplus q_k^{\emptyset}) * P_j^{\mathsf{b}} * Q_k^{\mathsf{b}}\right) \quad \text{where} \quad \mathsf{nf}\,(P) \triangleq \bigvee_{j\in J} \left(p_j^{\emptyset} * P_j^{\mathsf{b}}\right)$$
$$\text{and} \quad \mathsf{nf}\,(Q) \triangleq \bigvee_{k\in K} \left(q_k^{\emptyset} * Q_k^{\mathsf{b}}\right)$$

$$\mathsf{nf}\,\left(\boxed{P}_I\right) \triangleq \bigvee_{j\in J} \left(\boxed{p_j^{\emptyset}}_I * P_j^{\mathsf{b}}\right) \quad \text{where} \quad \mathsf{nf}\,(P) \triangleq \bigvee_{j\in J} \left(p_j^{\emptyset} * P_j^{\mathsf{b}}\right)$$

Lemma 12 (Flattening normalisation)**.** *For all assertions $P \in$ Ast (Def. 103) in the prenex normal form:*

$$\frac{}{\vdash P \Leftrightarrow \mathsf{nf}\,(P)}$$

where $\mathsf{nf}\,(.)$ *denotes the flattening normalisation function in Def. 118.*

Proof. The full proof is given in §C (Lemma 43).

We now present a number of syntactic judgements in Fig. 9.3 that allows us to ascertain the stability of normalised flat assertions without performing the necessary semantic checks. These judgements reduce stability checks to standard separation logical entailments. We assume that the logical variables $\bar{\mathrm{x}}$ do not appear free in p, q and r.

The (st-$\textcircled{\mathcal{Q}}$) rule states that the quantifiers can be eliminated; the (st-Local) rule states that local assertions are always stable. The (st-$\vee$) judgement states that the stability of disjunctions can be checked piecemeal.

The (st-$\rhd$) rule allows for proof reuse: once the fencing judgement $p \rhd I$ is established, it immediately implies that p is stable under I. The other direction depicted below is not valid in general:

$$\cancel{\frac{\mathsf{stable}\left(\boxed{p}_I\right)}{p \rhd I}}$$

This is because the stability of p under I may omit part of the state resulting from an action application to re-establish p, which is not allowed

$$\frac{\mathsf{stable}\,(P)}{\mathsf{stable}\,(\textcircled{$\mathcal{Q}$}P)}\;\textsf{st-}\textcircled{$\mathcal{Q}$} \qquad \frac{}{\mathsf{stable}\,\left(p^{\emptyset}\right)}\;\textsf{st-}\textsc{Local} \qquad \frac{\mathsf{stable}\,\left(P^{\emptyset}\right) \quad \mathsf{stable}\,\left(Q^{\emptyset}\right)}{\mathsf{stable}\,\left(P^{\emptyset} \vee Q^{\emptyset}\right)}\;\textsf{st-}\vee$$

$$\frac{p \rhd I}{\mathsf{stable}\,\left(\boxed{p}_{I}\right)}\;\textsf{st-}\rhd \qquad \frac{\forall j \in J.\,\mathsf{stable}\left(p_j^{\emptyset}, I_j, q^{\emptyset}, \biguplus_{j\in J} p_j^{\emptyset}\right)}{\mathsf{stable}\left(q^{\emptyset} * \left(\circledast_{j\in J} \boxed{p_j^{\emptyset}}_{I_j}\right)\right)}\;\textsf{st-}\textsc{Box}$$

$$\frac{}{\mathsf{stable}\,(p, \emptyset, q, r)}\;\textsf{st-}\textsc{Empty} \qquad \frac{\mathsf{stable}\,(p, I_1, q, r) \quad \mathsf{stable}\,(p, I_2, q, r)}{\mathsf{stable}\,(p, I_1 \cup I_2, q, r)}\;\textsf{st-}\textsc{Comp}$$

$$\frac{\mathsf{stable}\,(p, I, q, r)}{\mathsf{stable}\,(p, I, q, r * r')}\;\textsf{st-}\textsc{Weak} \qquad \frac{q * r * r' \vdash_{\mathsf{SL}} \mathsf{false}}{\mathsf{stable}\,(p, \{r' : \exists\bar{\mathrm{x}}.\; p' \rightsquigarrow q'\}, q, r)}\;\textsf{st-}\textsc{CapFalse}$$

$$\frac{q * (r \uplus p') \vdash_{\mathsf{SL}} \mathsf{false}}{\mathsf{stable}\,(p, \{r' : \exists\bar{\mathrm{x}}.\; p' \rightsquigarrow q'\}, q, r)}\;\textsf{st-}\mathrm{L}\textsc{False}$$

$$\frac{q * (p' \mathrel{-\!\!\circledast} (r \uplus p')) * q' \vdash_{\mathsf{SL}} \mathsf{false}}{\mathsf{stable}\,(p, \{r' : \exists\bar{\mathrm{x}}.\; p' \rightsquigarrow q'\}, q, r)}\;\textsf{st-}\mathrm{R}\textsc{False}$$

$$\frac{(p' \mathrel{-\!\!\circledast} (p \uplus p')\,) * q' \vdash_{\mathsf{SL}} p * \mathsf{true}}{\mathsf{stable}\,(p, \{r' : \exists\bar{\mathrm{x}}.\; p' \rightsquigarrow q'\}, q, r)}\;\textsf{st-}\textsc{Inv}$$

Figure 9.3.: CoLoSL stability judgements

in fencing. For instance, let p and I be defined as follows:

$$p \triangleq \mathtt{x} \mapsto 1 \vee \mathtt{x} \mapsto 2 \qquad\qquad I \triangleq \{\mathsf{true} : \mathtt{x} \mapsto 1 \rightsquigarrow \mathtt{x} \mapsto 2 * \mathtt{y} \mapsto 2\}$$

Note that while $\mathsf{stable}\left(\boxed{p}_{I}\right)$ holds, the p does *not* fence I: $p \not\rhd I$.

The ($\textsf{st-}\textsc{Box}$) rule states that the stability of $q^{\emptyset} * \circledast_{i\in J} \boxed{p_j^{\emptyset}}_{I_j}$ is distilled to establishing for each $j \in J$ that the four-place predicate $\mathsf{stable}(p_j^{\emptyset}, I_j, q^{\emptyset}, \biguplus_{j\in J} p_j^{\emptyset})$ holds. Each four-place predicate states that the subjective assertion $p_j^{\emptyset}$ is stable under the associated interference assertion I_j, a local context $q^{\emptyset}$, and a shared context made of the $\uplus$-combination of all the subjective assertions (including $p_j^{\emptyset}$ itself). The next two rules, ($\textsf{st-}\textsc{Empty}$) and ($\textsf{st-}\textsc{Comp}$), state that checking the four-place predicate in turn reduces to checking the stability under each action of I_j. The ($\textsf{st-}\textsc{Weak}$) rule states that the context resources with respect to which stability is established may be weakened. The last four rules deal with checking stability for a single action, in a similar way to the fencing rules above.

The first of these rules (st-CapFalse) is unfamiliar. Unlike fencing, we must establish the assertion stability only against those actions for which the environment may have the associated capability. If the capability required by the action cannot exist separately from those held by the assertion (that is, the required capability is not compatible with the resources of the context $q * r$) then the environment cannot possibly own the capability to perform the action and thus the assertion is immediately stable with respect to that action. The (st-LFalse) states that an action whose precondition is incompatible with the context resources $q * r$ cannot possibly fire and thus the assertion p is trivially stable under this action. Similarly, the (st-RFalse) rule states that an action whose postcondition is incompatible with the context resources cannot fire and thus the assertion p is trivially stable under this action. The last rule (st-Inv) checks that the assertion p is preserved by the effect of an action (and is thus stable under that action). Again, there is a crucial difference with the corresponding check for fencing: through the action postcondition q', the action may bring in newly-shared resources not previously in p, hence the result $p * \mathtt{true}$ (rather than p); however, the Forget principle allows us to immediately discard it, if necessary.

9.6. Programming Language and Proof Rules

We define the CoLoSL proof system for deriving local Hoare triples for a simple concurrent imperative programming language. The proof system and programming language of CoLoSL are defined as an instantiation of the Views framework [14].

Programming Language The CoLoSL programming language is the concurrent imperative language of the Views framework [14], instantiated with a set of *atomic* operations. This language comprises `skip`, sequential composition ($\mathtt{C}_1$`;` $\mathtt{C}_2$), non-deterministic choice ($\mathtt{C}_1 + \mathtt{C}_2$), loops ($\mathtt{C}^*$) and parallel composition ($\mathtt{C}_1 || \mathtt{C}_2$). The set of CoLoSL atomic commands comprises an *atomic* construct `<.>` enforcing an atomic behaviour when used with a *sequential* operation. The sequential operations of CoLoSL comprise *primitive* operations, `skip`, sequential composition, non-deterministic choice and loops, excluding atomic constructs and parallel composition. That is, atomic operations cannot be nested or contain parallel composition. CoLoSL is parametric in the choice of primitive operations allowing for its suitable instantiation depending on the programs being verified. For instance, in the token ring example of §8.2, the set of primitive operations comprises variable lookup and assignment. We proceed with the formalisation of CoLoSL programming language.

CoLoSL Parameter

Parameter 26 (Primitive operations)**.** Assume a set of primitive operations, $\mathtt{C}^{\mathrm{P}} \in \textsc{Prim}$.
Given the set of logical states $\textsc{LState}$ (Par. 22), assume a set of axioms associated with primitive operations:

$$\textsc{Ax}_{\mathrm{P}} : \mathcal{P}\left(\mathcal{P}\left(\textsc{LState}\right) \times \textsc{Prim} \times \mathcal{P}\left(\textsc{LState}\right)\right)$$

Definition 119 (Sequential operations)**.** Given the set of primitive operations $\textsc{Prim}$ (Def. 26), the set of *sequential operations*, $\mathtt{C}^{\mathrm{S}} \in \textsc{Seq}$, are defined by the following grammar, where $\mathtt{C}^{\mathrm{P}} \in \textsc{Prim}$:

$$\mathtt{C}^{\mathrm{S}} ::= \mathtt{C}^{\mathrm{P}} \mid \mathtt{skip} \mid \mathtt{C}^{\mathrm{S}}_1 \mathtt{;} \mathtt{C}^{\mathrm{S}}_2 \mid \mathtt{C}^{\mathrm{S}}_1 + \mathtt{C}^{\mathrm{S}}_2 \mid (\mathtt{C}^{\mathrm{S}})^*$$

Definition 120 (Sequential command axioms)**.** Given the set of logical states $\textsc{LState}$ (Par. 22) and the axiomatisation of primitive operations $\textsc{Ax}_{\mathrm{P}}$ (Par. 26), the set of *axioms for sequential operations*, $\textsc{Ax}_{\mathrm{S}} : \mathcal{P}\left(\mathcal{P}\left(\textsc{LState}\right) \times \textsc{Seq} \times \mathcal{P}\left(\textsc{LState}\right)\right)$ is defined as follows:

$$\begin{aligned}
\textsc{Ax}_{\mathrm{S}} &\triangleq \textsc{Ax}_{\mathrm{P}} \cup A_{\mathtt{skip}} \cup A_{;} \cup A_{+} \cup A_{*} \\
A_{\mathtt{skip}} &\triangleq \left\{ (L, \mathtt{skip}, L) \,\middle|\, L \in \mathcal{P}\left(\textsc{LState}\right) \right\} \\
A_{;} &\triangleq \left\{ (L, \mathtt{C}^{\mathrm{S}}_1 \mathtt{;} \mathtt{C}^{\mathrm{S}}_2, L') \,\middle|\, (L, \mathtt{C}^{\mathrm{S}}_1, L'') \in \textsc{Ax}_{\mathrm{S}} \wedge (L'', \mathtt{C}^{\mathrm{S}}_2, L') \in \textsc{Ax}_{\mathrm{S}} \right\} \\
A_{+} &\triangleq \left\{ (L, \mathtt{C}^{\mathrm{S}}_1 + \mathtt{C}^{\mathrm{S}}_2, L') \,\middle|\, (L, \mathtt{C}^{\mathrm{S}}_1, L') \in \textsc{Ax}_{\mathrm{S}} \wedge (L, \mathtt{C}^{\mathrm{S}}_2, L') \in \textsc{Ax}_{\mathrm{S}} \right\} \\
A_{*} &\triangleq \left\{ (L, (\mathtt{C}^{\mathrm{S}})^*, L) \,\middle|\, (L, \mathtt{C}^{\mathrm{S}}, L) \in \textsc{Ax}_{\mathrm{S}} \right\}
\end{aligned}$$

Definition 121 (Atomic operations)**.** Given the set of sequential operations $\textsc{Seq}$ (Def. 119), the set of *atomic operations*, $\texttt{C}^{\text{A}} \in \textsc{Atom}$, is defined by the folllowing grammar, where $\texttt{C}^{\text{S}} \in \textsc{Seq}$:

$$\texttt{C}^{\text{A}} ::= \texttt{<C}^{\text{S}}\texttt{>}$$

Definition 122 (Atomic axioms)**.** Given the set of CoLoSL worlds (Def. 102) and the rely relation $\mathcal{R}$ (Def. 112), a set of worlds $W \subseteq \textsc{World}$ is *stable*, written $\mathsf{stable}(W)$, if and only if:

$$\mathsf{stable}(W) \overset{\text{def}}{\iff} \forall w, w' \in \textsc{World}.\ w \in W \land (w, w') \in \mathcal{R} \Rightarrow w' \in W$$

The set of *stable world sets*, $\widehat{\textsc{World}} \subset \mathcal{P}(\textsc{World})$, is defined as follows:

$$\widehat{\textsc{World}} \triangleq \{W \mid W \in \mathcal{P}(\textsc{World}) \land \mathsf{stable}(W)\}$$

Given the partitioning relation $\Rrightarrow$ (Def. 116) and the axioms of sequential commands $\textsc{Ax}_{\text{S}}$ (Def. 120), the set of *axioms for atomic operations*, $\textsc{Ax}_{\text{A}} : \mathcal{P}\left(\widehat{\textsc{World}} \times \textsc{Atom} \times \widehat{\textsc{World}}\right)$, is defined as follows where $W_1, W_2 \in \widehat{\textsc{World}}$:

$$\textsc{Ax}_{\text{A}} \triangleq \left\{(W_1, \texttt{<C}^{\text{S}}\texttt{>}, W_2) \,\middle|\, \exists L_1, L_2.\ (L_1, \texttt{C}^{\text{S}}, L_2) \in \textsc{Ax}_{\text{S}} \land W_1 \Rrightarrow^{\{L_1\}\{L_2\}} W_2\right\}$$

Definition 123 (Programming language)**.** Given the set of atomic operations $\textsc{Atom}$ (Def. 121), the set of CoLoSL *operations*, $\texttt{C} \in \textsc{Op}$, is defined by the following grammar:

$$\texttt{C} ::= \texttt{C}^{\text{A}} \mid \texttt{skip} \mid \texttt{C}_1\texttt{;}\texttt{C}_2 \mid \texttt{C}_1 + \texttt{C}_2 \mid \texttt{C}_1 \texttt{||} \texttt{C}_2 \mid \texttt{C}^*$$

Proof Rules Our proof rules are of the form $\vdash \{P\}\ \texttt{C}\ \{Q\}$ where $P, Q \in \textsc{Ast}$ (Def. 103) and $\texttt{C} \in \textsc{Op}$ (Def. 123). Our triples carry an implicit assumption that the pre- and postconditions of their judgements are stable. Since we build CoLoSL as an instance of the Views framework [14], our proof rules correspond to those of Views with the atomic commands axiomatised as per Def. 122.

Definition 124 (Proof rules)**.** Given the set of CoLoSL assertions $\textsc{Ast}$ (Def. 103), and the set of CoLoSL operations $\textsc{Op}$ (Def. 123), The *proof* *rules* of CoLoSL, $\textsc{Triples} \in \mathcal{P}(\textsc{Ast} \times \textsc{Op} \times \textsc{Ast})$, are defined as follows, where $|P|_\Gamma \triangleq \{w \mid \Gamma, w \models P\}$ and $\models$ denotes the satisfiability relation in Def. 104:

$$\frac{}{\vdash \{P\}\ \texttt{skip}\ \{P\}}\ \textsc{Skip} \qquad \frac{\forall \Gamma.\ (|P|_\Gamma, \texttt{C}^{\texttt{A}}, |Q|_\Gamma) \in \textsc{Ax}_{\textsc{a}}}{\vdash \{P\}\ \texttt{C}^{\texttt{A}}\ \{Q\}}\ \textsc{Atom}$$

$$\frac{\vdash \{P\}\ \texttt{C}_1\ \{R\} \quad \vdash \{R\}\ \texttt{C}_2\ \{Q\}}{\vdash \{P\}\ \texttt{C}_1\,;\texttt{C}_2\ \{Q\}}\ \textsc{Seq}$$

$$\frac{\vdash \{P_1\}\ \texttt{C}_1\ \{Q_1\} \quad \vdash \{P_2\}\ \texttt{C}_2\ \{Q_2\}}{\vdash \{P_1 * P_2\}\ \texttt{C}_1 \,||\, \texttt{C}_2\ \{Q_1 * Q_2\}}\ \textsc{Par} \qquad \frac{\vdash \{P\}\ \texttt{C}\ \{Q\}}{\vdash \{P * R\}\ \texttt{C}\ \{Q * R\}}\ \textsc{Frame}$$

$$\frac{P \Rrightarrow P' \quad \vdash \{P'\}\ \texttt{C}\ \{Q'\} \quad Q' \Rrightarrow Q}{\vdash \{P\}\ \texttt{C}\ \{Q\}}\ \textsc{Cons}$$

$$\frac{\vdash \{P\}\ \texttt{C}_1\ \{Q\} \quad \vdash \{P\}\ \texttt{C}_2\ \{Q\}}{\vdash \{P\}\ \texttt{C}_1 + \texttt{C}_2\ \{Q\}}\ \textsc{Choice} \qquad \frac{\vdash \{P\}\ \texttt{C}\ \{P\}}{\vdash \{P\}\ \texttt{C}^*\ \{P\}}\ \textsc{Rec}$$

Most proof rules are standard from disjoint concurrent separation logic [39]. In the CONS rule, the $\Rrightarrow$ denotes the repartitioning of the state as described in Def. 116.

9.7. Operational Semantics and Soundness

Operational semantics We define the operational semantics of CoLoSL in terms of a set of *program* (low-level) states. Recall that the CoLoSL worlds are parametric in the monoid of logical states $(\textsc{LState}, \bullet_{\textsc{l}}, \textsc{Unit}_{\textsc{l}})$. As such, we also require that the choice of program states be provided as a parameter to CoLoSL. Since CoLoSL is an instance of the Views framework [14], its operational semantics is as defined in [14] instantiated with the set of program states and the *semantics of atomic operations*. The semantics of atomic operations are defined in terms of the *interpretation of sequential and primitive operations*. As CoLoSL may be instantiated with any set of primitive operations PRIM, the interpretation of these primitive operations must also be provided as a parameter to CoLoSL. We proceed with the formalisation of the ingredients necessary for defining the operational semantics of CoLoSL atomic operations.

CoLoSL Parameter

Parameter 27 (Program states)**.** Assume a set of program states $\sigma \in \textsc{PState}$.

CoLoSL Parameter

Parameter 28 (Primitive interpretation)**.** Given the set of primitive operations $\textsc{Prim}$ (Par. 26) and the set of program states $\textsc{PState}$ (Par. 27), assume a *primitive interpretation function*, $[\![.]\!]_{\mathrm{P}}(.) : \textsc{Prim} \to \textsc{PState} \to \mathcal{P}(\textsc{PState})$, associating each primitive operation in $\textsc{Prim}$ with a non-deterministic state transformer.
The interpretation function $[\![.]\!]_{\mathrm{P}}(.)$ is lifted to sets of program states as follows, for all $S \in \mathcal{P}(\textsc{PState})$:

$$[\![\mathtt{C}^{\mathrm{P}}]\!]_{\mathrm{P}}(S) \triangleq \bigcup_{\sigma \in S} [\![\mathtt{C}^{\mathrm{P}}]\!]_{\mathrm{P}}(\sigma)$$

Definition 125 (Sequential interpretation)**.** Given the set of sequential operations $\textsc{Seq}$ (Par. 119), the set of program states $\textsc{PState}$ (Par. 27) and the primitive interpretation function $[\![.]\!]_{\mathrm{P}}(.)$ (Par. 28), the *sequential interpretation function*, $[\![.]\!]_{\mathrm{S}}(.) : \textsc{Seq} \to \textsc{PState} \to \mathcal{P}(\textsc{PState})$, is defined as follows:

$$\begin{aligned}
[\![\mathtt{C}^{\mathrm{P}}]\!]_{\mathrm{S}}(\sigma) &\triangleq [\![\mathtt{C}^{\mathrm{P}}]\!]_{\mathrm{P}}(\sigma) \\
[\![\mathtt{skip}]\!]_{\mathrm{S}}(\sigma) &\triangleq \{\sigma\} \\
[\![\mathtt{C}^{\mathrm{S}}_1 ; \mathtt{C}^{\mathrm{S}}_2]\!]_{\mathrm{S}}(\sigma) &\triangleq \left\{\sigma' \,\middle|\, \exists \sigma''.\ \sigma'' \in [\![\mathtt{C}^{\mathrm{S}}_1]\!]_{\mathrm{S}}(\sigma) \wedge \sigma' \in [\![\mathtt{C}^{\mathrm{S}}_2]\!]_{\mathrm{S}}(\sigma'')\right\} \\
[\![\mathtt{C}^{\mathrm{S}}_1 + \mathtt{C}^{\mathrm{S}}_2]\!]_{\mathrm{S}}(\sigma) &\triangleq [\![\mathtt{C}^{\mathrm{S}}_1]\!]_{\mathrm{S}}(\sigma) \cup [\![\mathtt{C}^{\mathrm{S}}_2]\!]_{\mathrm{S}}(\sigma) \\
[\![(\mathtt{C}^{\mathrm{S}})^*]\!]_{\mathrm{S}}(\sigma) &\triangleq [\![\mathtt{skip} + \mathtt{C}^{\mathrm{S}}; (\mathtt{C}^{\mathrm{S}})^*]\!]_{\mathrm{S}}(\sigma)
\end{aligned}$$

The interpretation function $[\![.]\!]_{\mathrm{S}}(.)$ is lifted to sets of program states as follows, for all $S \in \mathcal{P}(\textsc{PState})$:

$$[\![\mathtt{C}^{\mathrm{S}}]\!]_{\mathrm{S}}(S) \triangleq \bigcup_{\sigma \in S} [\![\mathtt{C}^{\mathrm{S}}]\!]_{\mathrm{S}}(\sigma)$$

Definition 126 (Atomic interpretation)**.** Given the set of atomic operations Atom (Par. 121), the set of program states PState (Par. 27) and the sequential interpretation function $[\![.]\!]_{\text{S}}(.)$ (Par. 125), the *atomic interpretation function*, $[\![.]\!]_{\text{A}}(.) : \textsc{Atom} \to \textsc{PState} \to \mathcal{P}(\textsc{PState})$, is defined as follows:

$$[\![\texttt{<C}^{\text{s}}\texttt{>}]\!]_{\text{A}}(\sigma) \triangleq [\![\texttt{C}^{\text{s}}]\!]_{\text{S}}(\sigma)$$

The atomic interpretation function $[\![.]\!]_{\text{A}}(.)$ is lifted to sets of concrete states ad follows, for all $S \in \mathcal{P}(\textsc{PState})$:

$$[\![\texttt{C}^{\text{s}}]\!]_{\text{A}}(S) \triangleq \bigcup_{\sigma \in S} [\![\texttt{C}^{\text{s}}]\!]_{\text{A}}(\sigma)$$

Soundness In order to establish the soundness of CoLoSL program logic, we relate its proof judgements to its operational semantics. To this end, we relate the high-level CoLoSL states, namely worlds, to low-level program states by means of a *reification function.* The reification of worlds is defined in terms of relating (reifying) logical states in LState (Par. 22) to program states in PState (Par. 27). As CoLoSL is parametric in the choice of both logical and program states, the *reification of logical states* is also a parameter in CoLoSL.

Since CoLoSL is an instance of the Views framework [14], its soundness follows immediately from the soundness of Views, provided that the atomic axioms are sound with respect to their operational semantics. Recall that the axioms of atomic operations are defined in terms of the axioms of primitive operations in Prim (Par. 28) which are parametrised in CoLoSL. As such, to ensure the soundness of atomic operations, we require that the primitive axioms be sound with respect to their operational semantics.

CoLoSL Parameter

Parameter 29 (Logical state reification)**.** Given the set of logical states LState (Par. 22) and the set of program states PState (Par. 27), assume a *reification function for logical states*, $\lfloor . \rfloor_{\text{L}} : \textsc{LState} \to \mathcal{P}(\textsc{PState})$, relating logical states to sets of program states.
The logical state reification function $\lfloor . \rfloor_{\text{L}}$ is lifted to sets of logical

states as follows, for all $L \in \mathcal{P}(\textsc{LState})$:

$$\lfloor L \rfloor_{\text{L}} \triangleq \bigcup_{h \in L} \lfloor h \rfloor_{\text{L}}$$

CoLoSL Parameter

Parameter 30 (Primitive soundness)**.** Given the partial commutative monoid of logical states $(\textsc{LState}, \bullet_{\text{L}}, \textsc{Unit}_{\text{L}})$ in Par. 22, the set of program states $\textsc{PState}$ (Par. 27), the reification function $\lfloor . \rfloor_{\text{L}}$ (Par. 29), the set of primitive operations $\textsc{Prim}$ (Par. 26), the primitive axioms $\textsc{Ax}_{\text{P}}$ (Par. 30) and the primitive interpretation function $[\![.]\!]_{\text{P}}(.)$ (Par. 28), assume that for each primitive command $\mathtt{C}^{\text{P}} \in \textsc{Prim}$ the following *soundness property* holds:

$$\forall (L_1, \mathtt{C}^{\text{P}}, L_2) \in \textsc{Ax}_{\text{P}}. \forall h \in \textsc{LState}.\ [\![\mathtt{C}^{\text{P}}]\!]_{\text{P}} \left(\lfloor L_1 \bullet_{\text{L}} \{h\} \rfloor_{\text{L}} \right) \subseteq \lfloor L_2 \bullet_{\text{L}} \{h\} \rfloor_{\text{L}}$$

where the logical state composition $\bullet_{\text{L}}$ is lifted to sets of logical states as follows, for all $L, L' \in \mathcal{P}(\textsc{LState})$:

$$L \bullet_{\text{L}} L' \triangleq \left\{ h \bullet_{\text{L}} h' \,\middle|\, h \in L \wedge h' \in L' \right\}$$

Definition 127 (Reification)**.** Given the set of worlds $\textsc{World}$ (Def. 102), the set of program states $\textsc{PState}$ (Par. 27) and the logical state reification function $\lfloor . \rfloor_{\text{L}}$ (Par. 29), the *world reification function*, $\lfloor . \rfloor_W : \textsc{World} \to \mathcal{P}(\textsc{PState})$, is defined as follows, for all $(l, g, \mathcal{I}) \in \textsc{World}$:

$$\lfloor (l, g, \mathcal{I}) \rfloor_W \triangleq \lfloor (l \circ g)_{\text{L}} \rfloor_{\text{L}}$$

where $\circ$ denotes the composition of instrumented states and $(.)_{\text{L}}$ denotes the logical state component of an instrumented state (Def. 91).

Theorem 4 (Atomic soundness)**.** *For all* $\mathtt{C}^{\text{A}} \in \textsc{Atom}$ *(Def. 121),* $(W_1, \mathtt{C}^{\text{A}}, W_2) \in \textsc{Ax}_{\text{A}}$ *(Def. 122) and* $w \in \textsc{World}$ *(Def. 102):*

$$[\![\mathtt{C}^{\text{A}}]\!]_{\text{A}} \left(\lfloor W_1 \bullet \{w\} \rfloor_W \right) \subseteq \lfloor W_2 \bullet \mathcal{R}(w) \rfloor_W$$

where $\bullet$ denotes the composition of worlds (Def. 102), the $\lfloor . \rfloor_W$ denotes the world reification function (Def. 127), the $\mathcal{R}$ denotes the rely relation (Def. 112) with $\mathcal{R}(w) \triangleq \{w' \mid (w, w') \in \mathcal{R}\}$, where the world composition $\bullet$ is lifted to sets of worlds as follows, for all $W, W' \in \mathcal{P}(\textsc{World})$:

$$W \bullet W' \triangleq \left\{ w \bullet w' \,\middle|\, w \in W \land w' \in W' \right\}$$

Proof. Let $\texttt{C}^{\text{A}} = \texttt{<C}^{\text{S}}\texttt{>}$ for an arbitrary $\texttt{C}^{\text{S}} \in \textsc{Seq}$. Pick arbitrary $w = (l, g, \mathcal{I}) \in \textsc{World}$ and $W_1, W_2 \in \mathcal{P}(\textsc{World})$ such that $(W_1, \texttt{<C}^{\text{S}}\texttt{>}, W_2) \in \textsc{Ax}_{\text{A}}$. From the definition of $\textsc{Ax}_{\text{A}}$ (Def. 122) we then know there exist $L_1, L_2 \in \mathcal{P}(\textsc{LState})$ such that:

$$(L_1, \texttt{C}^{\text{S}}, L_2) \in \textsc{Ax}_{\text{S}} \land W_1 \Rrightarrow^{\{L_1\}\{L_2\}} W_2 \tag{9.33}$$

We are then required to show:

$$[\![\texttt{<C}^{\text{S}}\texttt{>}]\!]_{\text{A}} \left(\lfloor W_1 \bullet \{w\} \rfloor_W \right) \subseteq \lfloor W_2 \bullet \mathcal{R}(w) \rfloor_W$$

We first demonstrate that sequential operations are sound in that they preserve all frames. That is,

$$\begin{aligned} &\forall \texttt{C}^{\text{S}} \in \textsc{Seq}.\, \forall (L_1, \texttt{C}^{\text{S}}, L_2) \in \textsc{Ax}_{\text{S}}.\, \forall h \in \textsc{LState}. \\ &\qquad [\![\texttt{C}^{\text{S}}]\!]_{\text{S}} \left(\lfloor L_1 \bullet_{\text{L}} \{h\} \rfloor_{\text{L}} \right) \subseteq \lfloor L_2 \bullet_{\text{L}} \{h\} \rfloor_{\text{L}} \end{aligned} \tag{9.34}$$

This is proved in Lemma 44 of §C.

We then demonstrate that when a global state (and its associated action model) is updated by a thread via its guarantee relation, this change does not come as a surprise to the other threads in the environment in that it is captured by their rely relations. That is,

$$\begin{aligned} &\forall w_1, w_2 = (l_2, g_2, \mathcal{I}_2), w, w' = (l', g', \mathcal{I}') \in \textsc{World}. \\ &\qquad w_1 \bullet w_2 = w \land (l', g', \mathcal{I}') \in \mathcal{G}(w_1) \implies (l_2, g', \mathcal{I}') \in \mathcal{R}(w_2) \end{aligned} \tag{9.35}$$

This is proved in Lemma 47 of §C.

It then suffices to show that for an arbitrary $w_1 = (l_1, g_1, \mathcal{I}_1) \in W_1$ such that $w_1 \bullet w$ is defined, there exist $w_2 \in W_2$ and $w' \in \mathcal{R}(w)$ such that:

$$[\![\texttt{<C}^{\text{S}}\texttt{>}]\!]_{\text{A}} \left(\lfloor w_1 \bullet w \rfloor_W \right) = \lfloor w_2 \bullet w' \rfloor_W \tag{9.36}$$

From the definitions of $\llbracket . \rrbracket_{\mathrm{A}}(.)$ and $\lfloor . \rfloor_W$, and the properties of $\bullet$ and $\bullet_{\mathrm{L}}$ we know $g_1{=}g$, $\mathcal{I}_1{=}\mathcal{I}$ and:

$$\begin{aligned} \llbracket \texttt{<C}^{\mathrm{S}}\texttt{>} \rrbracket_{\mathrm{A}}(\lfloor w_1 \bullet w \rfloor_W) &= \llbracket \texttt{C}^{\mathrm{S}} \rrbracket_{\mathrm{S}}(\lfloor w_1 \bullet w \rfloor_W) \\ &= \llbracket \texttt{C}^{\mathrm{S}} \rrbracket_{\mathrm{S}}(\lfloor (l_1 \circ g_1)_{\mathrm{L}} \bullet_{\mathrm{L}} l_{\mathrm{L}} \rfloor_{\mathrm{L}}) \end{aligned} \tag{9.37}$$

On the other hand, from (9.33) and the definition of $\Rrightarrow$ we know there exist $h_1 \in L_1$ and $h' \in$ LState such that:

$$h_1 \bullet_{\mathrm{L}} h' = (l_1 \circ g_1)_{\mathrm{L}} \wedge \tag{9.38}$$

$$\begin{aligned} &\forall h_2 \in L_2.\, \exists w_2 = (l_2, g_2, \mathcal{I}_2) \in W_2. \\ &\quad h_2 \bullet_{\mathrm{L}} h' = (l_2 \circ g_2)_{\mathrm{L}} \wedge (w_1, w_2) \in \mathcal{G} \end{aligned} \tag{9.39}$$

Consequently from (9.37) and (9.38) we have:

$$\llbracket \texttt{<C}^{\mathrm{S}}\texttt{>} \rrbracket_{\mathrm{A}}(\lfloor w_1 \bullet w \rfloor_W) = \llbracket \texttt{C}^{\mathrm{S}} \rrbracket_{\mathrm{S}}\left(\left\lfloor h_1 \bullet_{\mathrm{L}} h' \bullet_{\mathrm{L}} l_{\mathrm{L}} \right\rfloor_{\mathrm{L}}\right) \tag{9.40}$$

From (9.33) and 9.34 we can rewrite (9.40) as:

$$\llbracket \texttt{<C}^{\mathrm{S}}\texttt{>} \rrbracket_{\mathrm{A}}(\lfloor w_1 \bullet w \rfloor_W) \subseteq \left\lfloor L_2 \bullet_{\mathrm{L}} \{h' \bullet_{\mathrm{L}} l_{\mathrm{L}}\} \right\rfloor_{\mathrm{L}}$$

That is, there exits $h_2 \in L_2$ such that:

$$\llbracket \texttt{<C}^{\mathrm{S}}\texttt{>} \rrbracket_{\mathrm{A}}(\lfloor w_1 \bullet w \rfloor_W) = \left\lfloor h_2 \bullet_{\mathrm{L}} h' \bullet_{\mathrm{L}} l_{\mathrm{L}} \right\rfloor_{\mathrm{L}} \tag{9.41}$$

From (9.39) we know there exists $w_2 \in$ World such that

$$w_2 = (l_2, g_2, \mathcal{I}_2) \in W_2 \wedge h_2 \circ h' = (l_2 \circ g_2)_{\mathrm{L}} \tag{9.42}$$

$$(w_1, w_2) \in \mathcal{G} \tag{9.43}$$

From the definition of $\lfloor . \rfloor_W$ and the properties of $\bullet_{\mathrm{L}}$ and $\bullet$ we can thus rewrite (9.41) as:

$$\llbracket \texttt{<C}^{\mathrm{S}}\texttt{>} \rrbracket_{\mathrm{A}}(\lfloor w_1 \bullet w \rfloor_W) = \lfloor (l_2 \circ w_{\mathtt{L}}, g_2, \mathcal{I}_2) \rfloor_W \tag{9.44}$$

From (9.35) and (9.43) we know there exists $w' \in$ World such that:

$$w' = (l, g_2, \mathcal{I}_2) \wedge w' \in \mathcal{R}(w) \tag{9.45}$$

Consequently, from (9.42), (9.44) and (9.45) we know there exist $w_2 \in W_2$ and $w' \in \mathcal{R}(w)$ such that:

$$[\![\texttt{<C}^{\mathrm{s}}\texttt{>}]\!]_{\mathrm{A}} \left(\lfloor w_1 \bullet w \rfloor_W \right) = \lfloor w_2 \bullet w' \rfloor_W$$

as required. □

Concluding remarks This concludes the formal development of CoLoSL. We have introduced CoLoSL, a new program logic for reasoning locally about the shared state. We focus on subjective views, which expand and contract to provide a flexible treatment of both the shared state and its interference. However, CoLoSL is still young and lacks many features of its various cousins such as abstract predicates [15, 54, 53, 38, 10, 36], higher-order reasoning [54, 53, 36, 35, 37] and abstract atomicity [10, 36]. These ideas suggest interesting directions and require further investigation. Here, our aim was to simply introduce subjective views as a fundamental new way of underpinning compositional reasoning. Extending CoLoSL with abstract predicates allows for building layers of abstraction and building black-box library specifications. This can be done in a similar fashion as in program logics such as [15, 38, 53, 36]. In order to extend CoLoSL with higher-order reasoning capabilities, we can employ the existing techniques in the literature [54, 53, 36, 35, 37]. Lastly, extending CoLoSL with abstractly atomic triples can be done in a similar fashion to the works in [10, 36]. An interesting challenge in each of these directions is preserving the general notion of interference composition and framing.

In the following chapter, we use CoLoSL to verify several challenging graph-manipulating algorithms. To do this, we devise a general proof pattern that divides the correctness proof into two parts: the functional correctness proof carried out on abstract mathematical structures, independently from the in-memory (heap) representation of the underlying data structures; and the memory safety proof connecting the abstract mathematical representations of structures to their counterpart in-memory representations, thus establishing the absence of memory leaks and invalid pointer dereferences. Combining the two parts of the proof, one can then establish the full correctness of an algorithm.

10. CoLoSL Examples

The verification of fine-grained concurrent algorithms is nontrivial. There has been much recent progress verifying such algorithms modularly using variants of concurrent separation logic [38, 48, 55, 53, 19, 15]. One area of particular difficulty has been verifying such algorithms that manipulate graphs. This is only to be expected: even in a semi-formal "algorithmic" sense, the correctness arguments of concurrent graph algorithms can be dauntingly subtle [12].

To verify such algorithms, we must not only understand these algorithmic arguments, but must also determine a precise way to express them in a suitable formal system. Even sequential graph algorithms are challenging to verify due to the overlapping nature of the graph structures, preventing e.g. easy use of the frame rule of separation logic [31]. Concurrent graph algorithms pose a number of additional challenges, such as reasoning how the actions of each thread advance the overall goal despite the possible interference from other threads. Unsurprisingly, verifications of such algorithms are rare in the literature.

We verify the functional correctness of three challenging concurrent fine-grained graph algorithms. We study a structure-preserving copy, a speculatively-parallel version of Dijkstra's shortest-path algorithm, and a spanning tree algorithm. We have found common "proof patterns" for tackling these algorithms, principally reasoning about the functional correctness of the algorithm on abstract *mathematical* graphs γ, defined as sets of vertices and edges. We use such abstractions to state and prove key invariants. We then track the progress of each thread using a notion of *tokens* to record each thread's portion of the computation. Informally, if the token of thread t is on vertex v, then t is responsible for some work on/around v. Our tokens are sufficiently general to handle sophisticated parallelism. (e.g. dynamic thread creation/destruction).

We then reason about the memory safety of the algorithm by connecting our reasoning on mathematical graphs to *spatial* graphs (sets of memory cells in the heap) by defining spatial predicates that implement mathematical structures in the heap e.g. $\mathsf{graph}(\gamma) \triangleq \ldots$. We define our spatial predicates in such a way that simplifies many of the proof obligations (e.g. when parallel computations join).

This pattern of doing the algorithmic reasoning on abstract states is similar to the style of reasoning used in logics such as CaReSL [55] and iCAP [53]. CaReSL introduced the idea of reasoning on abstract states. Later, iCAP extended the program logic of CAP [15] to reason about higher-order code and adopted CaReSL's abstract states. Just as with these logics, we carry out our reasoning on abstract states, which enables simpler proofs and lessens the burden of side conditions such as establishing stability. With these logics, this abstract style of reasoning has been "baked in" to the semantic models. Here, we demonstrate that this baking is unnecessary by using a logic (CoLoSL [48]) without such built-in support. We do not use any of the unique features of CoLoSL. As such, we believe that our proofs and style of abstract reasoning port to other program logics without difficulty.

There has been much work on reasoning about graph algorithms using separation logic. For *sequential* graph algorithms, Bornat et al. presented preliminary work on dags in [4], Yang studied the Schorr-Waite graph algorithm [61], Reynolds conjectured how to reason about dags [50], and Hobor and Villard showed how to reason about dags and graphs [31]. We make critical use of some of Hobor and Villard's graph-related verification infrastructure.

Many *concurrent* program logics have been proposed in recent years; both iCAP and CaReSL encourage the kind of abstract reasoning we employ in our verifications. However, published examples in these logics focus heavily on verifying concurrent data structures, whereas we focus on verifying concurrent graph algorithms. Moreover, the semantic models for both of these logics incorporate significant machinery to enable this kind of abstract reasoning, whereas we are able to use it without built-in support.

There has hardly been any work on *concurrent* graph algorithms. Raad et al. [48] and Sergey et al. [51] have verified a concurrent spanning tree algorithm, `span`, one of our examples. In [48], Raad et al. introduced

CoLoSL and gave a shaped-based proof of spanning tree to demonstrate CoLoSL reasoning. A full functional correctness proof in CoLoSL was available at the time, although not using the proof pattern presented here. Later in [51], Sergey et al. gave a full functional correctness proof in Coq, but only that single example. As we demonstrate shortly, we provide a *local* specification whereby the footprint of `span(x)` is captured accurately and is limited to the *sub-graph* at `x`. This is in contrast to the work of Sergey et al. in [51] where the `span(x)` footprint encompasses the *entire* (global) graph, and thus the interference observed by each thread comprises the interference on all nodes, regardless of whether they are included in the `span(x)` footprint. This is because the framing mechanisms of CoLoSL afforded by the FORGET and SHIFT principles allows us to carve out the accurate footprint. More concretely, we can limit the `span(x)` footprint to the sub-graph at `x` (those nodes reachable from `x`), and similarly limit the observed interference to the interference incurred by those nodes in the sub-graph at `x` only.

We believe we are the first to verify `copy_dag`, which is known to be difficult, and `parrellel_dijkstra`, which we believe is the first verification of an algorithm that uses speculative parallel decomposition [26].

We proceed with an overview of our proof pattern. We then use our proof pattern to verify the concurrent algorithms of `copy_dag` (§10.2), `parallel_dijkstra` (§10.3) and `span` (§10.1).

Proof Pattern: Combining Mathematical and Spatial Reasoning Our graph verifications follow a common pattern which we outline as follows. First, we select an appropriate abstract model for *mathematical graphs*, which is typically sets of vertices and edges together with labels.

Second, we choose a *token* model. We use tokens to identify each thread uniquely and to track the contribution of each thread to the global computation. For instance, for an algorithm with only two threads this might be as simple as the set $\{\mathsf{red}, \mathsf{blue}\}$, identifying each thread as a distinct colour.

Third, we define *mathematical actions* to capture the operations performed by threads. These actions model both *concrete* updates to the graph (e.g. removing an edge), as well as *ghost* updates used solely for reasoning (e.g. adding or removing tokens to track the computation progress).

Fourth, we define *mathematical assertions* to describe program invariants and pre-/postconditions. These assertions are on mathematical graphs and involve abstract concepts (e.g. reachability along a path). As a key proof obligation, we must prove that our mathematical assertions are *stable* with respect to our mathematical actions, i.e. they remain true under the actions of other threads in the environment.

Fifth, we define *spatial predicates* (e.g. $\mathsf{graph}(\gamma)$) that describe how mathematical graphs are implemented in the heap. For instance, a graph may be implemented as a set of heap-linked nodes or as an adjacency matrix. We then combine these spatial predicates with our mathematical actions to define *spatial actions.* Intuitively, if a mathematical action transforms γ to γ', then the corresponding spatial action transforms $\mathsf{graph}(\gamma)$ to $\mathsf{graph}(\gamma')$.

10.1. Parallel Spanning Tree Computation

Mathematical graphs The `span` program in Fig. 10.1 operates on a directed binary graph (henceforth simply graph) where each node has at most two successors, referred to as its left and right children. We assume that the graph is connected and that all nodes in the graph are reachable from its top node on which the initial call to `span` is executed.

The `span` program concurrently computes an in-place spanning tree of the graph (i.e. a tree that covers all nodes of the graph from a given root) as follows: each time a new node is encountered, two new threads are spawned each pruning the edges of its left and right children recursively. A mark bit is associated with each node to keep track of the nodes that have already been visited. Each thread returns whether it was responsible for marking the node it was called on or whether another thread had already marked it. In the latter case, when the thread joins its parent thread removes the link from its own root node to the corresponding child. Intuitively, it is allowed to do so because the child has already been reached via some other path in the graph since it was marked by another thread.

Our language is C with a few cosmetic differences. Line 1 gives the data type of heap-represented graphs. The statements between angle brackets <.> (e.g. line 4) denote atomic instructions that cannot be interrupted by

```
 1. struct node {int m, node *l, *r}; bool b;
 2. b = span(struct node *x){
 3.   if(!x){ return 1; }
 4.   bool res = <&CAS(x->m, 0, 1)>;
 5.   if(res){
 6.     bool b1=span(x->l) ||bool b1=span(x->r)
 7.     if(!b1) { x->l = null; }
 8.     if(!b2) { x->r = null; }
 9.   }
10.   return res;
11. }
```

Figure 10.1.: The concurrent `span` program

other threads. We write `C1 || C2` (e.g. line 6) for the parallel computation of `C1` and `C2`. This corresponds to the standard fork-join parallelism.

Although the code is short, its correctness argument is rather subtle as we need to reason simultaneously about both deep unspecified sharing inside the graph as well as the parallel behaviour. This is not surprising since the unspecified sharing makes verifying even the sequential version of similar algorithms non-trivial [31]. However, the non-deterministic behaviour of parallel computation makes even *specifying* the behaviour of `span` challenging. Observe that each node x of the graph may be in one of the following three stages:

1. x is not visited by any thread (not marked yet), and thus its mark field is 0.

2. x has already been marked by a thread π, and the mark field of x has been accordingly updated to 1. However, the edges of x have not been updated accordingly. That is, the thread marking x has not yet finished executing line 8.

3. x has been marked and its edges of have been updated accordingly.

Note that in stage 2 when x has already been visited by a thread π, if another thread π' visits x, it simply returns even though x and its children may not have been fully processed yet. How do we then specify the postcondition of thread π' since we cannot promise that the subgraph at x is fully spanned when it returns? Intuitively, thread π' can safely return because another thread (π) has marked x and has made a *promise* to visit its children and ensure that they are also spanned (by which time the said children may have been marked by other threads, incurring further promises).

In order to track the contribution of each thread and record the overall spanning progress, we must identify each thread uniquely. To this end, we appeal to a *token* (identification) mechanism that can i) distinguish one token (thread) from another; ii) identify two distinct sub-tokens given any token, to reflect the new threads spawned at recursive call points; and iii) model a parent-child relationship to discern the spawner thread from its sub-threads. We model our tokens as a variation of the tree share algebra in [17] as described below.

Trees as tokens A tree token (hereafter simply a token), $\pi \in \Pi$, is defined by the grammar below as a binary tree with boolean leaves ($\circ$, $\bullet$), exactly one $\bullet$ leaf, and unlabelled internal nodes.

$$\Pi \ni \pi ::= \bullet \mid \widehat{\circ \quad \pi} \mid \widehat{\pi \quad \circ}$$

We refer to the thread associated with π as thread π. To model the parent-child relation between thread π and its two sub-threads (left and right), we define a mechanism for creating two distinct sibling tokens $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ defined below. Intuitively, $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ denote replacing the $\bullet$ leaf of π with $\widehat{\circ \quad \bullet}$ and $\widehat{\bullet \quad \circ}$, respectively.

$$\begin{array}{lll}
\bullet.\mathsf{l} = \widehat{\circ \quad \bullet} & \left(\widehat{\circ \quad \pi}\right).\mathsf{l} = \widehat{\circ \quad \pi.\mathsf{l}} & \left(\widehat{\pi \quad \circ}\right).\mathsf{l} = \widehat{\pi.\mathsf{l} \quad \circ} \\
\bullet.\mathsf{r} = \widehat{\bullet \quad \circ} & \left(\widehat{\circ \quad \pi}\right).\mathsf{r} = \widehat{\circ \quad \pi.\mathsf{r}} & \left(\widehat{\pi \quad \circ}\right).\mathsf{r} = \widehat{\pi.\mathsf{r} \quad \circ}
\end{array}$$

We model the ancestor-descendant relation between threads by the $\sqsubset$ ordering defined below where $+$ denotes the transitive closure of the relation:

$$\sqsubset \triangleq \{(\pi.\mathsf{l}, \pi), (\pi.\mathsf{r}, \pi) \mid \pi \in \Pi\}^+$$

We write $\pi \sqsubseteq \pi'$ for $\pi = \pi' \vee \pi \sqsubset \pi'$, and write $\pi \not\sqsubset \pi'$ (respectively $\pi \not\sqsubseteq \pi'$) for $\neg(\pi \sqsubset \pi')$ (respectively $\neg(\pi \sqsubseteq \pi')$). Observe that $\bullet$ is the maximal token, i.e. $\forall \pi \in \Pi.\ \pi \sqsubseteq \bullet$. As such, the top-level thread is associated with the $\bullet$ token, since all other threads are its sub-threads and are subsequently spawned by it or its descendants. In what follows we write $\overline{\pi}$ to denote the token set comprising the descendants of π, i.e. $\overline{\pi} \triangleq \{\pi' \mid \pi' \sqsubseteq \pi\}$.

As discussed earlier, we carry out most of our reasoning abstractly by appealing to *mathematical objects*. To this end, we define *mathematical graphs* as an abstraction of the graph structure in `span`.

Mathematical graphs A mathematical graph, $\gamma \in \textsc{Graph}$, is a triple in $(\mathrm{V} \times \mathrm{E} \times \mathrm{L})$ where V is the vertex set; $\mathrm{E} : \mathrm{V} \to \mathrm{V}_0 \times \mathrm{V}_0$, is the edge function with $\mathrm{V}_0 \triangleq \mathrm{V} \uplus \{0\}$, where 0 denotes the absence of an edge (e.g. a null pointer); and $\mathrm{L} : \mathrm{V} \to \mathrm{D}$, is the vertex labelling function with the label set D. We define our labels as $\mathrm{D} \triangleq \{0\} \uplus \Pi$. That is, for each node the label function records whether it is yet to be visited (0), or it has been already visited by a thread π and subsequently marked (π).

Given a graph $\gamma = (V, E, L)$, we write γ^{V} for V, γ^{E} for E, and γ^{L} for L. We write $\gamma^{\mathsf{l}}(x) = l$ and $\gamma^{\mathsf{r}}(x) = r$ when $\gamma^{\mathrm{E}}(x) = (l, r)$, and write $\gamma^{\mathsf{m}}(x)$ for $\gamma^{\mathrm{L}}(x)$. We write $\gamma(x)$ for $(\gamma^{\mathsf{m}}(x), \gamma^{\mathsf{l}}(x), \gamma^{\mathsf{r}}(x))$ when $x \in V$. Given a function f (e.g. E, L), we write $f[x \mapsto v]$ for updating $f(x)$ to v, and write $f \uplus [x \mapsto v]$ for extending f with x and value v.

We define the *path* relation, $\overset{\gamma}{\rightsquigarrow}$, and its reflexive transitive closure, $\overset{\gamma}{\rightsquigarrow}^*$, as follows:

$$x \overset{\gamma}{\rightsquigarrow} y \triangleq \gamma^{\mathsf{l}}(x) = y \vee \gamma^{\mathsf{r}}(x) = y \qquad x \overset{\gamma}{\rightsquigarrow}^* y \triangleq x = y \vee (\exists z.\ x \overset{\gamma}{\rightsquigarrow} z \wedge z \overset{\gamma}{\rightsquigarrow}^* y)$$

Similarly, we define the *marked path* relation, $\overset{\gamma}{\rightsquigarrow}_{\pi}$, as:

$$x \overset{\gamma}{\rightsquigarrow}_{\pi} y \triangleq \gamma^{\mathsf{m}}(y) = \pi \wedge x \overset{\gamma}{\rightsquigarrow}\!\mid_{\pi} y$$
$$x \overset{\gamma}{\rightsquigarrow}\!\mid_{\pi} y \triangleq x = y \vee \exists z, \pi'.\ x \overset{\gamma}{\rightsquigarrow}_{\pi'} z \wedge \big((\pi = \pi'.\mathsf{l} \wedge \gamma^{\mathsf{l}}(z) = y) \vee (\pi = \pi'.\mathsf{r} \wedge \gamma^{\mathsf{r}}(z) = y)\big)$$

That is, the $\overset{\gamma}{\rightsquigarrow}_{\pi}$ states that there is a path from x to y, that every node along this path (each ancestor of y) is marked by a super-thread of π, and that the sink node y itself is marked by thread π. For instance, when there is a marked path from node x to y via z with x marked by thread π, z marked by its left sub-thread $\pi.\mathsf{l}$ and y marked by the right sub-thread of $\pi.\mathsf{l}$ (i.e. $\pi.\mathsf{l}.\mathsf{r}$), then $x \overset{\gamma}{\rightsquigarrow}_{\pi.\mathsf{l}.\mathsf{r}} y$, $x \overset{\gamma}{\rightsquigarrow}_{\pi.\mathsf{l}} z$, and $x \overset{\gamma}{\rightsquigarrow}_{\pi} x$ all hold. The $x \overset{\gamma}{\rightsquigarrow}\!\mid_{\pi} y$ relation is similar and excludes the stipulation about the sink node y being marked. That is, the $x \overset{\gamma}{\rightsquigarrow}\!\mid_{\pi} y$ states that there is a path from x to y and that every node along this path is marked by a super-thread of π, while not specifying if and how y is marked. It is possible to inline the definition of $x \overset{\gamma}{\rightsquigarrow}\!\mid_{\pi} y$ in that of $x \overset{\gamma}{\rightsquigarrow}_{\pi} y$. However, as we demonstrate shortly, we appeal to the $x \overset{\gamma}{\rightsquigarrow}\!\mid_{\pi} y$ relation to specify the precondition of `span`. This is because upon calling `span` on y, we know of the existence of a marked path to y ending at π. However, node y itself may be either unmarked (not visited by a thread yet) or it may already be marked by another thread other than π.

Actions As discussed earlier, to model the interactions of each thread π with the shared data structure, we define mathematical *actions* as relations on mathematical objects. We thus define two families of actions, each of which indexed by a token $\pi \in \Pi$. The first set, A^1_π, describes the atomic operation of line 1 in the algorithm: the state of a node is changed from unmarked to marked by thread π.

$$A^1_\pi \triangleq \left\{ \big((V,E,L),(V,E,L')\big) \;\middle|\; \begin{array}{l} \exists x, p, \pi'.\ L(x){=}0 \land \land\ L'{=}L[x \mapsto \pi] \\ \land \big[(x{=}\top \land \pi{=}\bullet) \\ \quad \lor (E(p){=}(x,-) \land L(p){=}\pi' \land \pi{=}\pi'.\mathsf{l}) \\ \quad \lor (E(p){=}(-,x) \land L(p){=}\pi' \land \pi{=}\pi'.\mathsf{r})\big] \end{array} \right\}$$

The next two set of actions respectively describe the atomic operations of lines 7 and 8.

$$A^2_\pi \triangleq \left\{ \big((V,E,L),(V,E',L)\big) \;\middle|\; \begin{array}{l} \exists x, l, r, \pi'. \\ L(x){=}\pi \land E(x){=}(l,r) \land L(l){=}\pi' \land \pi'{\neq}\pi.\mathsf{l} \\ \land E'{=}E[x \mapsto (0,r)] \end{array} \right\}$$

$$A^3_\pi \triangleq \left\{ \big((V,E,L),(V,E',L)\big) \;\middle|\; \begin{array}{l} \exists x, l, r, \pi'. \\ L(x){=}\pi \land E(x){=}(l,r) \land L(r){=}\pi' \land \pi'{\neq}\pi.\mathsf{r} \\ \land E'{=}E[x \mapsto (l,0)] \end{array} \right\}$$

We write A_π for actions of thread π: $A_\pi \triangleq A^1_\pi \cup A^2_\pi \cup A^3_\pi$. We can now specify the behaviour of `span` mathematically.

Mathematical specification Let γ_0 denote the original graph (over which the initial call to `span` is executed) with its top node denoted by $\textsc{t}$ from which all other nodes are reachable. Throughout the execution of `span`, the current graph γ satisfies the invariant $\mathsf{Inv}(\gamma, \textsc{t}, \gamma_0)$ defined below:

$$\begin{aligned}
\mathsf{Inv}(\gamma, \textsc{t}, \gamma_0) &\triangleq \gamma \leq_{\textsc{t}} \gamma_0 \wedge \forall \textsc{x}, \pi.\ \gamma^{\mathsf{m}}(\textsc{x}) = \pi \Rightarrow \textsc{t} \overset{\gamma}{\rightsquigarrow}_{\pi} \textsc{x} \\
\gamma \leq_{\textsc{t}} \gamma_0 &\triangleq \gamma^{\textsc{v}} {=} \gamma_0^{\textsc{v}} \wedge \gamma^{\textsc{e}} \subseteq \gamma_0^{\textsc{e}} \wedge \forall \textsc{x}.\ \textsc{t} \overset{\gamma_0}{\rightsquigarrow}{}^{*}\ \textsc{x} \Rightarrow \textsc{t} \overset{\gamma}{\rightsquigarrow}{}^{*}\ \textsc{x} \\
\gamma^{\textsc{e}} \subseteq \gamma_0^{\textsc{e}} &\triangleq \forall \textsc{x}.\ \big(\gamma^{\mathsf{l}}(\textsc{x}){=}0 \vee \gamma^{\mathsf{l}}(\textsc{x}){=}\gamma_0^{\mathsf{l}}(\textsc{x})\big) \wedge \big(\gamma^{\mathsf{r}}(\textsc{x}){=}0 \vee \gamma^{\mathsf{r}}(\textsc{x}){=}\gamma_0^{\mathsf{r}}(\textsc{x})\big)
\end{aligned}$$

Informally, the invariant asserts that the current graph γ is a partial spanning graph of the original graph γ_0 (first conjunct) and that for each node $\textsc{x}$ in γ, if $\textsc{x}$ is labelled π, then there is a marked path π from the top node $\textsc{t}$ to $\textsc{x}$ (second conjunct). The graph γ is a partial spanning graph of the original graph γ_0 if i) γ has the same vertices as γ_0; ii) the edges of γ are those of γ_0 unless they have been pruned (set to 0); and iii) every node reachable from the top node $\textsc{t}$ in the original graph γ_0 is also reachable from $\textsc{t}$ in γ.

Observe that `span` does not eliminate nodes and thus $\textsc{t}$ remains as the top node in γ. In the remainder of this section, we write γ_0 for the original graph (prior to the initial call to `span`) and write $\textsc{t}$ for its top node which remains unchanged as γ_0 evolves.

When calling `span` on a sub-graph at $\textsc{x}$, the mathematical precondition of `span`, $\mathsf{P}^{\pi}(\gamma, \textsc{t}, \textsc{x})$, can be specified as follows, where π denotes the thread identifier executing `span`, γ is the current graph and $\textsc{t}$ denotes the top node:

$$\mathsf{P}^{\pi}(\gamma, \textsc{t}, \textsc{x}) \triangleq (\textsc{x}{=}0 \vee \textsc{t} \overset{\gamma}{\rightsquigarrow}\!|_{\pi}\ \textsc{x}) \wedge \forall \textsc{y}, \pi'.\ \gamma^{\mathsf{m}}(\textsc{y}){=}\pi' \Rightarrow \pi' \not\sqsubseteq \pi$$

The first conjunct of the precondition asserts that either the sub-graph at $\textsc{x}$ is empty ($\textsc{x}{=}0$), or there is a marked path from the top node $\textsc{t}$ to $\textsc{x}$. Moreover, all ancestors of $\textsc{x}$ along this path (excluding $\textsc{x}$ itself) are marked by a super-thread of π (excluding π). Recall that each token uniquely identifies a thread and thus the descendants of π correspond to the sub-threads subsequently spawned by π. As such, the second conjunct of the precondition asserts that prior to the execution of `span` by thread π, none of the descendants of π (including π itself) have yet marked any nodes.

The $\mathsf{Q}_{\mathsf{s}}^{\pi}(\gamma,\mathrm{X})$ and $\mathsf{Q}_{\mathsf{f}}^{\pi}(\gamma,\mathrm{X})$ defined below describe the mathematical postconditions of `span` when called by thread π on a subgraph at X. The $\mathsf{Q}_{\mathsf{s}}^{\pi}(\gamma,\mathrm{X})$ describes the case when thread π $\underline{s}$uccessfully marks the node at X and thus $\gamma^{\mathsf{m}}(\mathrm{X})=\pi$ (if $\mathrm{X} \neq 0$). The $\mathsf{Q}_{\mathsf{f}}^{\pi}(\gamma,\mathrm{X})$ captures the case when thread π $\underline{f}$ails to mark the node at X as it has already been marked by another thread other than π or its descendants. Note that in this case the thread responsible for marking X cannot be a descendant of X as the thread simply returns (line 3) and does not spawn new sub-threads.

$$\mathsf{Q}_{\mathsf{s}}^{\pi}(\gamma,\mathrm{X}) \triangleq (\mathrm{X}=0 \vee \gamma^{\mathsf{m}}(\mathrm{X})=\pi) \qquad \mathsf{Q}_{\mathsf{f}}^{\pi}(\gamma,\mathrm{X}) \triangleq \exists \pi'.\, \pi' \not\sqsubseteq \pi \wedge \gamma^{\mathsf{m}}(\mathrm{X})=\pi'$$

Recall that as a key proof obligation we must prove that our mathematical assertions are stable with respect to our mathematical actions. This is captured by Lemma 13 below. Part (10.1) states that the invariant Inv is stable with respect to the actions of all threads (A_π for any tokens π). Parts (10.3) and (10.4) state that the postconditions of thread π' ($\mathsf{Q}_{\mathsf{s}}^{\pi'}$ and $\mathsf{Q}_{\mathsf{f}}^{\pi'}$) are stable with respect to the actions of all threads. Part (10.2) states that the precondition of thread π' ($\mathsf{P}^{\pi'}$) is stable with respect to the actions of all threads but those of its descendants ($\pi \not\in \overline{\pi'}$). Observe that despite the additional stipulation $\pi \not\in \overline{\pi'}$, the actions of π are irrelevant and do not affect the stability of $\mathsf{P}^{\pi'}$. More concretely, the precondition $\mathsf{P}^{\pi'}$ only holds at the beginning of the program *before* new descendants are spawned (line 6). As such, at these program points $\mathsf{P}^{\pi'}$ is trivially stable with respect to the actions of its (non-existing) descendants.

Lemma 13 (`span` stability)**.** *For all mathematical graphs γ and γ', nodes t and x, and tokens π and π',*

$$\mathsf{Inv}(\gamma, \mathrm{T}, \gamma_0) \wedge \gamma\, A_\pi\, \gamma' \Rightarrow \mathsf{Inv}(\gamma', \mathrm{T}, \gamma_0) \tag{10.1}$$

$$\mathsf{P}^{\pi'}(\gamma, \mathrm{T}, \mathrm{X}) \wedge \pi \not\in \overline{\pi'} \wedge \gamma\, A_\pi\, \gamma' \Rightarrow \mathsf{P}^{\pi'}(\gamma', \mathrm{T}, \mathrm{X}) \tag{10.2}$$

$$\mathsf{Q}_{\mathsf{s}}^{\pi'}(\gamma,\mathrm{X}) \wedge \gamma\, A_\pi\, \gamma' \Rightarrow \mathsf{Q}_{\mathsf{s}}^{\pi'}(\gamma',\mathrm{X}) \tag{10.3}$$

$$\mathsf{Q}_{\mathsf{f}}^{\pi'}(\gamma,\mathrm{X}) \wedge \gamma\, A_\pi\, \gamma' \Rightarrow \mathsf{Q}_{\mathsf{f}}^{\pi'}(\gamma',\mathrm{X}) \tag{10.4}$$

Proof. Follows from the definitions of A_π, Inv, P, Q_{s}, and Q_{f}.

We are almost in a position to verify `span`. As discussed earlier, in order to verify `span` we integrate our mathematical correctness argument with a machine-level memory safety argument by linking our abstract mathematical objects to concrete structures in the heap. We proceed with the spatial representation of our mathematical graphs in the heap.

Spatial graphs We represent a mathematical sub-graph γ at X in the heap through the G predicate below as a location (G) in the ghost heap tracking the current abstract state of the graph (γ), together with a collection of $*$-separated nodes reachable from X (the g predicate). Observe that this way of tracking the abstract state of the graph in the ghost heap eliminates the need for baking in the abstract state into the model. That is, rather than incorporating the abstract state into the model as in [53, 55], we encode it as an additional resource in the ghost heap. We use $\Rrightarrow$ for ghost heap cells to differentiate them from concrete heap cells indicated by $\mapsto$. As before, we write e.g. $\mathrm{X} \dot{=} 0$ for $\mathrm{X}{=}0 \wedge \mathsf{emp}$.

A node X in γ may be either unmarked ($\mathsf{U}(\gamma, \mathrm{X})$) or marked by a thread π ($\mathsf{M}(\gamma,\mathrm{X},\pi)$). A node is represented as three adjacent cells in the heap together with an additional cell in the ghost heap. The cells in the heap track the mark bit of the node (0 when unmarked, 1 when marked), and the left (L) and right (R) children, respectively. The ghost location is used to track the thread identifier responsible for marking the node (0 when unmarked, π when marked by thread π).

$$\mathsf{G}(\gamma,\mathrm{X}) \triangleq \mathrm{G} \Rrightarrow \gamma * \mathsf{g}(\gamma,\mathrm{X}) \qquad\qquad \mathsf{g}(\gamma,\mathrm{X}) \triangleq \mathrm{X}\dot{=}0 \vee \underset{\mathrm{X} \overset{\gamma}{\leadsto}^{*} \mathrm{Y}}{\circledast} \mathsf{node}(\gamma, \mathrm{Y})$$

$$\begin{aligned}
\mathsf{node}(\gamma, \mathrm{X}) &\triangleq \exists \pi \in \Pi.\ \mathsf{M}(\gamma,\mathrm{X},\pi) \vee \mathsf{U}(\gamma, \mathrm{X}) \\
\mathsf{M}(\gamma,\mathrm{X},\pi) &\triangleq \gamma^{\mathsf{m}}(\mathrm{X}) \dot{=} \pi * \exists \mathrm{L}, \mathrm{R}.\ \gamma^{\mathsf{l}}(\mathrm{X}) \dot{=} \mathrm{L} * \gamma^{\mathsf{r}}(\mathrm{X}) \dot{=} \mathrm{R} * \mathrm{X} \mapsto 1, \mathrm{L}, \mathrm{R} * \mathrm{X} \Rrightarrow \pi \\
\mathsf{U}(\gamma, \mathrm{X}) &\triangleq \gamma^{\mathsf{m}}(\mathrm{X}) \dot{=} 0 * \exists \mathrm{L}, \mathrm{R}.\ \gamma^{\mathsf{l}}(\mathrm{X}) \dot{=} \mathrm{L} * \gamma^{\mathsf{r}}(\mathrm{X}) \dot{=} \mathrm{R} * \mathrm{X} \mapsto 0, \mathrm{L}, \mathrm{R} * \mathrm{X} \Rrightarrow 0
\end{aligned}$$

Observe that each recursive call to `span(x)` computes a spanning graph of the nodes accessible in the *sub-graph* at `x` rather than the entire graph and the nodes accessible from its top node T. In order to give a local specification of `span` and focus on the relevant sub-graph at each recursive call point, we use the following lemma (due to Wang et al in [58]) to fold and unfold the graph predicate g and thus zoom in on the appropriate sub-graph.

Lemma 14 (Graph unfolding due to [58]). *For all mathematical graphs γ*

and nodes X*:*

$$\mathsf{g}(\gamma,\mathrm{X}) \iff \mathrm{X}\doteq 0 \vee \big(\mathsf{node}(\gamma,\mathrm{X}) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathrm{X})) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathrm{X}))\big)$$

Spatial specification We can now specify the spatial precondition of span, $\mathsf{Pre}(\gamma_0,\mathrm{T},\mathrm{X},\pi)$, as a CoLoSL assertion defined below where γ_0 denotes the original graph, T denotes the top node of γ_0, X denotes the top node of the sub-graph over which span is called and π denotes the thread identifier executing span. Recall that the spatial actions in CoLoSL are indexed by *capabilities*. That is, a CoLoSL action may be performed by a thread only when it holds the necessary capabilities. Since CoLoSL is parametric in its primitive capability model, to verify span we instantiate our primitive capabilities as sets of tokens in $\mathcal{P}(\Pi)$. That is, the partial commutative monoid of primitive capabilities (Par. 23) is given by $(\mathcal{P}(\Pi), \uplus, \emptyset)$.

The precondition Pre states that the current thread π holds the capabilities associated with itself and all its descendants ($[\overline{\pi}]$). Thread π will subsequently pass on the descendant capabilities when spawning new sub-threads and reclaim them as the sub-threads return and join. The Pre further asserts that the abstract state of the graph currently corresponds to γ. That is, since the graph is concurrently manipulated by several threads, to ensure the stability of the shared state assertion to the actions of the environment, Pre states that the original graph γ_0 may have evolved to another graph γ (captured by the existential quantifier). The Pre also states that the shared state contains the spatial resources of the sub-graph ($\mathsf{G}(\gamma,\mathrm{X})$), that γ satisfies the invariant Inv, and that γ satisfies the mathematical precondition P^{π}. The spatial actions on the sub-graph at X are declared in I where mathematical actions are simply lifted to spatial ones indexed by the associated capability. That is, if thread π holds the $[\overline{\pi}]$ capability and the actions of π (A_{π}) admit the update of the mathematical object γ to γ', then thread π may update the spatial sub-graph $\mathsf{G}(\gamma,\mathrm{X})$ to $\mathsf{G}(\gamma',\mathrm{X})$.

The spatial postcondition Post is analogous to Pre and describes the two possible outcomes of executing span returned in B. In the first case the execution of span is successful ($\mathrm{B}\doteq 1$) as captured by the mathematical postcondition $\mathsf{Q}^{\pi}_{\mathsf{s}}(\gamma,\mathrm{X})$. In the second case the execution of span fails ($\mathrm{B}\doteq 0$) as denoted by the mathematical postcondition $\mathsf{Q}^{\pi}_{\mathsf{f}}(\gamma,\mathrm{X})$. Moreover, in the failure case the resulting sub-graph is described by the spatial predicate G, whereas in the success case the resulting sub-graph is indeed a *tree* (i.e. there is at most one path from the top node of the sub-graph to each descendant node) as described by the spatial predicate T. In particular, compare the recursive definition of t (in T) to the recursive unfolding of g in G (Lemma 14). While the sub-structures in a graph are combined by the overlapping conjunction connective $\uplus$ to account for duplicate nodes (accessible via multiple paths), they are separated by the $*$ connective in a tree as multiple paths are prohibited.

$$
\begin{aligned}
\mathsf{Pre}(\gamma_0,\mathrm{T},\mathrm{X},\pi) &\triangleq [\overline{\pi}] * \boxed{\exists\gamma.\ \mathsf{G}(\gamma,\mathrm{X}) * (\mathsf{Inv}(\gamma,\mathrm{T},\gamma_0) \land \mathsf{P}^{\pi}(\gamma,\mathrm{T},\mathrm{X}))}_{I(\mathrm{X})} \\
\mathsf{Post}(\gamma_0,\mathrm{T},\mathrm{X},\pi,\mathrm{B}) &\triangleq [\overline{\pi}] * \begin{pmatrix} \mathrm{B}\doteq 1 * \boxed{\exists\gamma.\mathsf{T}(\gamma,\mathrm{X},\pi) * (\mathsf{Inv}(\gamma,\mathrm{T},\gamma_0) \land \mathsf{Q}^{\pi}_{\mathsf{s}}(\gamma,\mathrm{X}))}_{I(\mathrm{X})} \\ \lor\ \mathrm{B}\doteq 0 * \boxed{\exists\gamma.\mathsf{G}(\gamma,\mathrm{X}) * (\mathsf{Inv}(\gamma,\mathrm{T},\gamma_0) \land \mathsf{Q}^{\pi}_{\mathsf{f}}(\gamma,\mathrm{X}))}_{I(\mathrm{X})} \end{pmatrix} \\
\mathsf{T}(\gamma,\mathrm{X},\pi) &\triangleq \mathrm{G} \Rrightarrow \gamma * \mathsf{t}(\gamma,\mathrm{X},\pi) \\
\mathsf{t}(\gamma,\mathrm{X},\pi) &\triangleq \mathrm{X}\doteq 0 \lor (\mathsf{M}(\gamma,\mathrm{X},\pi) * \mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathrm{X}),\pi.\mathsf{l}) * \mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathrm{X}),\pi.\mathsf{r})) \\
I(\mathrm{X}) &\triangleq \left\{[\overline{\pi}] : \mathsf{G}(\gamma,\mathrm{X}) \land \gamma\, A_{\pi}\, \gamma' \leadsto \mathsf{G}(\gamma',\mathrm{X})\right\}
\end{aligned}
$$

Recall that the top-level thread is associated with the maximal token $\bullet$ and executes `span` on the sub-graph at the root node T, i.e. the entire graph. Observe that when the $\bullet$ thread terminates its execution of `span`, only the first disjunct (the success case) of its spatial postcondition $\mathsf{Post}(\gamma_0,\mathrm{T},\mathrm{T},\bullet,\mathrm{B})$ applies. More precisely, the second disjunct (failure case) stipulates that T be marked by a thread that is not a descendant of $\bullet$ (via the mathematical postcondition (Q_{f}). This however cannot be the case since $\bullet$ is the maximal token (i.e. $\forall\pi.\ \pi \sqsubseteq \bullet$). As such, the spatial tree predicate T in the first disjunct together with the invariant Inv assert that the final graph is a spanning tree of the original graph γ_0. More concretely, the invariant states that γ is a spanning graph of γ_0 (via $\gamma \leq_{\mathrm{T}} \gamma_0$) while T asserts that γ is a tree.

As with the work of Sergey et al. in [51], it is possible to define a more sophisticated invariant and mathematical pre- and postconditions that single-handedly ensure that the resulting graph is a spanning tree of the original graph. That is, it it is possible to delegate the entire correctness argument to the mathematical level without appealing to the spatial representation of the graphs. Indeed, this is the reasoning style we employ in the subsequent examples of §10.2 and §10.3. Here, we choose to keep the invariant and the pre- and postconditions simple and instead utilise the resulting in-memory shape of the graph, namely its "tree-ness", to our advantage. After all, one of the motivations for the $*$ connective of separation logic was the ability to describe disjoint structures in the heap. By handling part of the correctness argument at memory level, we keep the mathematical argument simpler.

Verifying `span` We give a proof sketch of `span` in Fig. 10.2. As before, at each proof point we have highlighted the effect of the preceding command, where applicable.

On line 3 we check if `x` is 0. If so the program returns and the postcondition, $\mathsf{Post}(\gamma_0,\textsc{t},\mathtt{x},\pi,1)$, follows trivially from the definition of the precondition $\mathsf{Pre}(\gamma_0,\textsc{t},\mathtt{x},\pi)$. If $\mathtt{x} \neq 0$, then the atomic block of line 4 is executed. We first check if `x` is marked; if so then the desired postcondition $\mathsf{Post}(\gamma_0,\textsc{t},\mathtt{x},\pi,0)$ follows from the precondition $\mathsf{Pre}(\gamma_0,\textsc{t},\mathtt{x},\pi)$. More concretely, the postcondition $\mathsf{Q}^{\pi}_{\mathsf{f}}(\gamma,\mathtt{x})$ in the failure case stipulates that `x` be marked by a thread other than π and its descendants. This simply follows from the precondition since `x` is marked and $\mathsf{P}^{\pi}(\gamma,\textsc{t},\mathtt{x})$ states that neither π nor its descendants have marked any nodes.

On the other hand, if `x` is not marked, we set `res` to true and perform A^1_{π} by setting the mark bit of `x` to 1 and the ghost bit of `x` to π. In this (success) case, we immediately obtain the preconditions $\mathsf{Pre}(\gamma_0,\textsc{t},\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l})$ and $\mathsf{Pre}(\gamma_0,\textsc{t},\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})$, as shown in the derivation below. In the first implication we move the existential quantifier $\exists\gamma$ to the beginning of the assertion and split the capabilities $[\overline{\pi}]$. Recall from the definition of $\overline{\pi}$ that $\overline{\pi}=\{\pi\} \uplus \overline{\pi.\mathsf{l}} \uplus \overline{\pi.\mathsf{r}}$. As such from the semantics of capability assertions we have: $[\overline{\pi}] \Leftrightarrow \pi * [\overline{\pi.\mathsf{l}}] * [\overline{\pi.\mathsf{r}}]$. We next apply the Copy principle and duplicate the shared resources. We then weaken each subjective assertion by applying the Forget principle and dropping the irrelevant resources and pure assertions. Finally, we apply the Shift principle to weaken the interference assertions in the last two subjective views. It is straightforward to check that $I(\mathtt{x}) \sqsubseteq^{P_1} I(\gamma^{\mathsf{l}}(\mathtt{x}))$ and $I(\mathtt{x}) \sqsubseteq^{P_2} I(\gamma^{\mathsf{r}}(\mathtt{x}))$ where $P_1 \triangleq \textsc{g} \Rightarrow \gamma * \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}))$ and $P_2 \triangleq \textsc{g} \Rightarrow \gamma * \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}))$.

$$[\overline{\pi}] * \boxed{\begin{array}{l}\exists\gamma.\ \textsc{g} \Rightarrow \gamma * \big(\mathsf{M}(\gamma,\mathtt{x},\pi) \circledast \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x})) \circledast \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}))\big) \\ *(\gamma^{\mathsf{m}}(\mathtt{x}) \doteq \pi \wedge \mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \mathsf{P}^{\pi.\mathsf{l}}(\gamma,\textsc{t},\gamma^{\mathsf{l}}(\mathtt{x})) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\gamma,\textsc{t},\gamma^{\mathsf{r}}(\mathtt{x})))\end{array}}_{I(\mathtt{x})}$$

$$\begin{array}{l}
\Longrightarrow \begin{array}{l}\exists\gamma.\,[\pi]*[\overline{\pi.\mathsf{l}}]\\ *\,[\overline{\pi.\mathsf{r}}]\end{array} * \boxed{\begin{array}{l}\textsc{g} \Rightarrow \gamma * \big(\mathsf{M}(\gamma,\mathtt{x},\pi) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x})) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}))\big)\\ *(\gamma^{\mathsf{m}}(\mathtt{x}) \doteq \pi \wedge \mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \mathsf{P}^{\pi.\mathsf{l}}(\gamma,\textsc{t},\gamma^{\mathsf{l}}(\mathtt{x})) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\gamma,\textsc{t},\gamma^{\mathsf{r}}(\mathtt{x})))\end{array}}_{I(\mathtt{x})}\\
\overset{\textsc{Copy}}{\Longrightarrow} \exists\gamma.\ [\pi] * \boxed{\begin{array}{l}\textsc{g} \Rightarrow \gamma * \big(\mathsf{M}(\gamma,\mathtt{x},\pi) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x})) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}))\big)\\ *(\gamma^{\mathsf{m}}(\mathtt{x}) \doteq \pi \wedge \mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \mathsf{P}^{\pi.\mathsf{l}}(\gamma,\textsc{t},\gamma^{\mathsf{l}}(\mathtt{x})) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\gamma,\textsc{t},\gamma^{\mathsf{r}}(\mathtt{x})))\end{array}}_{I(\mathtt{x})}\\
\qquad * [\overline{\pi.\mathsf{l}}] * \boxed{\begin{array}{l}\textsc{g} \Rightarrow \gamma * \big(\mathsf{M}(\gamma,\mathtt{x},\pi) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x})) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}))\big)\\ *(\gamma^{\mathsf{m}}(\mathtt{x}) \doteq \pi \wedge \mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \mathsf{P}^{\pi.\mathsf{l}}(\gamma,\textsc{t},\gamma^{\mathsf{l}}(\mathtt{x})) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\gamma,\textsc{t},\gamma^{\mathsf{r}}(\mathtt{x})))\end{array}}_{I(\mathtt{x})}\\
\qquad * [\overline{\pi.\mathsf{r}}] * \boxed{\begin{array}{l}\textsc{g} \Rightarrow \gamma * \big(\mathsf{M}(\gamma,\mathtt{x},\pi) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x})) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}))\big)\\ *(\gamma^{\mathsf{m}}(\mathtt{x}) \doteq \pi \wedge \mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \mathsf{P}^{\pi.\mathsf{l}}(\gamma,\textsc{t},\gamma^{\mathsf{l}}(\mathtt{x})) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\gamma,\textsc{t},\gamma^{\mathsf{r}}(\mathtt{x})))\end{array}}_{I(\mathtt{x})}\\
\overset{\textsc{Forget}}{\Longrightarrow} \exists\gamma.\ [\pi] * \boxed{\textsc{g} \Rightarrow \gamma * \mathsf{M}(\gamma,\mathtt{x},\pi) * (\mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \gamma^{\mathsf{m}}(\mathtt{x}) \doteq \pi)}_{I(\mathtt{x})}\\
\qquad * [\overline{\pi.\mathsf{l}}] * \boxed{\textsc{g} \Rightarrow \gamma * \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x})) * (\mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \mathsf{P}^{\pi.\mathsf{l}}(\gamma,\textsc{t},\gamma^{\mathsf{l}}(\mathtt{x})))}_{I(\mathtt{x})}\\
\qquad * [\overline{\pi.\mathsf{r}}] * \boxed{\textsc{g} \Rightarrow \gamma * \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x})) * (\mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\gamma,\textsc{t},\gamma^{\mathsf{r}}(\mathtt{x})))}_{I(\mathtt{x})}\\
\overset{\textsc{Shift}}{\Longrightarrow} \exists\gamma.\ [\pi] * \boxed{\textsc{g} \Rightarrow \gamma * \mathsf{M}(\gamma,\mathtt{x},\pi) * (\mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \gamma^{\mathsf{m}}(\mathtt{x}) \doteq \pi)}_{I(\mathtt{x})}\\
\qquad * [\overline{\pi.\mathsf{l}}] * \boxed{\textsc{g} \Rightarrow \gamma * \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x})) * (\mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \mathsf{P}^{\pi.\mathsf{l}}(\gamma,\textsc{t},\gamma^{\mathsf{l}}(\mathtt{x})))}_{I(\gamma^{\mathsf{l}}(\mathtt{x}))}\\
\qquad * [\overline{\pi.\mathsf{r}}] * \boxed{\textsc{g} \Rightarrow \gamma * \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x})) * (\mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\gamma,\textsc{t},\gamma^{\mathsf{r}}(\mathtt{x})))}_{I(\gamma^{\mathsf{r}}(\mathtt{x}))}\\
\Longleftrightarrow \exists\gamma.\ [\pi] * \boxed{\textsc{g} \Rightarrow \gamma * \mathsf{M}(\gamma,\mathtt{x},\pi) * (\mathsf{Inv}(\gamma,\textsc{t},\gamma_0) \wedge \gamma^{\mathsf{m}}(\mathtt{x}) \doteq \pi)}_{I(\mathtt{x})}\\
\qquad * \mathsf{Pre}(\gamma_0,\textsc{t},\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l}) * \mathsf{Pre}(\gamma_0,\textsc{t},\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})
\end{array}$$

On line 6 we call `span` on the left and right sub-graphs of `x`. We then use the Par rule (Def. 124) to distribute the resources between the sub-threads and collect them back when they join. In the failure case of each sub-thread (`b1`=0 or `b2`=0), we apply the Forget principle on the subjective assertions to drop the spatial resources of $\mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}))$ and $\mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}))$ and weaken them to emp. On lines 7 and 8 we apply the A^2_π and A^3_π actions to remove the respective edges in failure cases. We can then rewrite the subjective assertion emp as $\mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l})$ since $\gamma^{\mathsf{l}}(\mathtt{x})=0$ and thus $\mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l})$ holds trivially; *mutatis mutandis* for $\mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})$. Finally, we combine the subjective views using Merge, as shown in the following derivation. In the last implication, the $I(\mathtt{x}) \cup I(\gamma^{\mathsf{l}}(\mathtt{x})) \cup I(\gamma^{\mathsf{r}}(\mathtt{x}))$ is replaced by the equivalent interference assertion $I(\mathtt{x})$ (see the definition of $I(\mathtt{x})$). Moreover, observe that:

$$\begin{array}{l}\mathsf{M}(\gamma,\mathtt{x},\pi) \uplus \mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l}) \uplus \mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r}) \implies\\ \qquad\qquad \mathsf{M}(\gamma,\mathtt{x},\pi) * \mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l}) * \mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})\end{array}$$

```
1.  struct node {int m, node *l, *r}; bool b;
    $\{\mathsf{Pre}(\gamma_0,\mathrm{T},\mathtt{x},\pi)\}$
2.  b = span(struct node *x){ $\{\mathsf{Pre}(\gamma_0,\mathrm{T},\mathtt{x},\pi)\}$
3.    if(!x){ $\{\mathsf{Post}(\gamma_0,\mathrm{T},\mathtt{x},\pi,1) \wedge \mathtt{x}{=}0\}$ return 1; $\{\mathsf{Post}(\gamma_0,\mathrm{T},\mathtt{x},\pi,\mathtt{ret})\}$ }
    $\{\mathsf{Pre}(\gamma_0,\mathrm{T},\mathtt{x},\pi) \wedge \mathtt{x}{\neq}0\}$
4.    bool res = <CAS(x->m, 0, 1)>; // apply $A^1_\pi$ if possible
    $\left\{\begin{array}{l} \mathtt{res}{\doteq}0 * \mathsf{Post}(\gamma_0,\mathrm{T},\mathtt{x},\pi,\mathtt{res}) \vee \\ \mathtt{res}{\doteq}1 * [\overline{\pi}] * \boxed{\begin{array}{l}\exists\gamma.\ \mathrm{G} \Rightarrow \gamma * (\mathsf{M}(\gamma,\mathtt{x},\pi) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x})) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}))) \\ *(\mathsf{Inv}(\gamma,\mathrm{T},\gamma_0) \wedge \gamma^{\mathsf{m}}(\mathtt{x}){=}\pi \wedge \forall \mathrm{Y},\pi'.\ \gamma^{\mathsf{m}}(\mathrm{Y}){=}\pi' \Rightarrow \pi' \not\sqsubset \pi)\end{array}}_{I(\mathtt{x})} \end{array}\right\}$
5.    if(res){
    $\left\{\begin{array}{l} \mathtt{res}{\doteq}1 * [\pi] * \\ *[\overline{\pi.\mathsf{l}}] * [\overline{\pi.\mathsf{r}}] \end{array} \boxed{\begin{array}{l}\exists\gamma.\ \mathrm{G} \Rightarrow \gamma * (\mathsf{M}(\gamma,\mathtt{x},\pi) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x})) \uplus \mathsf{g}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}))) \\ *(\gamma^{\mathsf{m}}(\mathtt{x}){=}\pi \wedge \mathsf{P}^{\pi.\mathsf{l}}(\gamma,\mathrm{T},\gamma^{\mathsf{l}}(\mathtt{x})) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\gamma,\mathrm{T},\gamma^{\mathsf{r}}(\mathtt{x})))\end{array}}_{I(\mathtt{x})} \right\}$
    $\left\{\begin{array}{l} \mathtt{res}{\doteq}1 * [\pi] * \exists\gamma.\ \boxed{\mathrm{G} \Rightarrow \gamma * (\mathsf{Inv}(\gamma,\mathrm{T},\gamma_0) \wedge \gamma^{\mathsf{m}}(\mathtt{x}){=}\pi) * \mathsf{M}(\gamma,\mathtt{x},\pi)}_{I(\mathtt{x})} \\ * \mathsf{Pre}(\gamma_0,\mathrm{T},\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l}) * \mathsf{Pre}(\gamma_0,\mathrm{T},\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r}) \end{array}\right\}$
          $\{\mathsf{Pre}(\gamma_0,\mathrm{T},\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l})\}$    ||  $\{\mathsf{Pre}(\gamma_0,\mathrm{T},\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})\}$
6.       bool b1=span(x->l)           ||  bool b2=span(x->r)
         $\{\mathsf{Post}(\gamma_0,\mathrm{T},\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l},\mathtt{b1})\}$  ||  $\{\mathsf{Post}(\gamma_0,\mathrm{T},\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r},\mathtt{b2})\}$
    $\left\{\begin{array}{l} \mathtt{res}{\doteq}1 * [\pi] * \exists\gamma.\ \boxed{\mathrm{G} \Rightarrow \gamma * (\mathsf{Inv}(\gamma,\mathrm{T},\gamma_0) \wedge \gamma^{\mathsf{m}}(\mathtt{x}){=}\pi) * \mathsf{M}(\gamma,\mathtt{x},\pi)}_{I(\mathtt{x})} \\ * \mathsf{Post}(\gamma_0,\mathrm{T},\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l},\mathtt{b1}) * \mathsf{Post}(\gamma_0,\mathrm{T},\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r},\mathtt{b2}) \end{array}\right\}$
    $\left\{\begin{array}{l} \mathtt{res}{\doteq}1 * [\pi] * [\overline{\pi.\mathsf{l}}] * [\overline{\pi.\mathsf{r}}] * \exists\gamma.\ \boxed{\mathrm{G} \Rightarrow \gamma * (\mathsf{Inv}(\gamma,\mathrm{T},\gamma_0) \wedge \gamma^{\mathsf{m}}(\mathtt{x}){=}\pi) * \mathsf{M}(\gamma,\mathtt{x},\pi)}_{I(\mathtt{x})} \\ *\Big(\mathtt{b1}{\doteq}1 * \boxed{\mathrm{G} \Rightarrow \gamma * \mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l})}_{I(\gamma^{\mathsf{l}}(\mathtt{x}))} \\ \quad \vee \mathtt{b1}{\doteq}0 * \boxed{\mathrm{G} \Rightarrow \gamma * (\mathsf{Q}_{\mathsf{f}}^{\pi.\mathsf{l}}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x})) \wedge \mathsf{emp})}_{I(\gamma^{\mathsf{l}}(\mathtt{x}))}\Big) \\ *\Big(\mathtt{b2}{\doteq}1 * \boxed{\mathrm{G} \Rightarrow \gamma * \mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})}_{I(\gamma^{\mathsf{r}}(\mathtt{x}))} \\ \quad \vee \mathtt{b2}{\doteq}0 * \boxed{\mathrm{G} \Rightarrow \gamma * (\mathsf{Q}_{\mathsf{f}}^{\pi.\mathsf{r}}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x})) \wedge \mathsf{emp})}_{I(\gamma^{\mathsf{r}}(\mathtt{x}))}\Big) \end{array}\right\}$
7.       if(!b1) { x->l = null; } // perform $A^2_\pi$ if applicable
    $\left\{\begin{array}{l} \mathtt{res}{\doteq}1 * [\overline{\pi}] * \exists\gamma.\ \boxed{\mathrm{G} \Rightarrow \gamma * (\mathsf{Inv}(\gamma,\mathrm{T},\gamma_0) \wedge \gamma^{\mathsf{m}}(\mathtt{x}){=}\pi) * \mathsf{M}(\gamma,\mathtt{x},\pi)}_{I(\mathtt{x})} \\ * \boxed{\mathrm{G} \Rightarrow \gamma * \mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l})}_{I(\gamma^{\mathsf{l}}(\mathtt{x}))} * \Big(\mathtt{b2}{\doteq}1 * \boxed{\mathrm{G} \Rightarrow \gamma * \mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})}_{I(\gamma^{\mathsf{r}}(\mathtt{x}))} \\ \qquad \vee \mathtt{b2}{\doteq}0 * \boxed{\mathrm{G} \Rightarrow \gamma * (\mathsf{Q}_{\mathsf{f}}^{\pi.\mathsf{r}}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x})) \wedge \mathsf{emp})}_{I(\gamma^{\mathsf{r}}(\mathtt{x}))}\Big) \end{array}\right\}$
8.       if(!b2) { x->r = null; } // perform $A^3_\pi$ if applicable
    $\left\{\begin{array}{l} \mathtt{res}{\doteq}1 * [\overline{\pi}] * \exists\gamma.\ \boxed{\mathrm{G} \Rightarrow \gamma * (\mathsf{Inv}(\gamma,\mathrm{T},\gamma_0) \wedge \gamma^{\mathsf{m}}(\mathtt{x}){=}\pi) * \mathsf{M}(\gamma,\mathtt{x},\pi)}_{I(\mathtt{x})} \\ * \boxed{\mathrm{G} \Rightarrow \gamma * \mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l})}_{I(\gamma^{\mathsf{l}}(\mathtt{x}))} * \boxed{\mathrm{G} \Rightarrow \gamma * \mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})}_{I(\gamma^{\mathsf{r}}(\mathtt{x}))} \end{array}\right\}$
9.    } // apply the MERGE principle  $\{\mathsf{Post}(\gamma_0,\mathrm{T},\mathtt{x},\pi,\mathtt{res})\}$
10.   return res;
11. } $\{\mathsf{Post}(\gamma_0,\mathrm{T},\mathtt{x},\pi,\mathtt{ret})\}$
```

Figure 10.2.: Code and a proof sketch of `span`

In particular, note that $\mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l})$ describes a collection of nodes each of which marked by either $\pi.\mathsf{l}$ or one of its descendants (i.e. marked by a thread whose identifier is in $\overline{\pi.\mathsf{l}}$). Similarly for $\mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})$. As such, since the tokens in $\overline{\pi.\mathsf{l}}$ are disjoint from those in $\overline{\pi.\mathsf{r}}$ and $\{\pi\}$, the resources described by $\mathsf{M}(\gamma,\mathtt{x},\pi)$, $\mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l})$ and $\mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})$ are pairwise disjoint.

$$\begin{aligned}
& \boxed{\mathrm{G} \Rrightarrow \gamma * (\mathsf{Inv}(\gamma, \mathrm{T}, \gamma_0) \land \gamma^{\mathsf{m}}(\mathtt{x}){=}\pi) * \mathsf{M}(\gamma,\mathtt{x},\pi)}_{I(\mathtt{x})} \\
& * \boxed{\mathrm{G} \Rrightarrow \gamma * \mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l})}_{I(\gamma^{\mathsf{l}}(\mathtt{x}))} * \boxed{\mathrm{G} \Rrightarrow \gamma * \mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})}_{I(\gamma^{\mathsf{r}}(\mathtt{x}))} \\
\overset{\text{MERGE}}{\Longrightarrow} & \boxed{\begin{array}{l}\mathrm{G} \Rrightarrow \gamma * (\mathsf{Inv}(\gamma, \mathrm{T}, \gamma_0) \land \gamma^{\mathsf{m}}(\mathtt{x}){=}\pi) \\ \;*(\mathsf{M}(\gamma,\mathtt{x},\pi) \uplus\!\!\!* \; \mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l}) \uplus\!\!\!* \; \mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r}))\end{array}}_{I(\mathtt{x}) \cup I(\gamma^{\mathsf{l}}(\mathtt{x})) \cup I(\gamma^{\mathsf{r}}(\mathtt{x}))} \\
\Longrightarrow & \boxed{\mathrm{G} \Rrightarrow \gamma * (\mathsf{Inv}(\gamma,\mathrm{T},\gamma_0) \land \gamma^{\mathsf{m}}(\mathtt{x}){=}\pi) * \mathsf{M}(\gamma,\mathtt{x},\pi) * \mathsf{t}(\gamma,\gamma^{\mathsf{l}}(\mathtt{x}),\pi.\mathsf{l}) * \mathsf{t}(\gamma,\gamma^{\mathsf{r}}(\mathtt{x}),\pi.\mathsf{r})}_{I(\mathtt{x})} \\
\Longleftrightarrow & \boxed{\mathrm{G} \Rrightarrow \gamma * (\mathsf{Inv}(\gamma, \mathrm{T}, \gamma_0) \land \mathsf{Q}_{\mathsf{s}}^{\pi}(\gamma,\mathtt{x})) * \mathsf{t}(\gamma,\mathtt{x},\pi)}_{I(\mathtt{x})}
\end{aligned}$$

10.2. Copying Heap-represented Dags Concurrently

The `copy_dag(x)` program in Fig. 10.5 makes a deep structure-preserving copy of the dag (directed acyclic graph) rooted at `x` concurrently. To do this, each node `x` in the source dag records in its copy field (`x->c`) the location of its copy when it exists, or 0 otherwise.

A thread running `copy_dag(x)` first checks atomically (lines 5-7) if `x` has already been copied. If so, the address of the copy is returned. Otherwise, the thread allocates a new node `y` to serve as the copy of `x` and updates `x->c` accordingly; it then proceeds to copy the left and right subdags in parallel by spawning two new threads (line 9). At the beginning of the initial call, none of the nodes have been copied and all copy fields are 0; at the end of this call, all nodes are copied to a new dag whose root is returned by the algorithm. In the intermediate recursive calls, only parts of the dag rooted at the argument are copied. Note that the atomic block of lines 5-7 corresponds to a `CAS` (compare and set) operation. We have unwrapped the definition for better readability.

Observe that each node x of the source dag may be in one of the following three stages:

1. x is not visited by any thread (not copied yet), and thus its copy field is 0.

2. x has already been visited by a thread π, a copy node x' has been allocated, and the copy field of x has been accordingly updated to x'. However, the edges of x' have not been directed correctly. That is, the thread copying x has not yet finished executing line 10.

3. x has been copied and the edges of its copy have been updated accordingly.

Note that in stage 2 when x has already been visited by a thread π, if another thread π' visits x, it simply returns even though x and its children may not have been fully copied yet. Intuitively, thread π' can safely return because another thread (π) has copied x and has made a *promise* to visit its children and ensure that they are also copied (by which time the said children may have been copied by other threads, incurring further promises). More concretely, to reason about `copy_dag` we associate each node with a *promise set* identifying those threads that must visit it.

Consider the dags in Fig. 10.3 where a node x is depicted as i) a white circle when in stage 1, e.g. $(x,0)$ in 10.3a; ii) a grey ellipse when in stage 2, e.g. $\frac{x,x'}{\pi}$ in 10.3b where thread π has copied x to x'; and iii) a black circle when in stage 3, e.g. x,x' in 10.3g. Initially no node is copied and as such all copy fields are 0. Let us assume that the top thread (the thread running the very first call to `copy_dag`) is identified as π. That is, thread π has made a promise to visit the top node x and as such the promise set of x comprises π. This is depicted in the initial snapshot of the graph in Fig. 10.3a by the $\{\pi\}$ promise set next to x. Thread π proceeds with copying x to x', and transforming the dag to that of Fig. 10.3b. In doing so, thread π fulfils its promise to x and π is thus removed from the promise set of x. Recall that if another thread now visits x it simply returns, relinquishing the responsibility of copying the descendants of x. This is because the responsibility to copy the left and right subdags of x lies with the left and right sub-threads of π (spawned at line 9), respectively. As such, in transforming the dag from Fig. 10.3a to 10.3b, thread π extends the promise sets of l and r, where $\pi.\mathsf{l}$ (resp. $\pi.\mathsf{r}$) denotes the left (resp. right) sub-thread spawned by π at line 9. Subsequently, the $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ sub-threads copy l and r as illustrated in Fig. 10.3c, each incurring a promise to visit y via their sub-threads. That is, since both l and r have an edge to y, they race to copy the subdag

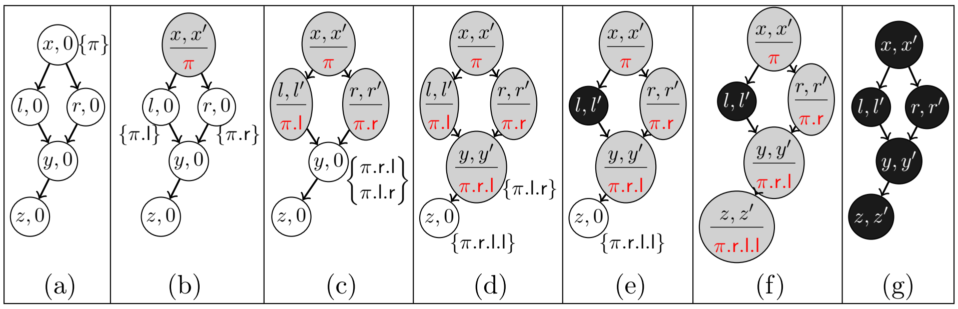


Figure 10.3.: An example trace of `copy_dag`

at y. In the trace detailed in Fig. 10.3, the $\pi.\mathsf{r}.\mathsf{l}$ sub-thread wins the race and transforms the dag to that of Fig. 10.3d by removing $\pi.\mathsf{r}.\mathsf{l}$ from the promise set of y, and incurring a promise at z. Since the $\pi.\mathsf{l}.\mathsf{r}$ sub-thread lost the race for copying y, it simply returns (line 3). That is, $\pi.\mathsf{l}.\mathsf{r}$ needs not proceed to copy y as it has already been copied. As such, the promise of $\pi.\mathsf{l}.\mathsf{r}$ to y is trivially fulfilled and the copying of l is finalised. This is captured in the transition from Fig. 10.3d to 10.3e where $\pi.\mathsf{l}.\mathsf{r}$ is removed from the promise set of y, and l is taken to stage 3. Thread $\pi.\mathsf{r}.\mathsf{l}.\mathsf{l}$ then proceeds to copy z, transforming the dag to that of Fig. 10.3f. Since z has no descendants, the copying of the subdag at z is now at an end; thread $\pi.\mathsf{r}.\mathsf{l}.\mathsf{l}$ thus returns, taking z to stage 3. In doing so, the copying of the entire dag is completed; sub-threads join and the effect of copying is propagated to the parent threads, taking the dag to that depicted in Fig. 10.3g.

Observe that as with `span` in §10.1, the `copy_dag` program spawns a new thread at each recursive call point in line 8. We thus take our tokens as elements of the tree share algebra, $\pi \in \Pi$, described in §10.1.

We associate the top-level thread with the $\bullet$ token (i.e. $\pi{=}\bullet$ in Figs. 10.3a-10.3g), since $\bullet$ is the maximal token and all other threads are its sub-threads and are subsequently spawned by it or its descendants. As before, we write $\overline{\pi}$ to denote the token set comprising the descendants of π, i.e. $\overline{\pi} \triangleq \{\pi' \mid \pi' \sqsubseteq \pi\}$.

Mathematical dags Similar to mathematical graphs in §10.1, a mathematical dag, $\delta \in \textsc{Dag}$, is a triple in $(\mathrm{V}, \mathrm{E}, \mathrm{L})$ where V is the vertex set; $\mathrm{E}\,{:}\,\mathrm{V} \to \mathrm{V}_0{\times}\mathrm{V}_0$, is the edge function with $\mathrm{V}_0 \triangleq \mathrm{V} \uplus \{0\}$, where 0 denotes the absence of an edge (e.g. a null pointer); and $L : \mathrm{V}{\to}\mathrm{D}$, is the vertex labelling function with the label set D defined shortly. As before, given a graph $\delta{=}(V, E, L)$, we write δ^{V}, δ^{E} and δ^{L}, for the first, second and third projections of δ, respectively. Moreover, we write $\delta^{\mathsf{l}}(x)$ and $\delta^{\mathsf{r}}(x)$ for the first and second projections of $E(x)$; and write $\delta(x)$ for $(\delta^{\mathrm{L}}(x), \delta^{\mathsf{l}}(x), \delta^{\mathsf{r}}(x))$ when $x \in V$. Given a function f (e.g. E, L), we write $f[x \mapsto v]$ for updating $f(x)$ to v, and write $f \uplus [x \mapsto v]$ for extending f with x and value v. Two dags are *congruent* if they have the same vertices and edges, i.e. $\delta_1 \cong \delta_2 \triangleq \delta_1^{\mathrm{V}}{=}\delta_2^{\mathrm{V}} \wedge \delta_1^{\mathrm{E}}{=}\delta_2^{\mathrm{E}}$. We define our mathematical objects as pairs of dags $(\delta, \delta') \in (\textsc{Dag} \times \textsc{Dag})$, where δ and δ' denote the source dag and its copy, respectively.

To capture the stages a node goes through, we define the node labels as $\mathrm{D} \triangleq \big(\mathrm{V}_0 \times (\Pi \uplus \{0\}) \times \mathcal{P}(\Pi)\big)$. The first component records the *copy* information (the address of the copy when in stage 2 or 3; 0 when in stage 1). This corresponds to the second components in the nodes of the dags in Fig. 10.3, e.g. 0 in $\textcircled{\scriptsize x,0}$. The second component tracks the node *stage* as described on page 339: 0 in stage 1 (white nodes in Fig. 10.3), some π in stage 2 (grey nodes in Fig. 10.3), and 0 in stage 3 (black nodes in Fig. 10.3). That is, when the node is being *processed* by thread π, this component reflects the thread's token. Note that this is a *ghost* component in that it is used purely for reasoning and does not appear in the physical memory.
The third (ghost) component denotes the *promise* set of the node and tracks the tokens of those threads that are yet to visit it. This corresponds to the sets adjacent to nodes in the dags of Fig. 10.3, e.g. $\{\pi.l\}$ in Fig. 10.3b. We write $\delta^{\mathsf{c}}(x)$, $\delta^{\mathsf{s}}(x)$ and $\delta^{\mathsf{p}}(x)$ for the first, second, and third projections of x's label, respectively. We define the *path* relation, $x \overset{\delta}{\rightsquigarrow} y$, and the *unprocessed path* relation, $x \overset{\delta}{\rightsquigarrow}_0 y$, as follows and write $\overset{\delta}{\rightsquigarrow}^*$ and $\overset{\delta}{\rightsquigarrow}_0^*$ for their reflexive transitive closure, respectively.

$$x \overset{\delta}{\rightsquigarrow} y \triangleq \delta^{\mathsf{l}}(x){=}y \vee \delta^{\mathsf{r}}(x){=}y \qquad\qquad x \overset{\delta}{\rightsquigarrow}_0 y \triangleq x \overset{\delta}{\rightsquigarrow} y \wedge \delta^{\mathsf{c}}(x) = 0 \wedge \delta^{\mathsf{c}}(y) = 0$$

The lifetime of a node x with label (c, s, P) can be described as follows. Initially, x is in stage 1 ($c{=}0$, $s{=}0$). When thread π visits x, it creates a copy node x' and takes x to stage 2 ($c{=}x'$, $s{=}\pi$). In doing so, it removes its token π from the promise set P, and adds $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ to the promise sets of its left and right children, respectively. Once π finishes executing

$$A^1_\pi \triangleq \left\{ \begin{array}{c|l} & \delta_1(x)=\big((0,0,P\uplus\{\pi\}),l,r\big) \\ & \land\, \delta_1{}^{\text{L}}(l)=(c_l,s_l,P_l) \land \delta_1{}^{\text{L}}(r)=(c_r,s_r,P_r) \\ \big((\delta_1,\delta_2), & \land\, \delta'_1 = (\delta^{\text{V}}_1,\delta^{\text{E}}_1,L'_1) \land \delta'_2 = (V'_2,E'_2,L'_2) \\ (\delta'_1,\delta'_2)\big) & \land\, L''_1{=}\delta^{\text{L}}_1[l \mapsto c_l,s_l,P_l \uplus \{\pi.\mathsf{l}\}][r \mapsto c_r,s_r,P_r{\uplus}\{\pi.\mathsf{r}\}] \\ & \land\, L'_1{=}L''_1[x \mapsto (y,\pi,P)] \\ & \land\, V'_2{=}\delta^{\text{V}}_2{\uplus}\{y\} \land E'_2{=}\delta^{\text{E}}_2{\uplus}[y \mapsto (0{,}0)] \land L'_2{=}\delta^{\text{L}}_2{\uplus}[y \mapsto (0{,}\pi{,}\emptyset)]\big) \end{array} \right\}$$

$$A^2_\pi \triangleq \left\{ \begin{array}{c|l} \big((\delta_1,\delta_2), & \delta_1(x)=\big((y{,}\pi{,}P),l,-\big) \land \big((l{=}0{\land}c_l{=}0) \lor (\delta^{\mathsf{c}}_1(l){=}c_l{\land}c_l{\neq}0)\big) \\ (\delta_1,\delta'_2)\big) & \land\, \delta_2(y)=\big((0{,}\pi{,}\emptyset),0,r\big) \land \delta'_2{=}(\delta^{\text{V}}_2,E'_2,\delta^{\text{L}}_2) \land E'_2{=}\delta^{\text{E}}_2[y \mapsto (c_l,r)] \end{array} \right\}$$

$$A^3_\pi \triangleq \left\{ \begin{array}{c|l} \big((\delta_1,\delta_2), & \delta_1(x)=\big((y{,}\pi{,}P),-,r\big) \land \big((r{=}0{\land}c_r{=}0) \lor (\delta^{\mathsf{c}}_1(r){=}c_r{\land}c_r{\neq}0)\big) \\ (\delta_1,\delta'_2)\big) & \land\delta_2(y)=\big((0{,}\pi{,}\emptyset),l,0\big) \land \delta'_2{=}(\delta^{\text{V}}_2,E'_2,\delta^{\text{L}}_2) \land E'_2{=}\delta^{\text{E}}_2[y \mapsto (l,c_r)] \end{array} \right\}$$

$$A^4_\pi \triangleq \left\{ \begin{array}{c|l} & \delta_1(x)=\big((y{,}\pi{,}P),l,r\big) \land \delta_2(y)=\big((0{,}\pi{,}\emptyset),c_l,c_r\big) \\ \big((\delta_1,\delta_2), & \land\, (l{=}0{\land}c_l{=}0 \lor \delta^{\mathsf{c}}_1(l){=}c_l{\land}c_l{\neq}0) \\ (\delta'_1,\delta'_2)\big) & \land\, (r{=}0{\land}c_r{=}0 \lor \delta^{\mathsf{c}}_1(r){=}c_r{\land}c_r{\neq}0) \\ & \land\, \delta'_1{=}(\delta^{\text{V}}_1,\delta^{\text{E}}_1,\delta^{\text{L}}_1[x{\mapsto}(y{,}0{,}P)]) \land \delta'_2{=}(\delta^{\text{V}}_2,\delta^{\text{E}}_2,\delta^{\text{L}}_2[y{\mapsto}(0{,}0{,}\emptyset)]) \end{array} \right\}$$

$$A^5_\pi \triangleq \left\{ \begin{array}{c|l} \big((\delta_1,\delta_2), & \\ (\delta'_1,\delta_2)\big) & \delta^{\text{L}}_1(x)=(y,s,P{\uplus}\{\pi\}) \land y{\neq}0 \land \delta'_1{=}\big(\delta^{\text{V}}_1,\delta^{\text{E}}_1,\delta^{\text{L}}_1[x \mapsto (y,s,P)]\big) \end{array} \right\}$$

Figure 10.4.: The mathematical actions of `copy_dag`

line 10, it takes x to stage 3 ($c{=}x'$, $s{=}0$). If another thread π' then visits x when it is in stage 2 or 3, it removes its token π' from the promise set P, leaving the node stage unchanged.

Actions The mathematical actions of `copy_dag` are given in Fig. 10.4. The A^1_π describes taking a node x from stage 1 to 2 by thread π. In doing so, it removes its token π from the promise set of x, and adds $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ to the promise sets of its left and right children respectively, indicating that they will be visited by its sub-threads, $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$. It then updates the copy field of x to y, and extends the copy graph with y. This action captures the atomic block of lines 5-7 when successful. The next two sets capture the execution of atomic commands in line 10 by thread π where A^2_π and A^3_π respectively describe updating the left and right edges of the copy node. Once thread π has finished executing line 10 (and has updated the edges of y), it takes x to stage 3 by updating the relevant ghost values. This is described by A^4_π. The A^5_π set describes the case where node x has already been visited by another thread (it is in stage 2 or 3 and thus its copy field is non-zero). Thread π then proceeds by removing its token from x's promise set. We write A_π to denote the actions of thread π:

$A_\pi \triangleq A^1_\pi \cup A^2_\pi \cup A^3_\pi \cup A^4_\pi \cup A^5_\pi$. We can now specify the behaviour of `copy_dag` mathematically.

Mathematical specification Throughout the execution of `copy_dag`, the source dag and its copy (δ, δ'), satisfy the invariant Inv below.

$$\begin{aligned}
\mathsf{Inv}(\delta, \delta') &\triangleq \mathsf{acyc}(\delta) \land \mathsf{acyc}(\delta') \\
&\quad \land\ (\forall \text{x}' \in \delta'.\ \exists! \text{x} \in \delta.\ \delta^{\mathsf{c}}(\text{x}){=}\text{x}') \land (\forall \text{x} \in \delta.\ \exists \text{x}'.\ \mathsf{ic}(\text{x}, \text{x}', \delta, \delta')) \\
\mathsf{acyc}(\delta) &\triangleq \neg \exists \text{x}.\ x \overset{\delta}{\rightsquigarrow}{}^{+} \text{x} \\
\mathsf{ic}(\text{x}, \text{x}', \delta, \delta') &\triangleq (\text{x}{=}0 \land \text{x}'{=}0) \lor \\
&\Big(\text{x}{\neq}0 \land \big[(\text{x}'{=}0 \land \delta^{\mathsf{c}}(\text{x}){=}\text{x}' \land \exists \text{y}.\ \delta^{\mathsf{p}}(\text{y}){\neq}\emptyset \land \text{y} \overset{\delta}{\rightsquigarrow}{}^{*}_{0}\ \text{x}) \\
&\quad \lor \big(\text{x}'{\neq}0 \land \text{x}' \in \delta' \land \exists \pi, \text{l}, \text{r}, \text{l}', \text{r}'.\ \delta(\text{x}){=}((\text{x}', \pi, -), \text{l}, \text{r}) \\
&\qquad \land \delta'(\text{x}'){=}(-, \text{l}', \text{r}') \\
&\qquad \land (\text{l}'{\neq}0 \Rightarrow \mathsf{ic}(\text{l}, \text{l}', \delta, \delta')) \land (\text{r}'{\neq}0 \Rightarrow \mathsf{ic}(\text{r}, \text{r}', \delta, \delta'))\big) \\
&\quad \lor \big(\text{x}'{\neq}0 \land \text{x}' \in \delta' \land \exists \text{l}, \text{r}, \text{l}', \text{r}'.\ \delta(\text{x}){=}((\text{x}', 0, -), \text{l}, \text{r}) \\
&\qquad \land \delta'(\text{x}'){=}(-, \text{l}', \text{r}') \land \mathsf{ic}(\text{l}, \text{l}', \delta, \delta') \land \mathsf{ic}(\text{r}, \text{r}', \delta, \delta')\big)\big]\Big)
\end{aligned}$$

where $\overset{\delta}{\rightsquigarrow}{}^{+}$ denotes the transitive closure of $\overset{\delta}{\rightsquigarrow}$.

Informally, the invariant asserts that δ and δ' are acyclic (first two conjuncts), and that each node x' of the copy dag δ' corresponds to a unique node x of the source dag δ (third conjunct). The last conjunct states that each node x of the source dag (i.e. $\text{x}{\neq}0$) is in one of the three stages described above, via the second disjunct of the icpredicate: i) x is not copied yet (stage 1), in which case there is an unprocessed path from a node y with a non-empty promise set to x, ensuring that it will eventually be visited (first disjunct); ii) x is currently being processed (stage 2) by thread π (second disjunct), and if its children have been copied they also satisfy the invariant; iii) x has been processed completely (stage 3) and thus its children also satisfy the invariant (last disjunct).

The mathematical precondition of `copy_dag`, $\mathsf{P}^\pi(\text{x}, \delta)$, is defined below where x identifies the top node being copied (the argument to `copy_dag`), π denotes the thread identifier, and δ is the source dag. It asserts that π is in the promise set of x, i.e. thread π has an obligation to visit x (first conjunct). Recall that each token uniquely identifies a thread and thus the descendants of π correspond to the sub-threads subsequently spawned by π. As such, prior to spawning new threads the precondition asserts that none of the strict descendants of π can be found anywhere in the promise sets (second conjunct), and π itself is only in the promise set of X (third conjunct). Similarly, neither π nor its descendants have yet processed any nodes (last conjunct). The mathematical postcondition, $\mathsf{Q}^{\pi}(\text{X}, \text{Y}, \delta, \delta')$, is as defined below and asserts that X (in δ) has been copied to Y (in δ'); that π and all its descendants have fulfilled their promises and thus cannot be found in promise sets; and that π and all its descendants have finished processing their charges and thus cannot correspond to the stage field of a node.

$$\begin{aligned}
\mathsf{P}^{\pi}(\text{X}, \delta) \triangleq\; & (\text{X}{=}0 \vee \pi \in \delta^{\mathsf{p}}(\text{X})) \\
& \wedge \forall \pi'.\, \forall \text{Y} \in \delta.\, (\pi' \in \delta^{\mathsf{p}}(\text{Y}) \Rightarrow \pi' \not\sqsubset \pi) \\
& \quad \wedge (\text{X}{\neq}\text{Y} \Rightarrow \pi \notin \delta^{\mathsf{p}}(\text{Y})) \wedge (\delta^{\mathsf{s}}(\text{Y}){=}\pi' \Rightarrow \pi' \not\sqsubseteq \pi) \\
\mathsf{Q}^{\pi}(\text{X}, \text{Y}, \delta, \delta') \triangleq\; & (\text{X}{=}0 \vee (\delta^{\mathsf{c}}(\text{X}){=}\text{Y} \wedge \text{Y} \in \delta')) \\
& \wedge \forall \pi'.\, \forall \text{Z} \in \delta.\, \pi' \in \delta^{\mathsf{p}}(\text{Z}) \vee \delta^{\mathsf{s}}(\text{Z}){=}\pi' \Rightarrow \pi' \not\sqsubseteq \pi
\end{aligned}$$

Observe that when the top level thread (associated with the $\bullet$ token) terminates its execution of `copy_dag(x)`, since $\bullet$ is the maximal token and all other tokens are its descendants (i.e. $\forall \pi.\, \pi \sqsubseteq \bullet$), the second conjunct of $\mathsf{Q}^{\bullet}(\mathtt{x}, \mathtt{ret}, \delta, \delta')$ entails that no tokens can be found anywhere in δ, i.e. $\forall \text{Y}.\, \delta^{\mathsf{p}}(\text{Y}){=}\emptyset \wedge \delta^{\mathsf{s}}(\text{Y}){=}0$. As such, $\mathsf{Q}^{\bullet}(\mathtt{x}, \mathtt{ret}, \delta, \delta')$ together with Inv entails that all nodes in δ have been correctly copied into δ', i.e. only the third disjunct of $\mathsf{ic}(\mathtt{x}, \mathtt{ret}, \delta, \delta')$ in Inv applies.

Recall that as a key proof obligation we must prove that our mathematical assertions are stable against our mathematical actions. This is captured by Lemma 15 below. Part (10.5) states that the invariant Inv is stable against the actions of all threads. That is, if the invariant holds for (δ_1, δ_2), and a thread π updates (δ_1, δ_2) to (δ_3, δ_4), then the invariant holds for (δ_3, δ_4). Parts (10.6) and (10.7) state that the pre- and postconditions of thread π' ($\mathsf{P}^{\pi'}$ and $\mathsf{Q}^{\pi'}$) are stable with respect to the actions of all threads π, but those of its descendants ($\pi \notin \overline{\pi'}$). Observe that despite this latter stipulation, the actions of π are irrelevant and do not affect the stability of $\mathsf{P}^{\pi'}$ and $\mathsf{Q}^{\pi'}$. More concretely, the precondition $\mathsf{P}^{\pi'}$ only holds at the beginning of the program *before* new descendants are spawned (line 9). As such, at these program points $\mathsf{P}^{\pi'}$ is trivially stable against the actions of its (non-existing) descendants. Analogously, the postcondition $\mathsf{Q}^{\pi'}$ only holds at the end of the program *after* the descendant threads have completed their execution and joined. Therefore, at these program points $\mathsf{Q}^{\pi'}$ is trivially stable against the actions of its descendants.

Lemma 15 (copy_dag stability). *For all mathematical objects* $(\delta_1,\delta_2), (\delta_3,\delta_4)$*, and all tokens* π, π'*,*

$$\mathsf{Inv}(\delta_1, \delta_2) \land (\delta_1,\delta_2)\, A_\pi\, (\delta_3,\delta_4) \Rightarrow \mathsf{Inv}(\delta_3, \delta_4) \tag{10.5}$$
$$\mathsf{P}^{\pi'}(\textsc{x}, \delta_1) \land (\delta_1,\delta_2)\, A_\pi\, (\delta_3,\delta_4) \land \pi \not\in \overline{\pi'} \Rightarrow \mathsf{P}^{\pi'}(\textsc{x}, \delta_3) \tag{10.6}$$
$$\mathsf{Q}^{\pi'}(\textsc{x}, \textsc{y}, \delta_1, \delta_2) \land (\delta_1,\delta_2)\, A_\pi\, (\delta_3,\delta_4) \land \pi \not\in \overline{\pi'} \Rightarrow \mathsf{Q}^{\pi'}(\textsc{x}, \textsc{y}, \delta_3, \delta_4) \tag{10.7}$$

Proof. Follows from the definitions of A_π, Inv, P, and Q.

Spatial graphs We represent a mathematical object (δ, δ') in the heap through the icdag (in-copy) predicate below as two disjoint ($*$-separated) dags, as well as a ghost location ($\textsc{d}$) in the ghost heap tracking the current abstract state of each dag. As mentioned earlier, this way of tracking the abstract state of dags in the ghost heap eliminates the need for baking in the abstract state into the model. We implement each dag as a collection of nodes in the heap. A node is represented as three adjacent cells in the heap together with two additional cells in the ghost heap. The cells in the heap track the addresses of the copy ($\textsc{c}$), and the left ($\textsc{l}$) and right ($\textsc{r}$) children, respectively. The ghost locations are used to track the node state ($\textsc{s}$) and the promise set (P).

$$\mathsf{icdag}(\delta_1, \delta_2) \triangleq \textsc{d} \Rrightarrow (\delta_1, \delta_2) * \mathsf{dag}(\delta_1) * \mathsf{dag}(\delta_2) \qquad \mathsf{dag}(\delta) \triangleq \underset{\textsc{x}\in\delta}{\circledast} \mathsf{node}(\textsc{x}, \delta)$$
$$\mathsf{node}(\textsc{x}, \delta) \triangleq \exists \textsc{l}, \textsc{r}, \textsc{c}, \textsc{s}, \mathrm{P}.\ \delta(\textsc{x}){=}(\textsc{c}, \textsc{s}, \mathrm{P}), \textsc{l}, \textsc{r} \land x \mapsto \textsc{c}, \textsc{l}, \textsc{r} * \textsc{x} \Rrightarrow \textsc{s}, \mathrm{P}$$

It is also possible (and perhaps more pleasing) to implement a dag via a *recursive* predicate using the overlapping conjunction $\uplus\!\!\!*$ as follows where $\mathsf{rdag}(\textsc{x}, \delta)$ describes the sub-dag in δ with its top node denoted by $\textsc{x}$:

$$\mathsf{rdag}(\textsc{x}, \delta) \triangleq \mathsf{node}(\textsc{x}, \delta) * \big(\mathsf{rdag}(\delta^{\mathsf{l}}(\textsc{x}), \delta) \uplus\!\!\!* \ \mathsf{rdag}(\delta^{\mathsf{r}}(\textsc{x}), \delta)\big)$$

Note that in the same way that the graph unfolding mechanism of Lemma 14 allows for a more local specification by focusing on the relevant sub-graph at each recursive call point, the recursive dag definition rdag allows us to fold and unfold the dags as required. Here, we choose the flat representation dag to demonstrate an alternative (global) reasoning style.

Spatial specification We can now specify the spatial precondition of `copy_dag`, $\mathsf{Pre}(\text{X}, \pi, \delta)$, as a CoLoSL assertion defined below where X is the top node being copied (the argument of `copy_dag`), π identifies the running thread, and δ denotes the initial top-level dag (where none of the nodes are copied yet). As before, we model our capabilities as elements of the partial commutative monoid given by $(\mathcal{P}(\Pi), \uplus, \emptyset)$. The precondition Pre states that the current thread π holds the capabilities associated with itself and all its descendants ($[\overline{\pi}]$). Thread π will subsequently pass on the descendant capabilities when spawning new sub-threads and reclaim them as the sub-threads return and join. The Pre further asserts that the initial dag δ and its copy currently correspond to δ_1 and δ_2, respectively. That is, since the dags are concurrently manipulated by several threads, to ensure the stability of the shared state assertion to the actions of the environment, Pre states that the initial dag δ may have evolved to another congruent dag δ_1 (captured by the existential quantifier). The Pre also states that the shared state contains the spatial resources of the dags ($\mathsf{icdag}(\delta_1, \delta_2)$), that (δ_1, δ_2) satisfies the invariant Inv, and that the source dag δ_1 satisfies the mathematical precondition P^π. The spatial actions on the shared state are declared in I where mathematical actions are simply lifted to spatial ones indexed by the associated capability. That is, if thread π holds the π capability, and the actions of π (A_π) admit the update of the mathematical object (δ_1, δ_2) to (δ_1', δ_2'), then thread π may update the spatial resources $\mathsf{icdag}(\delta_1, \delta_2)$ to $\mathsf{icdag}(\delta_1', \delta_2')$. Finally, the spatial postcondition Post is analogous to Pre and further states that node X has been copied to Y.

$$\mathsf{Pre}(\text{X}, \pi, \delta) \triangleq [\overline{\pi}] * \boxed{\begin{array}{l} \exists \delta_1, \delta_2.\ \mathsf{icdag}(\delta_1, \delta_2) \\ \quad * (\delta \dot{\cong} \delta_1 \wedge \mathsf{Inv}(\delta_1, \delta_2) \wedge \mathsf{P}^\pi(\text{X}, \delta_1)) \end{array}}_{I}$$

$$\mathsf{Post}(\text{X}, \text{Y}, \pi, \delta) \triangleq [\overline{\pi}] * \boxed{\begin{array}{l} \exists \delta_1, \delta_2.\ \mathsf{icdag}(\delta_1, \delta_2) \\ \quad * (\delta \dot{\cong} \delta_1 \wedge \mathsf{Inv}(\delta_1, \delta_2) \wedge \mathsf{Q}^\pi(\text{X}, \text{Y}, \delta_1, \delta_2)) \end{array}}_{I}$$

$$I \triangleq \left\{ [\pi] : \mathsf{icdag}(\delta_1, \delta_2) \wedge (\delta_1, \delta_2) A_\pi (\delta_1', \delta_2') \rightsquigarrow \mathsf{icdag}(\delta_1', \delta_2') \right\}$$

Verifying `copy_dag` We give a proof sketch of `copy_dag` in Fig. 10.5. As mentioned earlier, unlike the `span` example in §10.1 where we delegated part of the correctness argument to the spatial representation of the graph, for `copy_dag` we carry out the entire correctness argument at the mathematical level. As such, one thing jumps out when looking at the assertions at each program point: they have *identical* spatial parts in the shared state: $\mathsf{icdag}(\delta_1, \delta_2)$. Indeed, the spatial graph in the heap is changing constantly, due both to the actions of this thread and the environment. Nevertheless, the spatial graph in the heap remains in sync with the mathematical object (δ_1, δ_2), however (δ_1, δ_2) may be changing. Whenever this thread interacts with the shared state, the mathematical object (δ_1, δ_2) changes, reflected by the changes to the pure mathematical facts. Changes to (δ_1, δ_2) due to other threads in the environment are handled by the existential quantification of δ_1 and δ_2.

On line 3 we check if `x` is 0. If so the program returns and the postcondition, $\mathsf{Post}(\mathtt{x}, 0, \delta, \pi)$, follows trivially from the definition of the precondition $\mathsf{Pre}(\mathtt{x}, \delta, \pi)$. If $\mathtt{x} \neq 0$, then the atomic block of lines 5-7 is executed. We first check if `x` is copied; if so we set `b` to false, perform action A^5_π (i.e. remove π from the promise set of `x`) and thus arrive at the desired postcondition $\mathsf{Post}(\mathtt{x}, \delta_1^{\mathsf{c}}(\mathtt{x}), \pi, \delta)$. On the other hand, if `x` is not copied, we set `b` to true and perform A^1_π. That is, we remove π from the promise set of `x`, and add $\pi.\mathsf{l}$ and $\pi.\mathsf{r}$ to the left and right children of `x`, respectively. In doing so, we obtain the mathematical preconditions $\mathsf{P}^{\delta_1}(\textsc{l}, \pi.\mathsf{l})$ and $\mathsf{P}^{\delta_1}(\textsc{r}, \pi.\mathsf{r})$. On line 8 we check whether the thread did copy `x` and has thus incurred an obligation to call `copy_dag` on `x`'s children. If this is the case, we load the left and right children of `x` into `l` and `r`, and subsequently call `copy_dag` on them (line 9). To obtain the preconditions of the recursive calls, we duplicate the shared state twice ($\boxed{P}_I \overset{\textsc{Copy}\times 2}{\Longrightarrow} \boxed{P}_I * \boxed{P}_I * \boxed{P}_I$), drop the irrelevant pure assertions, and split $[\overline{\pi}]$. We then use the Par rule (Def. 124) to distribute the resources between the sub-threads and collect them back when they join. Subsequently, we combine the subjective views using Merge. Finally, on line 10 we perform actions A^2_π, A^3_π and A^4_π in order to update the edges of `y`, and arrive at the postcondition $\mathsf{Post}(\mathtt{x}, \mathtt{y}, \pi, \delta)$.

Copying graphs Recall that a dag is a directed graph that is *acyclic*. However, the `copy_dag` program does not depend on the acyclicity of the dag at `x` and thus `copy_dag` may be used to copy *both* dags and cyclic graphs. The specification of `copy_dag` for cyclic graphs is rather similar to that of dags. More concretely, the spatial pre- and postcondition (Pre and Post), as well as the mathematical pre- and postcondition (P and Q) remain unchanged, while the invariant Inv is weakened to allow for cyclic graphs. That is, the Inv for cyclic graphs does not include the first two conjuncts asserting that δ and δ' are acyclic. As such, when verifying `copy_dag` for cyclic graphs, the proof obligation for establishing the Inv stability (i.e. Lemma 15(10.5)) is somewhat simpler. The other stability proofs (Lemma 15(10.6) and (10.7)) and the proof sketch in Fig. 10.5 are essentially unchanged.

```
1.  struct node {struct node *c, *l, *r};
    $\{\mathsf{Pre}(\mathtt{x},\pi,\delta)\}$
2.  copy_dag(struct node *x) {struct node *l,*r,*ll,*rr,*y; bool b;
    $\left\{ [\overline{\pi}] * \boxed{\exists\delta_1,\delta_2.\ \mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \mathsf{P}^{\pi}(\mathtt{x},\delta_1))}_I \right\}$
3.    if(!x){ return 0; }
    $\left\{ [\overline{\pi}] * \mathtt{ret}\dot{=}0 * \boxed{\exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \mathsf{Q}^{\pi}(\mathtt{x},\mathtt{ret},\delta_1,\delta_2))}_I \right\}$
4.    y = malloc(sizeof(struct node));
    $\left\{ [\overline{\pi}] * \boxed{\exists\delta_1,\delta_2.\ \mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \mathsf{P}^{\pi}(\mathtt{x},\delta_1))}_I * \mathtt{y} \mapsto 0,0,0 * \mathtt{y} \Rrightarrow \pi,\emptyset \right\}$

5.    <if(x->c){ b = false;      // Perform the action $A^5_\pi$
    $\left\{ \begin{array}{l} [\overline{\pi}] * \boxed{\exists\delta_1,\delta_2.\ \mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \mathsf{Q}^{\pi}(\mathtt{x},\delta^{\mathsf{c}}_1(\mathtt{x}),\delta_1,\delta_2) \wedge \delta^{\mathsf{c}}_1(\mathtt{x}) \neq 0)}_I \\ * \ \mathtt{y} \mapsto 0,-,- * \mathtt{y} \Rrightarrow \pi,\emptyset * \mathtt{b}\dot{=}0 \end{array} \right\}$
6.    }else{ x->c = y; b = true;   // Perform the action $A^1_\pi$
    $\left\{ [\overline{\pi}] * \boxed{\begin{array}{l} \exists\delta_1,\delta_2.\ \mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \forall \mathrm{Y}\in\delta_1.\ \pi\notin\delta^{\mathsf{p}}_1(\mathrm{Y}) \wedge \\ (\mathtt{x}\neq\mathrm{Y} \Rightarrow \pi\neq\delta^{\mathsf{s}}_1(\mathrm{Y})) \wedge \exists \mathrm{L},\mathrm{R}.\ \delta_1(\mathtt{x}) = (\mathtt{y},\pi,-,\mathrm{L},\mathrm{R}) \wedge \mathtt{y}\dot{\in}\delta_2 \wedge \mathsf{P}^{\pi.\mathsf{l}}(\mathrm{L},\delta_1) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\mathrm{R},\delta_1)) \end{array}}_I * \mathtt{b}\dot{=}1 \right\}$
7.    }>
8.    if(b){ l = x->l; r = x->r;
    $\left\{ [\overline{\pi}] * \boxed{\begin{array}{l} \exists\delta_1,\delta_2.\ \mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \forall \mathrm{Y}\in\delta_1.\ \pi\notin\delta^{\mathsf{p}}_1(\mathrm{Y}) \wedge \\ (\mathtt{x}\neq\mathrm{Y} \Rightarrow \pi\neq\delta^{\mathsf{s}}_1(\mathrm{Y})) \wedge \ \delta_1(\mathtt{x}) = (\mathtt{y},\pi,-,\mathtt{l},\mathtt{r}) \wedge \mathtt{y}\dot{\in}\delta_2 \wedge \mathsf{P}^{\pi.\mathsf{l}}(\mathtt{l},\delta_1) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\mathtt{r},\delta_1)) \end{array}}_I \right\}$
    $\left\{ \begin{array}{l} [\overline{\pi}] * \boxed{\begin{array}{l} \exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \forall \mathrm{Y}\in\delta_1.\ \pi\notin\delta^{\mathsf{p}}_1(\mathrm{Y}) \wedge \\ (\mathtt{x}\neq\mathrm{Y} \Rightarrow \pi\neq\delta^{\mathsf{s}}_1(\mathrm{Y})) \wedge \delta_1(\mathtt{x}) = (\mathtt{y},-,\pi,\mathtt{l},\mathtt{r}) \wedge \mathtt{y}\dot{\in}\delta_2) \end{array}}_I \\ * \ [\overline{\pi.\mathsf{l}}] * \boxed{\exists\delta_1,\delta_2.\ \mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \mathsf{P}^{\pi.\mathsf{l}}(\mathtt{l},\delta_1))}_I \\ * \ [\overline{\pi.\mathsf{r}}] * \boxed{\exists\delta_1,\delta_2.\ \mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \mathsf{P}^{\pi.\mathsf{r}}(\mathtt{r},\delta_1))}_I \end{array} \right\}$
    $\left\{ [\overline{\pi}] * \boxed{\begin{array}{l} \exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \forall \mathrm{Y}\in\delta_1.\ \pi\notin\delta^{\mathsf{p}}_1(\mathrm{Y}) \\ \wedge\ (\mathtt{x}\neq\mathrm{Y} \Rightarrow \pi\neq\delta^{\mathsf{s}}_1(\mathrm{Y})) \wedge \delta_1(\mathtt{x}) = (\mathtt{y},-,\pi,\mathtt{l},\mathtt{r}) \wedge \mathtt{y}\dot{\in}\delta_2) \end{array}}_I \begin{array}{l} * \mathsf{Pre}(\mathtt{l},\pi.\mathsf{l},\delta) \\ * \mathsf{Pre}(\mathtt{r},\pi.\mathsf{r},\delta) \end{array} \right\}$
                 $\{\mathsf{Pre}(\mathtt{l},\pi.\mathsf{l},\delta)\}$        ||     $\{\mathsf{Pre}(\mathtt{r},\pi.\mathsf{r},\delta)\}$
9.               ll = copy_dag(l)             ||     rr = copy_dag(r)
               $\{\mathsf{Post}(\mathtt{l},\mathtt{ll},\pi.\mathsf{l},\delta)\}$      ||   $\{\mathsf{Post}(\mathtt{r},\mathtt{rr},\pi.\mathsf{r},\delta)\}$
    $\left\{ [\overline{\pi}] * \boxed{\begin{array}{l} \exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \forall \mathrm{Y}\in\delta_1.\ \pi\notin\delta^{\mathsf{p}}_1(\mathrm{Y}) \\ \wedge\ (\mathtt{x}\neq\mathrm{Y} \Rightarrow \pi\neq\delta^{\mathsf{s}}_1(\mathrm{Y})) \wedge \delta_1(\mathtt{x}) = (\mathtt{y},-,\pi,\mathtt{l},\mathtt{r}) \wedge \mathtt{y}\dot{\in}\delta_2) \end{array}}_I \begin{array}{l} * \mathsf{Post}(\mathtt{l},\mathtt{ll},\pi.\mathsf{l},\delta) \\ * \mathsf{Post}(\mathtt{r},\mathtt{rr},\pi.\mathsf{r},\delta) \end{array} \right\}$
    $\left\{ [\overline{\pi}] * \boxed{\begin{array}{l} \exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \forall \mathrm{Y}\in\delta_1. \pi\notin\delta^{\mathsf{p}}_1(\mathrm{Y}) \wedge (\mathtt{x}\neq\mathrm{Y} \Rightarrow \pi\neq\delta^{\mathsf{s}}_1(\mathrm{Y})) \\ \wedge\ \delta_1(\mathtt{x}) = (\mathtt{y},-,\pi,\mathtt{l},\mathtt{r}) \wedge \mathtt{y}\dot{\in}\delta_2 \wedge \mathsf{Q}^{\pi.\mathsf{l}}(\mathtt{l},\mathtt{ll},\delta_1,\delta_2) \wedge \mathsf{Q}^{\pi.\mathsf{r}}(\mathtt{r},\mathtt{rr},\delta_1,\delta_2)) \end{array}}_I \right\}$
10.     <y->l = ll>;<y->r = rr>; // Perform $A^2_\pi$, $A^3_\pi$ and $A^4_\pi$ in order
    $\left\{ [\overline{\pi}] * \boxed{\exists\delta_1,\delta_2.\ \mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \mathsf{Q}^{\pi}(\mathtt{x},\mathtt{y},\delta_1,\delta_2))}_I \right\}$
11.     return y; $\left\{ [\overline{\pi}] * \boxed{\exists\delta_1,\delta_2.\ \mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \mathsf{Q}^{\pi}(\mathtt{x},\mathtt{ret},\delta_1,\delta_2))}_I \right\}$
12.   }else{
    $\left\{ [\overline{\pi}] * \boxed{\exists\delta_1,\delta_2.\mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \mathsf{Q}^{\pi}(\mathtt{x},\delta^{\mathsf{c}}_1(\mathtt{x}),\delta_1,\delta_2) \wedge \delta^{\mathsf{c}}_1(\mathtt{x}) \dot{\neq} 0)}_I \begin{array}{l} * \mathtt{y} \mapsto 0,-,- \\ * \mathtt{y} \Rrightarrow \pi,\emptyset \end{array} \right\}$
13.     free(y, sizeof(struct node)) ; return x->c;
    $\left\{ [\overline{\pi}] * \boxed{\exists\delta_1,\delta_2.\ \mathsf{icdag}(\delta_1,\delta_2) * (\delta \dot{\simeq} \delta_1 \wedge \mathsf{Inv}(\delta_1,\delta_2) \wedge \mathsf{Q}^{\pi}(\mathtt{x},\mathtt{ret},\delta_1,\delta_2))}_I \right\}$
14. } }  $\{\mathsf{Post}(\mathtt{x},\mathtt{ret},\pi,\delta)\}$
```

Figure 10.5.: The code and a proof sketch of `copy_dag`

10.3. Parallel Speculative Shortest Path (Dijkstra)

Given a graph with `size` vertices, the weighted adjacency matrix `a`, and a designated source node `src`, Dijkstra's sequential algorithm calculates the shortest path from `src` to all other nodes incrementally. To do this, it maintains a cost array `c`, and two sets of vertices: those processed thus far (`done`), and those yet to be processed (`work`). The cost for each node (bar `src` itself) is initialised with the value of the adjacency matrix (i.e. `c`[`src`]=0; `c`[`i`]=`a`[`src`][`i`] for `i`≠`src`). Initially, all vertices are in `work` and the algorithm proceeds by iterating over `work` performing the following two steps at each iteration. First, it extracts a node `i` with the *cheapest* cost from `work` and inserts it to `done`. Second, for each vertex `j`, it updates its cost (`c`[`j`]) to *min*{`c`[`j`], `c`[`i`]+`a`[`i`][`j`]}. This greedy strategy ensures that at any one point the cost associated with the nodes in `done` is minimal. Once the `work` set is exhausted, `c` holds the minimal cost for all vertices.

We study a *parallel non-greedy* variant of Dijkstra's shortest path algorithm, `parallel_dijkstra` in Fig. 10.6, with `work` and `done` implemented as bit arrays. We initialize the `c`, `work` and `done` arrays as described above (lines 2-5), and find the shortest path from the source `src` concurrently, by spawning multiple threads, each executing the non-greedy `dijkstra` (line 6). The code for `dijkstra` is given in Fig. 10.6. In this non-greedy implementation, at each iteration an *arbitrary* node from the `work` set is selected rather than one with minimal cost. Unlike the greedy variant, when a node is processed and inserted into `done`, its associated cost is not necessarily the cheapest. As such, during the second step of each iteration, when updating the cost of node `j` to *min*{`c`[`j`], `c`[`i`]+`a`[`i`][`j`]} (as described above), we must further check if `j` is already processed. This is because if the cost of `j` goes down, the cost of its adjacent siblings may go down too and thus `j` needs to be *reprocessed.* When this is the case, `j` is removed from `done` and reinserted into `work` (lines 10-12). If on the other hand `j` is unprocessed (and is in `work`), we can safely decrease its cost (lines 8-9). Lastly, if `j` is currently being processed by another thread, we must wait until it is processed (loop back and try again).

The `parallel_dijkstra` algorithm is an instance of *speculative parallelism* [26]: each thread running `dijkstra` assumes that the costs of nodes in `done` are final and will not change as a result of processing the nodes

```
void parallel_dijkstra(int[][] a, int[] c, int size, src){
  work[size], done[size];
  for (i=0; i<size; i++){
    c[i] = a[src][i]; work[i] = 1; done[i] = 0;
  }; c[src] = 0;
  dijkstra(a,c,size,work,done) || ... || dijkstra(a,c,size,work,done)
  return c;
}
```

```
void dijkstra(int[][] a, int[] c, int size, bitarray work, done){
  i=0;
  while(done != $2^{size}$-1){ b=<CAS(work[i],1,0)>;
    if(b){ cost=c[i];
      for(j=0; j<size; j++){ newcost=cost+a[i][j]; b=true;
        do{ oldcost = c[j];
          if(newcost < oldcost){
            b = <CAS(work[j],1,0)>;
            if(b){ b=<CAS(c[j],oldcost,newcost)>; <work[j]=1>; }
            else { b=<CAS(done[j],1,0)>;
              if(b){ b=<CAS(c[j],oldcost,newcost)>;
                if(b){ <work[j]=1> } else { <done[j]=1> }
          } } }
        } while(!b)
      } < done[i]=1 >;
    } i=(i+1) mod size;
} }
```

Figure 10.6.: A parallel non-greedy variant of Dijkstra's algorithm

in work. However, if at a later point it detects that its assumption was wrong, it reinserts the affected nodes into work and recomputes their costs.

Mathematical graphs Similar to the graphs in §10.1, we define our mathematical graphs, $\gamma \in \textsc{Graph}$, as triples in $(\mathrm{V} \times \mathrm{E} \times \mathrm{L})$ where V is the vertex set; $\mathrm{E} : \mathrm{V} \to (\mathrm{V} \to \mathcal{W})$ is the weighted adjacency function with weights $\mathcal{W} \triangleq \mathbb{N} \uplus \{\infty\}$, and $\mathrm{L} : \mathrm{V} \to \mathrm{D}$ is the vertex labelling function with the label set D defined shortly. Given a graph (V, E, L), we use the matrix notation for adjacency functions and write $E[i][j]$ for $E(i)(j)$.

Unlike span (§10.1) and copy_dag (§10.2) where a new thread is spawned at every recursive call point, in parallel_dijkstra the number of threads to run concurrently is decided at the beginning (line 6) and remains unchanged thereafter. This allows for a simpler token mechanism. We define our tokens as elements of the (countably) infinite set $\theta \in \Theta \triangleq \mathbb{N} \setminus \{0, 1\}$. We refer to the thread with token θ simply as thread θ. Recall that each node x in the graph can be either: unprocessed (in work); processed (in done); or under process by a thread (neither in work nor in done). We define our labels as $\mathrm{D} \triangleq \mathcal{W} \times \big(\{0, 1\} \uplus \Theta\big) \times (\mathrm{V} \to \{\circ, \bullet\} \uplus \mathcal{W})$. The first component denotes the cost of the shortest path from the source (so far) to the node. The second component describes the node state (0 for unprocessed, 1 for processed, and θ when under process by thread θ). The last component denotes the *responsibility* function. Recall that when a thread is processing a node, it iterates over all vertices examining whether their cost can be improved. To do this, at each iteration the thread records the current cost of node j under inspection in oldcost (line 6). If the cost may be improved (i.e. the conditional of line 7 succeeds), it then *attempts* to update the cost of j with the improved value (lines 9, 11). Note that since the cost associated with j may have changed from the initial cost recorded (oldcost), the update operation may fail and thus the thread needs to re-examine j. To track the iteration progress, for each node the responsibility function records whether i) its cost is yet to be examined ($\circ$); ii) its cost has been examined ($\bullet$); or iii) its cost is currently being examined ($c \in \mathcal{W}$) with its initial cost recorded as c (oldcost$=c$). We use the string notation for responsibility functions and write e.g. $\bullet^n.c.\circ^m$, when the first n nodes are mapped to $\bullet$, the (n+1)st node is mapped to c, and the last m nodes are mapped to $\circ$. We write $\bigcirc$ (resp. $\bullet$) for a function that maps all elements to $\circ$ (resp. $\bullet$).

Given a graph $\gamma=(V,E,L)$, we write γ^{V} for V, γ^{E} for E, and γ^{L} for L. We write $\gamma^{\mathsf{c}}(x)$, $\gamma^{\mathsf{s}}(x)$ and $\gamma^{\mathsf{r}}(x)$, for the first, second and third projections of $L(x)$, respectively. Two graphs are *congruent* if they have equal vertices and edges: $\gamma_1 \cong \gamma_2 \triangleq \gamma_1^{\mathrm{V}}=\gamma_2^{\mathrm{V}} \wedge \gamma_1^{\mathrm{E}}=\gamma_2^{\mathrm{E}}$. We define the weighted path relation ($\overset{\gamma}{\rightsquigarrow}_c$), and its reflexive transitive closure as follows:

$$x \overset{\gamma}{\rightsquigarrow}_c y \triangleq (\gamma^{\mathrm{E}})[x][y]=c$$
$$x \overset{\gamma}{\rightsquigarrow}{}^*_c\, y \triangleq (x=y \wedge c=0) \vee (\exists c_1,c_2,z.\ c=c_1+c_2 \wedge x \overset{\gamma}{\rightsquigarrow}_{c_1} z \wedge z \overset{\gamma}{\rightsquigarrow}{}^*_{c_2}\, y)$$

Actions We define several families of actions in Fig. 10.7, each of which indexed by a token θ. The A^1_θ describes the `CAS` operation of line 3 in the algorithm: the state of a node is changed from unprocessed to being processed by thread θ (i is removed from `work`). The A^2_θ describes a *ghost* action at line 6 for iteration j when storing the current cost of j in `oldcost`. The thread has not yet examined the cost of node j ($R[j]=\circ$). It then reads the current cost (c') of j and (ghostly) updates the responsibility function. The A^3_θ describes the `CAS` operations of lines 8 and 10 when successful: when processing i, we discovered that the cost of j may be improved ($c+E[i][j] \leq c'$). In the former case, j is currently unprocessed (in `work`, $s=0$), while in the latter j is processed (in `done`, $s=1$). In both cases, we remove j from the respective set and temporarily change its state to under process by θ until its cost is updated and it is reinserted into the relevant set. The A^4_θ describes the `CAS` operations in lines 9 and 11 when successful. The cost of j has not changed since we first read it ($R[j]=c'$) and we discovered that this cost may be improved ($c''\leq c'$). The responsibility of i towards j is then marked as fulfilled ($R'[j]=\bullet$) and the cost of j is updated until it is subsequently reinserted into `work` via A^5_θ. The A^5_θ denotes the reinsertion of j into `work` in lines 9 and 12 following *successful* `CAS` operations at lines 9 and 11. The state of j is changed to 0 to reflect its insertion to `work`. The A^6_θ and A^7_θ sets respectively describe the reinsertion of j into `work` and `done` in lines 9 and 12, following *failed* `CAS` operations at lines 9 and 11. When attempting to update the cost of j, we discovered that the cost of j has changed since we first read it ($c'\neq c''$). We thus reinsert j into the relevant set and (ghostly) update the responsibility function to reflect that j is to be re-examined

$$A^1_\theta \triangleq \left\{ \big((V,E,L),(V,E,L')\big) \;\middle|\; L(i){=}(c,0,\bigcirc) \land L'{=}L[i \mapsto (c,\theta,\bigcirc)] \right\}$$

$$A^2_\theta \triangleq \left\{ \big((V,E,L),(V,E,L')\big) \;\middle|\; \begin{array}{l} L(i){=}(c,\theta,R) \land \forall k{<}j.\ R[k]{=}\bullet \land R[j]{=}\circ \\ \land\, L(j){=}(c',-,-) \\ \land\, R'{=}R[j \mapsto c'] \land L'{=}L[i \mapsto (c,\theta,R')] \end{array} \right\}$$

$$A^3_\theta \triangleq \left\{ \big((V,E,L),(V,E,L')\big) \;\middle|\; \begin{array}{l} L(i){=}(-,\theta,R) \land R[j]{=}c' \land c{+}E[i][j] \leq c' \\ \land\, L(j){=}(c,s,R') \land s{\in}\{0,1\} \land L'{=}L[j{\mapsto}(c,\theta,R')] \end{array} \right\}$$

$$A^4_\theta \triangleq \left\{ \big((V,E,L),(V,E,L')\big) \;\middle|\; \begin{array}{l} L(i){=}(c,\theta,R) \land R[j]{=}c' \land L(j){=}(c',\theta,R'') \\ \land\, c''{=}c{+}E[i][j] \land c''{<}c' \\ \land\, R'{=}R[j \mapsto \bullet] \land L'{=}L[i \mapsto (c,\theta,R')][j \mapsto (c'',\theta,R'')] \end{array} \right\}$$

$$A^5_\theta \triangleq \left\{ \big((V,E,L),(V,E,L')\big) \;\middle|\; \begin{array}{l} L(i){=}(c,\theta,R) \land R[j]{=}\bullet \land L(j){=}(c',\theta,-) \\ \land\, L'{=}L[j \mapsto (c',0,\bigcirc)] \end{array} \right\}$$

$$A^6_\theta \triangleq \left\{ \big((V,E,L),(V,E,L')\big) \;\middle|\; \begin{array}{l} L(i){=}(c,\theta,R) \land R[j]{=}c'' \land L(j){=}(c',\theta,\bigcirc) \land c'{\neq}c'' \\ \land\, R'{=}R[j{\mapsto}\circ] \land L'{=}L[i \mapsto (c,\theta,R')][j \mapsto (c',0,\bigcirc)] \end{array} \right\}$$

$$A^7_\theta \triangleq \left\{ \big((V,E,L),(V,E,L')\big) \;\middle|\; \begin{array}{l} L(i){=}(c,\theta,R) \land R[j]{=}c'' \land L(j){=}(c',\theta,\text{●}) \land c'{\neq}c'' \\ \land\, R'{=}R[j \mapsto \circ] \land L'{=}L[i \mapsto (c,\theta,R')][j \mapsto (c',1,\text{●})] \end{array} \right\}$$

$$A^8_\theta \triangleq \left\{ \big((V,E,L),(V,E,L')\big) \;\middle|\; \begin{array}{l} L(i){=}(c,\theta,R) \land R[j]{=}c' \land c{+}E[i][j] \geq c' \\ \land\, R'{=}[j \mapsto \bullet]R \land L'{=}[i \mapsto (c,\theta,R')]L \end{array} \right\}$$

$$A^9_\theta \triangleq \left\{ \big((V,E,L),(V,E,L')\big) \;\middle|\; L(x){=}(c,\theta,\text{●}) \land L'{=}L[x \mapsto (c,1,\text{●})] \right\}$$

Figure 10.7.: The mathematical actions of `dijkstra`

($R'[j]{=}\circ$). The A^8_θ describes a ghost action in line 7 when the conditional fails: examining j yielded no cost improvement and thus the responsibility of i towards j is marked as fulfilled. Lastly, the A^9_θ captures the atomic operation in line 15: processing of i is at an end since all nodes have been examined. The state of i is thus changed to processed (i is inserted into `done`). We write A_θ for actions of θ, i.e. $A_\theta {\triangleq} \bigcup_{i \in \{1 \ldots 9\}} A^i_\theta$.

Mathematical invariant Throughout the execution of `dijkstra` for a source node $\textsc{src}$, the graph γ satisfies the invariant $\mathsf{Inv}(\textsc{src}, \gamma)$ below.

$$\begin{aligned} \mathsf{Inv}(\gamma, \textsc{src}) &\triangleq \forall \textsc{x} \in \gamma.\ \mathsf{min}^{\textsc{src}}_\gamma(\textsc{x}, \gamma^{\mathsf{c}}(\textsc{x})) \\ &\qquad\qquad \lor \big(\exists \textsc{y},\textsc{z},\textsc{c}.\ \mathsf{min}^{\textsc{src}}_\gamma(\textsc{y}, \gamma^{\mathsf{c}}(\textsc{y})) \land \gamma(\textsc{y}){\neq}1 \land \gamma^{\mathsf{r}}[\textsc{y}][\textsc{z}]{=}0 \\ &\qquad\qquad\qquad \land\, \textsc{y} \overset{\gamma}{\rightsquigarrow}_{\textsc{c}} \textsc{z} \land \mathsf{wit}^{\textsc{src}}_\gamma(\gamma^{\mathsf{c}}(\textsc{y}){+}\textsc{c}, \textsc{z}, \textsc{x})\big) \\ \mathsf{min}^{\textsc{src}}_\gamma(\textsc{x}, \textsc{c}) &\triangleq \mathit{min}\{c \mid \textsc{src} \overset{\gamma}{\rightsquigarrow}^{*}_{c} \textsc{x}\} = \textsc{c} \\ \mathsf{wit}^{\textsc{src}}_\gamma(\textsc{c}, \textsc{z}, \textsc{x}) &\triangleq \mathsf{min}^{\textsc{src}}_\gamma(\textsc{z}, \textsc{c}) \land \gamma^{\mathsf{c}}(\textsc{z}) > \textsc{c} \\ &\qquad \land (\textsc{z}{=}\textsc{x} \lor (\exists \textsc{c}', \textsc{w}.\ \textsc{z} \overset{\gamma}{\rightsquigarrow}_{\textsc{c}'} \textsc{w} \land \mathsf{wit}^{\textsc{src}}_\gamma(\textsc{c}{+}\textsc{c}', \textsc{w}, \textsc{x}))) \end{aligned}$$

The $\mathsf{Inv}(\gamma, \textsc{src})$ asserts that for any node $\textsc{x}$, either its associated cost from $\textsc{src}$ is minimal; or there is a minimal path to $\textsc{x}$ from a node $\textsc{y}$ (via $\textsc{z}$), such that the cost of $\textsc{y}$ is minimal and $\textsc{y}$ is either unprocessed or is being processed. Moreover, none of the nodes along this path (except $\textsc{y}$) are yet associated with their correct (minimal) cost. As such, when $\textsc{y}$ is finally processed, its effect will be propagated down this path, correcting the costs of the nodes along the way. Observe that when `dijkstra` terminates, since all nodes are processed (i.e. $\forall \textsc{x}.\ \gamma^{\mathsf{s}}(\textsc{x}){=}1$), the $\mathsf{Inv}(\gamma, \textsc{src})$ entails that the cost associated with all nodes is minimal.

Lemma 16 (`dijkstra` stability). *For all graphs γ, γ', source nodes $\textsc{src}$, and tokens θ, the $\mathsf{Inv}(\gamma, \textsc{src})$ invariant is stable with respect to A_θ:*

$$\mathsf{Inv}(\gamma, \textsc{src}) \wedge \gamma \, A_\theta \, \gamma' \Rightarrow \mathsf{Inv}(\gamma', \textsc{src})$$

Proof. Follows from the definitions of A_θ and Inv.

Spatial graphs We represent a mathematical graph γ in the heap via the $\mathsf{g}(\gamma)$ predicate below as multiple $*$-separated arrays: two bit-arrays for the `work` and `done` sets, a two-dimensional array for the adjacency matrix, a one dimensional array for the cost function, and two ghost arrays for the label function (one for the responsibility function, another for the node states).

$$\begin{aligned}
\mathsf{g}(\gamma) &\triangleq \mathsf{work}(\gamma) * \mathsf{done}(\gamma) * \mathsf{adj}(\gamma) * \mathsf{cost}(\gamma) * \mathsf{resp}(\gamma) * \mathsf{state}(\gamma)\\
\mathsf{work}(\gamma) &\triangleq \mathop{\circledast}_{i \in \{i \mid \gamma^{\mathsf{s}}(i)=0\}} \big(\texttt{work}[i] \mapsto 1\big) * \mathop{\circledast}_{i \in \{i \mid \gamma^{\mathsf{s}}(i)\neq 0\}} \big(\texttt{work}[i] \mapsto 0\big)\\
\mathsf{done}(\gamma) &\triangleq \mathop{\circledast}_{i \in \{i \mid \gamma^{\mathsf{s}}(i)=1\}} \big(\texttt{done}[i] \mapsto 1\big) * \mathop{\circledast}_{i \in \{i \mid \gamma^{\mathsf{s}}(i)\neq 1\}} \big(\texttt{done}[i] \mapsto 0\big)\\
\mathsf{adj}(\gamma) &\triangleq \mathop{\circledast}_{i \in \gamma} \big(\mathop{\circledast}_{j \in \gamma} \mathsf{a}[i][j] \mapsto \gamma^{\mathrm{E}}[i][j]\big) \qquad \mathsf{cost}(\gamma) \triangleq \mathop{\circledast}_{i \in \gamma} \big(\mathsf{c}[i] \mapsto \gamma^{\mathsf{c}}(i)\big)\\
\mathsf{resp}(\gamma) &\triangleq \mathop{\circledast}_{i \in \gamma} \big(\mathop{\circledast}_{j \in \gamma} r[i][j] \Rrightarrow \gamma^{\mathsf{r}}[i][j]\big) \qquad \mathsf{state}(\gamma) \triangleq \mathop{\circledast}_{i \in \gamma} \big(s[i] \Rrightarrow \gamma^{\mathsf{s}}(i)\big)
\end{aligned}$$

Spatial specification We specify the spatial precondition of `dijkstra`, $\mathsf{Pre}(\theta,\gamma_0)$, as a CoLoSL assertion defined below where θ identifies the running thread, and γ_0 denotes the original graph (at the beginning of `parallel_dijkstra`, before spawning new threads). We instantiate our user-defined capabilities as sets of tokens in $\mathcal{P}(\Theta)$. That is, the partial commutative monoid of user-defined capabilities is given by $(\mathcal{P}(\Theta), \uplus, \emptyset)$. The precondition Pre states that the current thread θ holds the $[\theta]$ capability, that the original graph γ_0 may have evolved to another congruent graph γ (captured by the existential quantifier) satisfying the invariant Inv, and that the shared state contains the spatial resources of the graph $\mathsf{g}(\gamma)$. As before, the spatial actions on the shared state are declared in I by lifting mathematical actions to spatial ones indexed by the corresponding capability. Finally, the spatial postcondition Post is analogous to Pre and further states that all nodes in γ are processed (in `done`).

$$\begin{aligned}
\mathsf{Pre}(\theta,\gamma_0) &\triangleq [\theta] * \boxed{\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \texttt{src}))}_I \\
\mathsf{Post}(\theta,\gamma_0) &\triangleq [\theta] * \boxed{\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma, \texttt{src}) \land \forall \mathrm{x} \in \gamma.\ \gamma^{\mathsf{s}}(\mathrm{x}) \doteq 1)}_I \\
I &\triangleq \left\{ [\theta] : \mathsf{g}(\gamma) \land \gamma\, A_\theta\, \gamma' \rightsquigarrow \mathsf{g}(\gamma') \right\}
\end{aligned}$$

Verifying `parallel_dijkstra` A proof sketch of `dijkstra` is given in Figs. 10.8-10.9. As with `copy_dag` in §10.2, at all proof points the spatial part ($\mathsf{g}(\gamma)$) remains unchanged and the changes to the graph are reflected in the changes to the pure mathematical assertions. Observe that when all threads return, the pure part of the postcondition ($\mathsf{Inv}(\gamma, \texttt{src}) \land \forall \mathrm{x} \in \gamma.\ \gamma^{\mathsf{s}}(\mathrm{x}) \doteq 1$) entails that all costs in `cost` are minimal as per the first and only applicable disjunct in $\mathsf{Inv}(\gamma, \texttt{src})$. As such, the proof of `parallel_dijkstra` is immediate from the parallel rule (Par).

```
$\{\mathsf{Pre}(\theta,\gamma_0)\}$
1. void dijkstra(int[][] a,int[] c,int size,bitarray work,done){
2.  i=0;
3.  while(done != 2^size-1){b=<CAS(work[i],1,0)>;
    // apply $A^1_\theta$ if possible
4.   if(b){ $\left\{[\theta] * \boxed{\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \gamma^{\mathsf{r}}(\mathtt{i})=\bullet)}_I\right\}$
      cost = c[i];
   $\left\{[\theta] * \boxed{\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \gamma^{\mathsf{r}}(\mathtt{i})=\bullet \land \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i}))}_I\right\}$
5.    for(j=0;j<size;j++){
   $\left\{[\theta] * \boxed{\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.0^{\mathtt{size-j}} \land \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i}))}_I\right\}$
        newcost = cost + a[i][j]; b = 1;
   $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.0^{\mathtt{size-j}} \\ \land\, \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}=\mathtt{cost}+\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \mathtt{b}=1)\end{array}}_I\right\}$
6.      do{ oldcost=c[j];// apply $A^2_\theta$
   $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma,\mathrm{C}.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.\mathrm{C}.0^{\mathtt{size-j-1}} \\ \land\, \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}=\mathtt{cost}+\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \mathtt{b}=1 \land \mathtt{oldcost}=\mathrm{C})\end{array}}_I\right\}$
7.        if(newcost<oldcost){
   $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma,\mathrm{C}.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.\mathrm{C}.0^{\mathtt{size-j-1}} \\ \land\, \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}=\mathtt{cost}+\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \mathtt{oldcost}=\mathrm{C} \land \mathtt{newcost}<\mathtt{oldcost})\end{array}}_I\right\}$
8.          b=<CAS(work[j],1,0)>; // apply $A^3_\theta$ if possible
9.          if(b){
   $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma,\mathrm{C}.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.\mathrm{C}.0^{\mathtt{size-j-1}} \\ \land\, \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}=\mathtt{cost}+\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \mathtt{oldcost}=\mathrm{C} \land \mathtt{newcost}<\mathtt{oldcost} \\ \land\, \gamma^{\mathsf{s}}(\mathtt{j})=\theta \land \gamma^{\mathsf{r}}(\mathtt{j})=\circ)\end{array}}_I\right\}$
            b=<CAS(c[j],oldcost,newcost)>; // apply $A^4_\theta$ if possible
   $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma,\mathrm{C}.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}<\mathtt{oldcost} \\ \land\, ((\mathtt{b}=1 \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j+1}}.0^{\mathtt{size-j-1}}) \lor (\mathtt{b}=0 \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.\mathrm{C}.0^{\mathtt{size-j-1}})) \\ \land\, \gamma^{\mathsf{s}}(\mathtt{j})=\theta \land \gamma^{\mathsf{r}}(\mathtt{j})=\circ)\end{array}}_I\right\}$
            <work[j]=1>; } // apply $A^5_\theta$ or $A^6_\theta$ depending on the value of b
   $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0 \dot{\cong} \gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}<\mathtt{oldcost} \\ \land\, ((\mathtt{b}=1 \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j+1}}.0^{\mathtt{size-j-1}}) \lor (\mathtt{b}=0 \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.0^{\mathtt{size-j}})))\end{array}}_I\right\}$
```

Figure 10.8.: A proof sketch of dijkstra (continued in Fig. 10.9)

```
9.        else {
  $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma,\mathrm{C}.\ \mathsf{g}(\gamma) * (\gamma_0\dot{\cong}\gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.\mathrm{C}.0^{\mathtt{size\text{-}j\text{-}1}} \land \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i})\\ \land\,\mathtt{newcost}=\mathtt{cost}+\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \mathtt{oldcost}=\mathrm{C} \land \mathtt{newcost}<\mathtt{oldcost})\end{array}}_I\right\}$
          b=<CAS(done[j],1,0)>; // apply $A^3_\theta$ if possible
10.        if(b){
  $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma,\mathrm{C}.\ \mathsf{g}(\gamma) * (\gamma_0\dot{\cong}\gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.\mathrm{C}.0^{\mathtt{size\text{-}j\text{-}1}}\\ \land\,\mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}=\mathtt{cost}+\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \mathtt{oldcost}=\mathrm{C} \land \mathtt{newcost}<\mathtt{oldcost}\\ \land\,\gamma^{\mathsf{s}}(\mathtt{j})=\theta \land \gamma^{\mathsf{r}}(\mathtt{j})=\bullet)\end{array}}_I\right\}$
            b=<CAS(c[j],oldcost,newcost)>; // apply $A^4_\theta$ if possible
  $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma,\mathrm{C}.\ \mathsf{g}(\gamma) * (\gamma_0\dot{\cong}\gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i})\\ \land\,((\mathtt{b}=1 \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j+1}}.0^{\mathtt{size\text{-}j\text{-}1}}) \lor (\mathtt{b}=0 \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.\mathrm{C}.0^{\mathtt{size\text{-}j\text{-}1}}))\\ \land\,\mathtt{newcost}<\mathtt{oldcost} \land \mathtt{newcost}=\mathtt{cost}+\gamma^{\mathrm{E}}[\mathtt{i}][\mathtt{j}] \land \gamma^{\mathsf{s}}(\mathtt{j})=\theta \land \gamma^{\mathsf{r}}(\mathtt{j})=\bullet)\end{array}}_I\right\}$
11.          if(b){ <work[j]=1> } else { <done[j]=1> }
             // apply $A^5_\theta$ or $A^7_\theta$ depending on the value of b
  $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0\dot{\cong}\gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \mathtt{cost}=\gamma^{\mathsf{c}}(\mathtt{i}) \land \mathtt{newcost}<\mathtt{oldcost}\\ \land\,((\mathtt{b}=1 \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j+1}}.0^{\mathtt{size\text{-}j\text{-}1}}) \lor (\mathtt{b}=0 \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.0^{\mathtt{size\text{-}j}})))\end{array}}_I\right\}$
12.       }}}
  $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma.\mathsf{g}(\gamma) * (\gamma_0\dot{\cong}\gamma\land\mathsf{Inv}(\gamma,\mathtt{src})\land\gamma^{\mathsf{s}}(\mathtt{i})=\theta\\ ((\mathtt{newcost}<\mathtt{oldcost} \land \mathtt{b}=1 \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j+1}}.0^{\mathtt{size\text{-}j\text{-}1}})\\ \lor(\mathtt{newcost}<\mathtt{oldcost}\land\mathtt{b}=0\land\gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.0^{\mathtt{size\text{-}j}})\\ \lor(\mathtt{newcost}\geq\mathtt{oldcost}\land\mathtt{b}=1\land\gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.-.0^{\mathtt{size\text{-}j\text{-}1}}))\end{array}}_I\right\}$
  // apply $A^8_\theta$ on the third disjunct
  $\left\{[\theta] * \boxed{\begin{array}{l}\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0\dot{\cong}\gamma\land\mathsf{Inv}(\gamma,\mathtt{src})\land\gamma^{\mathsf{s}}(\mathtt{i})=\theta\\ ((\mathtt{b}=1\land\gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j+1}}.0^{\mathtt{size\text{-}j\text{-}1}})\lor(\mathtt{b}=0\land\gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j}}.0^{\mathtt{size\text{-}j}}))\end{array}}_I\right\}$
13.      } while(!b)
  $\left\{[\theta] * \boxed{\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0\dot{\cong}\gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \gamma^{\mathsf{s}}(\mathtt{i})=\theta \land \gamma^{\mathsf{r}}(\mathtt{i})=1^{\mathtt{j+1}}.0^{\mathtt{size\text{-}j\text{-}1}})}_I\right\}$
14.     } $\left\{[\theta] * \boxed{\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0\dot{\cong}\gamma\land\mathsf{Inv}(\gamma,\mathtt{src})\land\gamma^{\mathsf{s}}(\mathtt{i})=\theta\land\gamma^{\mathsf{r}}(\mathtt{i})=\bullet)}_I\right\}$
        <done[i]=1>; // apply $A^9_\theta$
  $\left\{[\theta] * \boxed{\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0\dot{\cong}\gamma\land\mathsf{Inv}(\gamma,\mathtt{src})\land\gamma^{\mathsf{s}}(\mathtt{i})=1)}_I\right\}$
15.   } i = (i+1) mod size;
16. } } $\left\{[\theta] * \boxed{\exists\gamma.\ \mathsf{g}(\gamma) * (\gamma_0\dot{\cong}\gamma \land \mathsf{Inv}(\gamma,\mathtt{src}) \land \forall x.\ \gamma^{\mathsf{s}}(x)=1)}_I\right\}$
$\left\{\mathsf{Post}(\theta,\gamma_0)\right\}$
```

Figure 10.9.: A proof sketch of `dijkstra` (continued from Fig. 10.8)

11. Conclusions

Throughout this thesis, we have considered numerous challenges concerning abstract library specification, library refinement and fine-grained concurrent reasoning. In doing so, we have explored numerous ideas, answered several questions and found new techniques and solutions.

As to abstract library specification, we have applied structural separation logic (SSL) [60] for abstract specification of several libraries and for local reasoning about their client programs. Most notably, we have used SSL to specify a fragment of the Document Object Model (DOM) Core Level 1 library [1], following the standard closely. We have demonstrated that our specification significantly improves over the existing DOM formalisms [24, 52] in that it is local, compositional and language-independent. We have generalised the theory of SSL from [60] to allow for a language-independent library specification and client reasoning. This way, our library specification can be used to reason about different client programs of the library written in different programming languages. We have demonstrated this by integrating SSL with several program logics based on separation logic (SL). Most notably we have integrated the SSL DOM specification with the SL-based JavaScript program logic of [21] and used it to verify several realistic JavaScript ad blocker programs.

As to library refinement, we have explored two existing approaches to library refinement for separation logic: the locality-breaking and locality-preserving refinements. We demonstrated that while the more popular locality-breaking approach is more suitable for the refinement of sequential libraries, the locality-preserving is better suited for the refinement of concurrent libraries as it simplifies the proof obligations. We have presented a JavaScript implementation of the DOM fragment formally specified in this thesis. We have established the correctness of our DOM implementation with respect to its formal specification by providing a locality-breaking refinement proof. Unlike existing formalism of DOM where the axiomatic DOM specification is justified against a high-level operational semantics [59, 60, 52, 24], we link our specification to an underlying implementation (in JavaScript) and demonstrate that our implementation satisfies the same specification. This allows us to obtain a stronger soundness result as our specification is justified against a realistic implementation rather than the semantics devised specifically to validate the axiomatic specification.

Concerning concurrent reasoning, we have introduced the *concurrent local subjective logic* (CoLoSL) for compositional reasoning about concurrent programs. We have introduced the notion of subjective views where we verify each thread with respect to its customised local view of the state. Subjective views may arbitrarily overlap with one another, and may expand or contract in accordance with the thread footprint. We have introduced the general *composition* and *framing* of interference relations (describing how the shared resources may be manipulated by each thread) in the spirit of resource composition and framing in standard separation logic. We have demonstrated that this fluidity allows for better proof reuse. We have used CoLoSL to reason about several nontrivial concurrent graph-manipulating algorithms, two of which had never been verified before.

Bibliography

[1] W3C DOM Core Level 1 Standard. `https://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html`.

[2] W3C DOM Standard. `https://www.w3.org/TR/#tr_DOM`.

[3] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge Tracts in Theoretical Computer Science 47. Cambridge University Press, 1998.

[4] Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. Local reasoning, separation and aliasing. In *SPACE*, volume 4, 2004.

[5] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables As Resource in Separation Logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, May 2006.

[6] Cristiano Calcagno, Thomas Dinsdale-Young, and Philippa Gardner. Adjunct Elimination in Context Logic for Trees. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, pages 255–270. Springer-Verlag, 2007.

[7] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context Logic and Tree Update. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–282. ACM, 2005.

[8] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, pages 366–378, 2007.

[9] Pedor da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular termination verification for non-blocking concurrency. In *Programming Languages and Systems: 25th European Symposium on Programming*, ESOP'16, pages 176–201, 2016.

[10] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, 2014.

[11] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[12] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. In *Language Hierarchies and Interfaces*, pages 43–56, 1975.

[13] Thomas Dinsdale-Young. *Abstract Data and Local Reasoning.* PhD thesis, Imperial College London, 2010.

[14] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 287–300. ACM, 2013.

[15] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, pages 504–528. Springer-Verlag, 2010.

[16] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning. In *Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments*, pages 199–215. Springer-Verlag, 2010.

[17] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *The Proceeding of the 7th Asian Symposium of Programming Languages and Systems*, pages 161–177. Springer Berlin Heidelberg, 2009.

[18] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*, pages 363–377, 2009.

[19] Xinyu Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.

[20] Ivana Filipović, Peter O'Hearn, Noah Torp-Smith, and Hongseok Yang. Blaming the Client: On Data Refinement in the Presence of Pointers. *Formal Aspects of Computing*, 22(5):547–583, September 2010.

[21] Philippa Gardner, Sergio Maffeis, and Gareth Smith. Towards a Program Logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–44. ACM, 2012.

[22] Philippa Gardner, Sergio Maffeis, and Gareth Smith. Towards a Program Logic for JavaScript. Technical report, Imperial College London, 2012.

[23] Philippa Gardner, Azalea Raad, Mark Wheelhouse, and Adam Wright. Abstract local reasoning for concurrent libraries: Mind the gap. *Electronic Notes in Theoretical Computer Science*, 308:147 – 166, 2014.

[24] Philippa Gardner, Gareth Smith, Mark Wheelhouse, and Uri Zarfaty. Local Hoare Reasoning about DOM. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 261–270. ACM, 2008.

[25] Philippa Gardner and Mark Wheelhouse. Small Specifications for Tree Update. In *Proceedings of the 6th International Conference on Web Services and Formal Methods*, pages 178–195. Springer-Verlag, 2010.

[26] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing (Second Ed.)*. Addison Wesley, 2003.

[27] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[28] Peter G. Hinman. *Fundamentals of Mathematical Logic*. A K Peters, Wellesley, Massachusetts, 2005.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[30] C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Inf.*, 1(4):271–281, December 1972.

[31] Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *POPL*, pages 523–536, 2013.

[32] Samin S. Ishtiaq and Peter W. O'Hearn. BI As an Assertion Language for Mutable Data Structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26. ACM, 2001.

[33] Jonas Braband Jensen and Lars Birkedal. Fictional separation logic. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 377–396, 2012.

[34] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[35] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 256–269, 2016.

[36] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'15, pages 637–650, 2015.

[37] Robert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'17, 2017.

[38] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, pages 290–310, 2014.

[39] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[40] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning About Programs that Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, pages 1–19. Springer-Verlag, 2001.

[41] Matthew Parkinson and Gavin Bierman. Separation Logic and Abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–258. ACM, 2005.

[42] Matthew J. Parkinson. *Local Reasoning for Java.* PhD thesis, University of Cambridge, 2005.

[43] Azalea Raad and Sophia Drossopoulou. A sip of the chalice. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*, FTfJP '11, pages 2:1–2:30. ACM, 2011.

[44] Azalea Raad, José Fragoso Santos, and Philippa Gardner. DOM: A JavaScript implementation. `http://www.soundandcomplete.org/DOM/implementation.zip`.

[45] Azalea Raad, José Fragoso Santos, and Philippa Gardner. Dom: Specification and client reasoning. In *Proceedings of the 14th Asian Symposium on Programming Languages and Systems*, APLAS '16, 2016.

[46] Azalea Raad, Aquinas Hobor, Jules Villard, and Philippa Gardner. Verifying concurrent graph algorithms. In *Proceedings of the 14th Asian Symposium on Programming Languages and Systems*, APLAS '16, 2016.

[47] Azalea Raad, Jules Villard, and Philippa Gardner. CoLoSL: Concurrent Local Subjective Logic. Technical report, 2014. `http://www.doc.ic.ac.uk/~azalea/ESOP2015/CoLoSL-TR.pdf`.

[48] Azalea Raad, Jules Villard, and Philippa Gardner. CoLoSL: Concurrent Local Subjective Logic. In *ESOP*, pages 710–735, 2015.

[49] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[50] John C. Reynolds. A short course on separation logic. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/wwwaac2003/notes7.ps`, 2003.

[51] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, 2015.

[52] Gareth D. Smith. *Local reasoning for Web Programs.* PhD thesis, Imperial College London, 2011.

[53] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, pages 149–168, 2014.

[54] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular reasoning about separation of concurrent data structures. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 169–188, 2013.

[55] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.

[56] Viktor Vafeiadis. Automatically proving linearizability. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 450–464, 2010.

[57] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

[58] Shengyi Wang, Qinxiang Cao, Asankhaya Sharma, and Aquinas Hobor. The ramifications of mechanized localizations within data structures. under submission, 2016.

[59] Mark J. Wheelhouse. *Segment Logic.* PhD thesis, Imperial College London, 2012.

[60] Adam D. Wright. *Structural Separation Logic.* PhD thesis, Imperial College London, 2013.

[61] Hongseok Yang. *Local Reasoning for Stateful Programs.* PhD thesis, University of Illinois, 2001.

A. DOM Specification

We proceed with a description of each of the DOM operations defined in Def. 61 and its axiomatisation in SSL.

A.1. Node Axioms

When `n` identifies a DOM node, then

- `r:=n.nodeName`: returns the name of `n` in `r`.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \gamma\right\}$$
$$\mathtt{r := n.nodeName}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \text{“}\#\mathrm{document}\text{”}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \gamma\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.nodeName}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{S}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.nodeName}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \text{“}\#\mathrm{text}\text{”}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.nodeName}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{S}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}}\right\}$$

- `r:=n.nodeValue`: returns the value of `n` in `r`.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \gamma\right\}$$
$$\mathtt{r := n.nodeValue}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \gamma\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.nodeValue}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.nodeValue}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{S}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}} * \mathsf{val}(\mathrm{T}, \mathrm{S}')\right\}$$
$$\mathtt{r := n.nodeValue}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{S}') * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$

- The `r:=n.nodeType`: returns the type of `n` in `r`. Each DOM node type is described as an integer value, with the text, element, attribute and document node types associated with integer values 3, 1, 2 and 9, respectively.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \gamma\right\}$$
$$\mathtt{r := n.nodeType}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : 9) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \gamma\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.nodeType}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : 1) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.nodeType}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : 3) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.nodeType}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : 2) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$

- `r:=n.parentNode`: returns the identifier of the parent node of `n` in `r` when it exists; it returns `null` if `n` is a document or attribute node (a document node is the top-most node and has no parent; an attribute node is associated with an element node, but is *not* the child of an element node), or if `n` resides in the grove.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \delta\right\}$$
$$\mathtt{r := n.parentNode}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{U}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \delta\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \epsilon_1 \otimes \mathrm{S}_{\mathrm{N}}[\gamma, \delta]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \epsilon_2]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.parentNode}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{U}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \epsilon_1 \otimes \mathrm{S}_{\mathrm{N}}[\gamma, \delta]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \epsilon_2]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.parentNode}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{U}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta_1 \oslash \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'}]_{\mathrm{F}} \oslash \beta_2\right\}$$
$$\mathtt{r := n.parentNode}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{U}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta \oslash \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'}]_{\mathrm{F}} \oslash \beta_2\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \gamma\right\}$$
$$\mathtt{r := n.parentNode}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \gamma\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.parentNode}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} \oplus \varnothing_g\right\}$$
$$\mathtt{r := n.parentNode}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} \oplus \varnothing_g\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \oplus \varnothing_g\right\}$$
$$\mathtt{r := n.parentNode}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \oplus \varnothing_g\right\}$$

- `r:=n.childNodes`: compiles a (live) NodeList containing the identifiers of the children of node `n` and returns the identifier of the NodeList in `r`.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}_1} \,\&\, \gamma * \mathsf{TIDs}(\mathrm{T}, \mathrm{L})\right\}$$
$$\mathtt{r := n.childNodes}$$
$$\left\{\exists \mathrm{F}, \mathrm{F}_2.\ \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{r}{:}\mathrm{F}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}_2} \,\&\, \gamma * \mathrm{F}_1 \dot{\subseteq} \mathrm{F}_2 * \mathrm{F} \dot{\in} \mathrm{F}_2\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}_1} * \mathsf{TIDs}(\mathrm{T}, \mathrm{L})\right\}$$
$$\mathtt{r := n.childNodes}$$
$$\left\{\exists \mathrm{F}, \mathrm{F}_2.\ \mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{F}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}_2} * \mathrm{F}_1 \dot{\subseteq} \mathrm{F}_2 * \mathrm{F} \dot{\in} \mathrm{F}_2\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}_1}\right\}$$
$$\mathtt{r := n.childNodes}$$
$$\left\{\exists \mathrm{F}, \mathrm{F}_2.\ \mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{F}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}_2} * \mathrm{F}_1 \dot{\subseteq} \mathrm{F}_2 * \mathrm{F} \dot{\in} \mathrm{F}_2\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}_1}\right\}$$
$$\mathtt{r := n.childNodes}$$
$$\left\{\exists \mathrm{F}, \mathrm{F}_2.\ \mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{F}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}_2} * \mathrm{F}_1 \dot{\subseteq} \mathrm{F}_2 * \mathrm{F} \dot{\in} \mathrm{F}_2\right\}$$

- `r:=n.firstChild`: returns the identifier of the first child of `n` in `r` when it exists; returns `null` if `n` has no children.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \delta\right\}$$
$$\mathtt{r := n.firstChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \delta\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \delta\right\}$$
$$\mathtt{r := n.firstChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \delta\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{S}_{\mathrm{M}}[\gamma, \delta]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \epsilon]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.firstChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{S}_{\mathrm{M}}[\gamma, \delta]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \epsilon]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'} \otimes \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.firstChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'} \otimes \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \varnothing_f]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.firstChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \varnothing_f]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.firstChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'} \oslash \beta]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.firstChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'} \oslash \beta]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\varnothing_{tf}]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.firstChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\varnothing_{tf}]_{\mathrm{F}}\right\}$$

- `r:=n.lastChild`: returns the identifier of the last child of `n` in `r` when it exists; returns `null` if `n` has no children.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \delta\right\}$$
$$\mathtt{r := n.lastChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \delta\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\varnothing_{e}]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \delta\right\}$$
$$\mathtt{r := n.lastChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\varnothing_{e}]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \delta\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma \otimes \mathrm{S}_{\mathrm{M}}[\delta, \epsilon]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.lastChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma \otimes \mathrm{S}_{\mathrm{M}}[\delta, \epsilon]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma \otimes \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.lastChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma \otimes \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \varnothing_{f}]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.lastChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \varnothing_{f}]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : -) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.lastChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \oslash \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}]_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.lastChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \oslash \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\varnothing_{tf}]_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.lastChild}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\varnothing_{tf}]_{\mathrm{F}}\right\}$$

- `r:=n.previousSibling`: returns the identifier of the previous sibling of `n` in `r` when it exists; it returns `null` if i) `n` is the first child of its parent; or ii) `n` identifies a document or attribute node; or iii) `n` resides in the grove.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{L}}[\mathrm{S}']_{\mathrm{F}'} \otimes \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{L}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{L}}[\mathrm{S}']_{\mathrm{F}'} \otimes \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{L}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{L}) * \alpha \mapsto \mathrm{S}_{\mathrm{L}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}'_{\mathrm{L}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \mathrm{S}_{\mathrm{N}}[\delta, \epsilon]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{L}) * \alpha \mapsto \mathrm{S}'_{\mathrm{L}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \mathrm{S}_{\mathrm{N}}[\delta, \epsilon]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{L}}[\mathrm{S}']_{\mathrm{F}'} \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{L}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{L}}[\mathrm{S}']_{\mathrm{F}'} \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}} \otimes \gamma]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$
$$\mathtt{r := n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}} \otimes \gamma]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}'_{\mathrm{U}}[\beta, \mathrm{S}_{\mathrm{N}}[\gamma, \delta]^{\mathrm{E}}_{\mathrm{F}} \otimes \epsilon]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$
$$\mathtt{r := n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}'_{\mathrm{U}}[\beta, \mathrm{S}_{\mathrm{N}}[\gamma, \delta]^{\mathrm{E}}_{\mathrm{F}} \otimes \epsilon]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}]^{\mathrm{E}'}_{\mathrm{F}'} \ \& \ \delta\right\}$$
$$\mathtt{r := n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}]^{\mathrm{E}'}_{\mathrm{F}'} \ \& \ \delta\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \ \& \ \gamma\right\}$$
$$\mathtt{r := n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \ \& \ \gamma\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} \oplus \varnothing_g\right\}$$
$$\mathtt{r := n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} \oplus \varnothing_g\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : -) * \alpha \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \oplus \varnothing_g\right\}$$
$$\mathtt{r := n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \oplus \varnothing_g\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathsf{text}_{\mathrm{L}}[\mathrm{S}']_{\mathrm{F}'} \oslash \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{L}) * \alpha \mapsto \#\mathsf{text}_{\mathrm{L}}[\mathrm{S}']_{\mathrm{F}'} \oslash \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\#\mathsf{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'} \oslash \beta]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.previousSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\#\mathsf{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'} \oslash \beta]_{\mathrm{F}}\right\}$$

- `r:=n.nextSibling`: returns the identifier of the next sibling of `n` in `r` when it exists; it returns `null` if i) `n` is the last child of its parent; or ii) `n` identifies a document or attribute node; or iii) `n` resides in the grove.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}} \otimes \mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$
$$\mathtt{r := n.nextSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}} \otimes \mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} \otimes \#\mathsf{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$
$$\mathtt{r := n.nextSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} \otimes \#\mathsf{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} \otimes \mathrm{S}'_{\mathrm{M}}[\delta, \epsilon]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$
$$\mathtt{r := n.nextSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} \otimes \mathrm{S}'_{\mathrm{M}}[\delta, \epsilon]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$

$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \otimes \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$
$$\mathtt{r} := \mathtt{n.nextSibling}$$
$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \otimes \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$

$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}}]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$
$$\mathtt{r} := \mathtt{n.nextSibling}$$
$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}}]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$

$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}'_{\mathrm{U}}[\beta, \gamma \otimes \mathrm{S}_{\mathrm{N}}[\delta, \epsilon]^{\mathrm{E}}_{\mathrm{F}}]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$
$$\mathtt{r} := \mathtt{n.nextSibling}$$
$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}'_{\mathrm{U}}[\beta, \gamma \otimes \mathrm{S}_{\mathrm{N}}[\delta, \epsilon]^{\mathrm{E}}_{\mathrm{F}}]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$

$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}_{\mathrm{N}}[\alpha, \beta]^{\mathrm{E}}_{\mathrm{F}}]^{\mathrm{E}'}_{\mathrm{F}'} \,\&\, \gamma\right\}$$
$$\mathtt{r} := \mathtt{n.nextSibling}$$
$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}_{\mathrm{N}}[\alpha, \beta]^{\mathrm{E}}_{\mathrm{F}}]^{\mathrm{E}'}_{\mathrm{F}'} \,\&\, \gamma\right\}$$

$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \gamma\right\}$$
$$\mathtt{r} := \mathtt{n.nextSibling}$$
$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \,\&\, \gamma\right\}$$

$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.nextSibling}$$
$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$

$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} \oplus \varnothing_{g}\right\}$$
$$\mathtt{r} := \mathtt{n.nextSibling}$$
$$\left\{\textsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} \oplus \varnothing_{g}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \oplus \varnothing_g\right\}$$
$$\mathtt{r:=n.nextSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathtt{null}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \oplus \varnothing_g\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \oslash \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$
$$\mathtt{r:=n.nextSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{M}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \oslash \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta \oslash \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'}]_{\mathrm{F}}\right\}$$
$$\mathtt{r:=n.nextSibling}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'} \oslash \beta]_{\mathrm{F}}\right\}$$

- `r:=n.ownerDocument`: returns in `r` the identifier of the document node with which `n` is associated. As we do not model document fragment nodes (lightweight document nodes in the grove), the result always corresponds to the document identifier d.

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]_{\mathrm{F}}^{\mathrm{E}} \mathbin{\&} \gamma\right\}$$
$$\mathtt{r:=n.ownerDocument}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]_{\mathrm{F}}^{\mathrm{E}} \mathbin{\&} \gamma\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta,\gamma]_{\mathrm{F}}^{\mathrm{E}}\right\}$$
$$\mathtt{r:=n.ownerDocument}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:d) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta,\gamma]_{\mathrm{F}}^{\mathrm{E}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r:=n.ownerDocument}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:d) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$

$$\mathtt{r} := \mathtt{n.ownerDocument}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : d) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$

When `u`, `n` and `o` identify DOM nodes, then

- `u.insertBefore(n,o)`: inserts `n` into the child list of `u` before the existing child `o`, and returns `n`. If `o` is null, `n` is appended to the end of the child list. It fails if i) `o` is not null and is not a child of `u`; or ii) the result of insertion does not correspond to a well-typed DOM node (e.g. when `n` is a document node); or iii) `n` is an ancestor of `u` (otherwise it would introduce a cycle and break the DOM structure).

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathrm{M}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \mathrm{S}''_{\mathrm{M}}[\delta, \epsilon]^{\mathrm{E2}}_{\mathrm{F2}} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \zeta \mapsto \mathrm{S}'_{\mathrm{N}}[\eta, \mathrm{T}]^{\mathrm{E1}}_{\mathrm{F1}} * \mathsf{complete}(\mathrm{T})\end{array}\right\}$$

$$\mathtt{r} := \mathtt{u.insertBefore(m,\ n)}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathrm{M}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) \\ * \, \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \mathrm{S}'_{\mathrm{N}}[\eta, \mathrm{T}]^{\mathrm{E1}}_{\mathrm{F1}} \otimes \mathrm{S}''_{\mathrm{M}}[\delta, \epsilon]^{\mathrm{E2}}_{\mathrm{F2}} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \zeta \mapsto (\varnothing_f \vee \varnothing_g)\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathrm{M}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \mathrm{S}'_{\mathrm{M}}[\delta, \epsilon]^{\mathrm{E2}}_{\mathrm{F2}} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \zeta \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}'']_{\mathrm{F1}}\end{array}\right\}$$

$$\mathtt{r} := \mathtt{u.insertBefore(m,\ n)}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathrm{M}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) \\ * \, \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}'']_{\mathrm{F1}} \otimes \mathrm{S}'_{\mathrm{M}}[\delta, \epsilon]^{\mathrm{E2}}_{\mathrm{F2}} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \zeta \mapsto (\varnothing_f \vee \varnothing_g)\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathrm{M}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathsf{text}_{\mathrm{M}}[\mathrm{S}'']_{\mathrm{F2}} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \mathrm{S}'_{\mathrm{N}}[\epsilon, \mathrm{T}]^{\mathrm{E1}}_{\mathrm{F1}} * \mathsf{complete}(\mathrm{T})\end{array}\right\}$$

$$\mathtt{r} := \mathtt{u.insertBefore(m,\ n)}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathrm{M}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) \\ * \, \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \mathrm{S}'_{\mathrm{N}}[\epsilon, \mathrm{T}]^{\mathrm{E1}}_{\mathrm{F1}} \otimes \#\mathsf{text}_{\mathrm{M}}[\mathrm{S}'']_{\mathrm{F2}} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto (\varnothing_f \vee \varnothing_g)\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathrm{M}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathsf{text}_{\mathrm{M}}[\mathrm{S}'']_{\mathrm{F2}} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F1}}\end{array}\right\}$$

$$\mathtt{r} := \mathtt{u.insertBefore(m,\ n)}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathrm{M}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) \\ * \, \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F1}} \otimes \#\mathsf{text}_{\mathrm{M}}[\mathrm{S}'']_{\mathrm{F2}} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto (\varnothing_f \vee \varnothing_g)\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathtt{null}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S}_\mathrm{U}[\beta, \gamma]^\mathrm{E}_\mathrm{F} * \zeta \mapsto \mathrm{S}'_\mathrm{N}[\eta, \mathrm{T}]^{\mathrm{E}'}_{\mathrm{F}'} * \mathsf{complete}(\mathrm{T})\end{array}\right\}$$

```
r := u.insertBefore(m, n)
```

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathtt{null}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) \\ * \, \alpha \mapsto \mathrm{S}_\mathrm{U}[\beta, \gamma \otimes \mathrm{S}'_\mathrm{N}[\eta, \mathrm{T}]^{\mathrm{E}'}_{\mathrm{F}'}]^\mathrm{E}_\mathrm{F} * \zeta \mapsto (\varnothing_f \vee \varnothing_g)\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathtt{null}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S}_\mathrm{U}[\beta, \gamma]^\mathrm{E}_\mathrm{F} * \zeta \mapsto \#\mathrm{text}_\mathrm{N}[\mathrm{S}']_{\mathrm{F}'}\end{array}\right\}$$

```
r := u.insertBefore(m, n)
```

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathtt{null}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) \\ * \, \alpha \mapsto \mathrm{S}_\mathrm{U}[\beta, \gamma \otimes \#\mathrm{text}_\mathrm{N}[\mathrm{S}']_{\mathrm{F}'}]^\mathrm{E}_\mathrm{F} * \zeta \mapsto (\varnothing_f \vee \varnothing_g)\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathrm{M}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S}_\mathrm{U}[\beta_1 \oslash \#\mathrm{text}_\mathrm{M}[\mathrm{S}'']_{\mathrm{F}_2} \oslash \beta_2]_\mathrm{F} * \delta \mapsto \#\mathrm{text}_\mathrm{N}[\mathrm{S}']_{\mathrm{F}_1}\end{array}\right\}$$

```
r := u.insertBefore(m, n)
```

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathrm{M}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) \\ * \, \alpha \mapsto \mathrm{S}_\mathrm{U}[\beta_1 \otimes \#\mathrm{text}_\mathrm{N}[\mathrm{S}']_{\mathrm{F}_1} \otimes \#\mathrm{text}_\mathrm{M}[\mathrm{S}'']_{\mathrm{F}_2} \otimes \beta_2]_\mathrm{F} * \delta \mapsto (\varnothing_f \vee \varnothing_g)\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathtt{null}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S}_\mathrm{U}[\beta]_\mathrm{F} * \zeta \mapsto \#\mathrm{text}_\mathrm{N}[\mathrm{S}']_{\mathrm{F}'}\end{array}\right\}$$

```
r := u.insertBefore(m, n)
```

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathtt{null}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) \\ * \, \alpha \mapsto \mathrm{S}_\mathrm{U}[\beta \otimes \#\mathrm{text}_\mathrm{N}[\mathrm{S}']_{\mathrm{F}'}]_\mathrm{F} * \zeta \mapsto (\varnothing_f \vee \varnothing_g)\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathtt{null}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \#\mathrm{doc}_\mathrm{U}[\varnothing_e]^\mathrm{E}_\mathrm{F} \,\&\, \beta * \gamma \mapsto \mathrm{S}_\mathrm{N}[\delta, \epsilon]^{\mathrm{E}1}_{\mathrm{F}1}\end{array}\right\}$$

```
r := u.insertBefore(m, n)
```

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{m} : \mathtt{null}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) \\ * \, \alpha \mapsto \#\mathrm{doc}_\mathrm{U}[\mathrm{S}_\mathrm{N}[\delta, \epsilon]^{\mathrm{E}1}_{\mathrm{F}1}]^\mathrm{E}_\mathrm{F} \,\&\, \beta * \gamma \mapsto (\varnothing_f \vee \varnothing_g)\end{array}\right\}$$

- `u.replaceChild(n,o)`: replaces `o` in the child list of `u` with `n`, and returns `o`. It fails if i) `o` is not a child of `u`; or ii) the result of replacement does not correspond to a well-typed DOM node (e.g. when `n` is a document node); or iii) `n` is an ancestor of `u` (otherwise it would introduce a cycle and break the DOM structure).

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{u}:\mathrm{U},\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{R}) \\ *\,\alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \mathrm{S}''_{\mathrm{O}}[\delta,\epsilon]^{\mathrm{E}2}_{\mathrm{F}2} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} \\ *\,\zeta \mapsto \mathrm{S}'_{\mathrm{N}}[\theta,\mathrm{T}]^{\mathrm{E}1}_{\mathrm{F}1} * \mathsf{complete}(\mathrm{T}) * \mu \mapsto \varnothing_g \end{array}\right\}$$

$$\mathtt{r := u.replaceChild(n,\ o)}$$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{u}:\mathrm{U},\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{O}) \\ *\,\alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \mathrm{S}'_{\mathrm{N}}[\theta,\mathrm{T}]^{\mathrm{E}1}_{\mathrm{F}1} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \zeta \mapsto (\varnothing_f \vee \varnothing_g) * \mu \mapsto \mathrm{S}''_{\mathrm{O}}[\delta,\epsilon]^{\mathrm{E}2}_{\mathrm{F}2} \end{array}\right\}$$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{u}:\mathrm{U},\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{R}) \\ *\,\alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \mathrm{S}'_{\mathrm{O}}[\delta,\epsilon]^{\mathrm{E}2}_{\mathrm{F}2} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \zeta \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}'']_{\mathrm{F}1} * \eta \mapsto \varnothing_g \end{array}\right\}$$

$$\mathtt{r := u.replaceChild(n,\ o)}$$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{u}:\mathrm{U},\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{O}) \\ *\,\alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}'']_{\mathrm{F}1} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \zeta \mapsto (\varnothing_f \vee \varnothing_g) * \eta \mapsto \mathrm{S}'_{\mathrm{O}}[\delta,\epsilon]^{\mathrm{E}2}_{\mathrm{F}2} \end{array}\right\}$$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{u}:\mathrm{U},\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{R}) \\ *\,\alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathsf{text}_{\mathrm{O}}[\mathrm{S}'']_{\mathrm{F}2} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} \\ *\ \delta \mapsto \mathrm{S}'_{\mathrm{N}}[\epsilon,\mathrm{T}]^{\mathrm{E}1}_{\mathrm{F}1} * \mathsf{complete}(\mathrm{T}) * \zeta \mapsto \varnothing_g \end{array}\right\}$$

$$\mathtt{r := u.replaceChild(n,\ o)}$$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{u}:\mathrm{U},\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{O}) \\ *\,\alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \mathrm{S}'_{\mathrm{N}}[\epsilon,\mathrm{T}]^{\mathrm{E}1}_{\mathrm{F}1} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto (\varnothing_f \vee \varnothing_g) * \zeta \mapsto \#\mathsf{text}_{\mathrm{O}}[\mathrm{S}'']_{\mathrm{F}2} \end{array}\right\}$$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{u}:\mathrm{U},\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{R}) \\ *\,\alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathsf{text}_{\mathrm{O}}[\mathrm{S}'']_{\mathrm{F}2} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}1} * \epsilon \mapsto \varnothing_g \end{array}\right\}$$

$$\mathtt{r := u.replaceChild(n,\ o)}$$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{u}:\mathrm{U},\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{O}) \\ *\,\alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}1} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto (\varnothing_f \vee \varnothing_g) * \epsilon \mapsto \#\mathsf{text}_{\mathrm{O}}[\mathrm{S}'']_{\mathrm{F}2} \end{array}\right\}$$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{u}:\mathrm{U},\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{R}) \\ *\,\alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta_1 \oslash \#\mathsf{text}_{\mathrm{O}}[\mathrm{S}'']_{\mathrm{F}2} \oslash \beta_2]_{\mathrm{F}} * \delta \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}1} * \epsilon \mapsto \varnothing_g \end{array}\right\}$$

$$\mathtt{r := u.replaceChild(n,\ o)}$$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{u}:\mathrm{U},\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{r}:\mathrm{O}) \\ *\,\alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta_1 \oslash \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}1} \oslash \beta_2]_{\mathrm{F}} * \delta \mapsto (\varnothing_f \vee \varnothing_g) * \epsilon \mapsto \#\mathsf{text}_{\mathrm{O}}[\mathrm{S}'']_{\mathrm{F}2} \end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{R}) \\ * \mu \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}'_{\mathrm{O}}[\alpha, \beta]^{\mathrm{E2}}_{\mathrm{F2}}]^{\mathrm{E}}_{\mathrm{F}} \ \& \ \eta * \gamma \mapsto \mathrm{S}_{\mathrm{N}}[\delta, \epsilon]^{\mathrm{E1}}_{\mathrm{F1}} * \zeta \mapsto \varnothing_g\end{array}\right\}$$

$$\mathtt{r := u.replaceChild(n,\ o)}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{O}) \\ * \mu \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}_{\mathrm{N}}[\delta, \epsilon]^{\mathrm{E1}}_{\mathrm{F1}}]^{\mathrm{E}}_{\mathrm{F}} \ \& \ \eta * \gamma \mapsto (\varnothing_f \vee \varnothing_g) * \zeta \mapsto \mathrm{S}'_{\mathrm{O}}[\alpha, \beta]^{\mathrm{E2}}_{\mathrm{F2}}\end{array}\right\}$$

- `u.removeChild(o)`: removes `o` from the child list of `u`, moves `o` to the document grove (DOM nodes are never deleted; orphaned nodes are added to the grove) and then returns `o`. It fails if `o` is not a child of `u`.

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \mathrm{S}'_{\mathrm{O}}[\delta, \epsilon]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \zeta \mapsto \varnothing_g\right\}$$

$$\mathtt{r := u.removeChild(o)}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{O}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \zeta \mapsto \mathrm{S}'_{\mathrm{O}}[\delta, \epsilon]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \varnothing_g\right\}$$

$$\mathtt{r := u.removeChild(o)}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{O}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta, \gamma_1 \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta_1 \oslash \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'} \oslash \beta_2]_{\mathrm{F}} * \gamma \mapsto \varnothing_g\right\}$$

$$\mathtt{r := u.removeChild(o)}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{O}) * \alpha \mapsto \mathrm{S}_{\mathrm{U}}[\beta_1 \oslash \beta_2]_{\mathrm{F}} * \gamma \mapsto \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}_{\mathrm{O}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \ \& \ * \delta \mapsto \varnothing_g\right\}$$

$$\mathtt{r := u.removeChild(o)}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{o} : \mathrm{O}, \mathtt{r} : \mathrm{O}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}} \ \& \ * \delta \mapsto \mathrm{S}_{\mathrm{O}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$

- `u.appendChild(n)`: appends `n` to the end of `u`'s child list and returns `n`. It fails if i) the result of appending does not correspond to a well-typed DOM node (e.g. when `n` is a document node); or ii) `n` is an ancestor of `u` (otherwise it would introduce a cycle and break the

DOM structure).

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S_U}[\beta, \gamma]^{\mathrm{E1}}_{\mathrm{F1}} * \delta \mapsto \mathrm{S}'_{\mathrm{N}}[\epsilon, \mathrm{T}]^{\mathrm{E2}}_{\mathrm{F2}} * \mathsf{complete}(\mathrm{T})\right\}$$

$$\mathtt{r := u.appendChild(n)}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) * \alpha \mapsto \mathrm{S_U}[\beta, \gamma \otimes \mathrm{S}'_{\mathrm{N}}[\epsilon, \mathrm{T}]^{\mathrm{E2}}_{\mathrm{F2}}]^{\mathrm{E1}}_{\mathrm{F1}} * \delta \mapsto (\varnothing_f \vee \varnothing_g)\right\}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S_U}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$

$$\mathtt{r := u.appendChild(n)}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) * \alpha \mapsto \mathrm{S_U}[\beta, \gamma \otimes \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto (\varnothing_f \vee \varnothing_g)\right\}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S_U}[\beta]_{\mathrm{F}} * \delta \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$

$$\mathtt{r := u.appendChild(n)}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) * \alpha \mapsto \mathrm{S_U}[\beta \oslash \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'}]_{\mathrm{F}} * \delta \mapsto (\varnothing_f \vee \varnothing_g)\right\}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \beta * \gamma \mapsto \mathrm{S_N}[\delta, \epsilon]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$

$$\mathtt{r := u.appendChild(n)}$$

$$\left\{\mathsf{vars}(\mathtt{u} : \mathrm{U}, \mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{N}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S_N}[\delta, \epsilon]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \beta * \gamma \mapsto (\varnothing_f \vee \varnothing_g)\right\}$$

- `r:=n.hasChildNodes`: returns a boolean value in `r` denoting whether `n` has any children.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S_M}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \delta\right\}$$

$$\mathtt{r := n.hasChildNodes()}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathsf{true}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S_M}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \delta\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \beta\right\}$$

$$\mathtt{r := n.hasChildNodes()}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathsf{false}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \beta\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : -) * \alpha \mapsto \mathrm{S_N}[\beta, \epsilon_1 \otimes \mathrm{S_M}[\gamma, \delta]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \epsilon_2]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\mathtt{r := n.hasChildNodes()}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathsf{true}) * \alpha \mapsto \mathrm{S_N}[\beta, \epsilon_1 \otimes \mathrm{S_M}[\gamma, \delta]^{\mathrm{E}'}_{\mathrm{F}'} \otimes \epsilon_2]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma_1 \otimes \#\mathsf{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.hasChildNodes()}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathsf{true}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma_1 \otimes \#\mathsf{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \varnothing_f]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.hasChildNodes()}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathsf{false}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \varnothing_f]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.hasChildNodes()}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathsf{false}) * \alpha \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta_1 \oslash \#\mathsf{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'} \oslash \beta_2]_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.hasChildNodes()}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathsf{true}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta_1 \oslash \#\mathsf{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'} \oslash \beta_2]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\varnothing_{tf}]_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.hasChildNodes()}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathsf{false}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\varnothing_{tf}]_{\mathrm{F}}\right\}$$

A.2. Text Node Axioms

When `n` identifies a text node, then

- `r:=n.data`: returns the value (text contents) of `n` in `r`.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r} := \mathtt{n.data}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{S}) * \alpha \mapsto \#\mathsf{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$

- `r:=n.length`: returns the length of the value (text contents) of `n` in `r` (i.e. returns a non-negative value corresponding to the number of characters in the value of `n`).

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{r:=n.length}$$
$$\left\{\exists \mathrm{L}.\ \mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{L}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} * \mathrm{L} \doteq |\mathrm{S}|\right\}$$

- `r:=n.substringData(o,c)`: when `o` and `c` hold integer values and the value of `n` is denoted by string s, the substring of s beginning at offset `o` (indexed from 0) and continuing for `c` characters is returned in `r`. If the sum of `o` and `c` exceeds the length of s, then all characters to the end of s are returned. This operation fails if `o` is an invalid offset (i.e. negative or greater than the length of s), or if `c` is negative. For instance, when the value of `n` is “lorem”, `o`=1 and `c`=3, then `r:=n.substringData(o,c)` yields `r`=“ore”. On the other hand, when `c`=7, then `r`=“orem”.

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{c}:\mathrm{C},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1.\mathrm{S}_2.\mathrm{S}_3]_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{S}_1| * \mathrm{C} \doteq |\mathrm{S}_2|\right\}$$
$$\mathtt{r:=n.substringData(o,\ c)}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{c}:\mathrm{C},\mathtt{r}:\mathrm{S}_2) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1.\mathrm{S}_2.\mathrm{S}_3]_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{c}:\mathrm{C},\mathtt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1.\mathrm{S}_2]_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{S}_1| * \mathrm{C} \dot{\geq} |\mathrm{S}_2|\right\}$$
$$\mathtt{r:=n.substringData(o,\ c)}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{o}:\mathrm{O},\mathtt{c}:\mathrm{C},\mathtt{r}:\mathrm{S}_2) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1.\mathrm{S}_2]_{\mathrm{F}}\right\}$$

- `n.appendData(s)`: when `s` holds a string, then `s` is appended to the end of the value of `n`.

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}') * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}}\right\}$$
$$\mathtt{n.appendData(s)}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}') * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}.\mathrm{S}']_{\mathrm{F}}\right\}$$

- `n.insertData(o,s)`: when `o` holds an integer value and `s` holds a string, then `s` is inserted into the text contents (value) of `n` at offset `o` (indexed from 0). This operation fails if `o` is an invalid offset (i.e. negative or greater than the length of the value of `n`). For instance, when the value of `n` is “lorem”, `o`=1 and `s`=“ipsum”, then `n.insertData(o,s)` updates the value of `n` to “lipsumorem”.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{o} : \mathrm{O}, \mathtt{s} : \mathrm{S}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{s}_1.\mathrm{s}_2]_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{s}_1|\right\}$$

`n.insertData(o, s)`

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{o} : \mathrm{O}, \mathtt{s} : \mathrm{S}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{s}_1.\mathrm{S}.\mathrm{s}_2]_{\mathrm{F}}\right\}$$

- `n.deleteData(o,c)`: when `o` and `c` hold integer values and the value of `n` is denoted by string s, the substring of s beginning at offset `o` (indexed from 0) and continuing for `c` characters is removed from s. If the sum of `o` and `c` exceeds the length of s, then all characters to the end of s are removed. This operation fails if `o` is an invalid offset (i.e. negative or greater than the length of s), or if `c` is negative. For instance, when the value of `n` is “lorem”, `o`=1 and `c`=3, then `n.deleteData(o,c)` updates the value of `n` to “lm”. On the other hand, when `c`=7, the value of `n` to is updated to “l”.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{o} : \mathrm{O}, \mathtt{c} : \mathrm{C}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{s}.\mathrm{s}'.\mathrm{s}'']_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{s}| * \mathrm{C} \doteq |\mathrm{s}'|\right\}$$

`n.deleteData(o, c)`

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{o} : \mathrm{O}, \mathtt{c} : \mathrm{C}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{s}.\mathrm{s}'']_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{o} : \mathrm{O}, \mathtt{c} : \mathrm{C}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{s}.\mathrm{s}']_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{s}| * \mathrm{C} \dot{\geq} |\mathrm{s}'|\right\}$$

`n.deleteData(o, c)`

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{o} : \mathrm{O}, \mathtt{c} : \mathrm{C}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{s}]_{\mathrm{F}}\right\}$$

- `n.replaceData(o,c,s)`: when `o` and `c` hold integer values, `s` holds a DOM string, and the value of `n` is denoted by string s, the substring of s beginning at offset `o` (indexed from 0) and continuing for `c` characters is replaced by s. If the sum of `o` and `c` exceeds the length of s, then all characters to the end of s are replaced. This operation fails if `o` is an invalid offset (i.e. negative or greater than the length of s), or if `c` is negative. For instance, when the value of `n` is “lorem”, `o`=1, c=3 and `s`=“ipsum”, then `n.replaceData(o,c,s)` updates the value of `n` to “lipsumm”. On the other hand, when `c`=7, the value of `n` to is updated to “lipsum”.

$$\left\{\mathsf{vars}(\texttt{n}:\mathrm{N}, \texttt{o}:\mathrm{O}, \texttt{c}:\mathrm{C}, \texttt{s}:\mathrm{S}_2) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}.\mathrm{S}_1.\mathrm{S}']_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{S}| * \mathrm{C} \doteq |\mathrm{S}_1|\right\}$$
$$\texttt{n.replaceData(o, c, s)}$$
$$\left\{\mathsf{vars}(\texttt{n}:\mathrm{N}, \texttt{o}:\mathrm{O}, \texttt{c}:\mathrm{C}, \texttt{s}:\mathrm{S}_2) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}.\mathrm{S}_2.\mathrm{S}']_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\texttt{n}:\mathrm{N}, \texttt{o}:\mathrm{O}, \texttt{c}:\mathrm{C}, \texttt{s}:\mathrm{S}'') * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}.\mathrm{S}']_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{S}| * \mathrm{C} \dot{\geq} |\mathrm{S}'|\right\}$$
$$\texttt{n.replaceData(o, c, s)}$$
$$\left\{\mathsf{vars}(\texttt{n}:\mathrm{N}, \texttt{o}:\mathrm{O}, \texttt{c}:\mathrm{C}, \texttt{s}:\mathrm{S}'') * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}.\mathrm{S}'']_{\mathrm{F}}\right\}$$

- `r:=n.splitText(o)`: when `o` holds an integer value, the text contents (value) of `n` are split into two text nodes at offset `o` (indexed from 0), keeping both in the DOM tree as siblings. The identifier of the new text node is returned in `r`. This operation fails if `o` is an invalid offset (i.e. negative or greater than the length of `n`’s value).

$$\left\{\mathsf{vars}(\texttt{n}:\mathrm{N}, \texttt{o}:\mathrm{O}, \texttt{r}:\mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}.\mathrm{S}']_{\mathrm{F}} * \mathrm{O} \doteq |\mathrm{S}|\right\}$$
$$\texttt{r:=n.splitText(o)}$$
$$\left\{\exists \mathrm{M}, \mathrm{F}'.\ \mathsf{vars}(\texttt{n}:\mathrm{N}, \texttt{o}:\mathrm{O}, \texttt{r}:\mathrm{M}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}} \otimes \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}']_{\mathrm{F}'}\right\}$$

A.3. Element Node Axioms

When `n` identifies an element node, then

- `r:=n.tagName`: returns the (tag) name of `n` in `r`.

$$\left\{\mathsf{vars}(\texttt{n}:\mathrm{N}, \texttt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\texttt{r:=n.tagName}$$
$$\left\{\mathsf{vars}(\texttt{n}:\mathrm{N}, \texttt{r}:\mathrm{S}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

- `r:=n.getAttribute(s)`: when `s` holds a string, the attributes of `n` are inspected and the value of the attribute named `s` is returned in `r` if such an attribute exists. If `n` has no attribute named `s` then the empty string is returned.

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}_\mathrm{N}[\beta \odot \mathrm{S}'_\mathrm{M}[\mathrm{T}]_{\mathrm{F}'},\gamma]^\mathrm{E}_\mathrm{F} * \mathsf{val}(\mathrm{T},\mathrm{S}'')\right\}$$
$$\mathtt{r := n.getAttribute(s)}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{r}:\mathrm{S}'') * \alpha \mapsto \mathrm{S}_\mathrm{N}[\beta \odot \mathrm{S}'_\mathrm{M}[\mathrm{T}]_{\mathrm{F}'},\gamma]^\mathrm{E}_\mathrm{F}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{r}:\mathrm{R}) * \alpha \mapsto \mathrm{S}_\mathrm{N}[\mathrm{A},\gamma]^\mathrm{E}_\mathrm{F} * \mathsf{out}(\mathrm{A},\mathrm{S}')\right\}$$
$$\mathtt{r := n.getAttribute(s)}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{r}:\text{“”}) * \alpha \mapsto \mathrm{S}_\mathrm{N}[\mathrm{A},\gamma]^\mathrm{E}_\mathrm{F}\right\}$$

- `n.setAttribute(s,v)`: when `s` and `v` hold strings, the attributes of `n` are inspected and the value of the attribute named `s` is set to `v` if such an attribute exists. If `n` has no attribute named `s` then the attribute set of `n` is extended with a new attribute with name `s` and value `v`.

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{v}:\mathrm{S}'') * \alpha \mapsto \mathrm{S}_\mathrm{N}[\beta \odot \mathrm{S}'_\mathrm{M}[\mathrm{T}]_{\mathrm{F}'},\gamma]^\mathrm{E}_\mathrm{F} * \delta \mapsto \varnothing_g * \mathsf{grove}(\mathrm{T},\mathrm{G})\right\}$$
$$\mathtt{n.setAttribute(s,\ v)}$$
$$\left\{\begin{array}{l}\exists \mathrm{R},\mathrm{F}''.\ \mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{v}:\mathrm{S}'')\\ * \alpha \mapsto \mathrm{S}_\mathrm{N}[\beta \odot \mathrm{S}'_\mathrm{M}[\#\mathrm{text}_\mathrm{R}[\mathrm{S}'']_{\mathrm{F}''}]_{\mathrm{F}'},\gamma]^\mathrm{E}_\mathrm{F} * \delta \mapsto \mathrm{G}\end{array}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{v}:\mathrm{S}'') * \alpha \mapsto \mathrm{S}_\mathrm{N}[\mathrm{A},\gamma]^\mathrm{E}_\mathrm{F} * \mathsf{out}(\mathrm{A},\mathrm{S}') * \mathsf{safe}(\mathrm{S}')\right\}$$
$$\mathtt{n.setAttribute(s,\ v)}$$
$$\left\{\begin{array}{l}\exists \mathrm{M},\mathrm{F}',\mathrm{R},\mathrm{F}''.\ \mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}',\mathtt{v}:\mathrm{S}'')\\ * \alpha \mapsto \mathrm{S}_\mathrm{N}[\mathrm{A} \odot \mathrm{S}'_\mathrm{M}[\#\mathrm{text}_\mathrm{R}[\mathrm{S}'']_{\mathrm{F}''}]_{\mathrm{F}'},\gamma]^\mathrm{E}_\mathrm{F}\end{array}\right\}$$

- `n.removeAttribute(s)`: when `s` holds a string, the attribute named `s` is removed from the attribute set of `n` when such an attribute exists. If `n` has no attribute named `s` then `n` remains unchanged.

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}') * \alpha \mapsto \mathrm{S}_\mathrm{N}[\beta \odot \mathrm{S}'_\mathrm{M}[\mathrm{T}]_{\mathrm{F}'},\gamma]^\mathrm{E}_\mathrm{F} * \delta \mapsto \varnothing_g\right\}$$
$$\mathtt{n.removeAttribute(s)}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S}') * \alpha \mapsto \mathrm{S}_\mathrm{N}[\beta,\gamma]^\mathrm{E}_\mathrm{F} * \delta \mapsto \mathrm{S}'_\mathrm{M}[\mathrm{T}]_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}') * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \gamma]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{out}(\mathrm{A}, \mathrm{S}')\right\}$$
$$\texttt{n.removeAttribute(s)}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}') * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

- `r:=n.getAttributeNode(s)`: when `s` holds a string, the attributes of `n` are inspected and the identifier of the attribute named `s` is returned in `r` if such an attribute exists. If `n` has no attribute named `s` then `null` is returned.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}', \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{S}'_{\mathrm{M}}[\mathrm{T}]_{\mathrm{F}'}, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$
$$\texttt{r := n.getAttributeNode(s)}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}', \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{S}'_{\mathrm{M}}[\mathrm{T}]_{\mathrm{F}'}, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}', \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \gamma]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{out}(\mathrm{A}, \mathrm{S}')\right\}$$
$$\texttt{r := n.getAttributeNode(s)}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}', \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \gamma]^{\mathrm{E}}_{\mathrm{F}}\right\}$$

- `r:=n.setAttributeNode(a)`: when `a` identifies a DOM attribute node named s, the attributes of `n` are inspected and the attribute named s is replaced with `a` if such an attribute exists. If `n` has no attribute named s then the attribute set of `n` is extended with `a`.

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{a} : \mathrm{M}, \mathrm{R} : \mathrm{R}) \\ * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{S}'_{\mathrm{P}}[\mathrm{T1}]_{\mathrm{F1}}, \gamma]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \mathrm{S}'_{\mathrm{M}}[\mathrm{T2}]_{\mathrm{F2}} \oplus \varnothing_g * \epsilon \mapsto \varnothing_g\end{array}\right\}$$
$$\texttt{r := n.setAttributeNode(a)}$$
$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{a} : \mathrm{M}, \mathrm{R} : \mathrm{P}) \\ * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta \odot \mathrm{S}'_{\mathrm{M}}[\mathrm{T2}]_{\mathrm{F2}}, \gamma]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \varnothing_g * \epsilon \mapsto \mathrm{S}'_{\mathrm{P}}[\mathrm{T1}]_{\mathrm{F1}}\end{array}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{a}{:}\mathrm{M}, \mathrm{R}{:}\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \gamma]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \mathrm{S}'_{\mathrm{M}}[\mathrm{T}]_{\mathrm{F}'} \oplus \varnothing_g * \mathsf{out}(\mathrm{A}, \mathrm{M})\right\}$$
$$\texttt{r := n.setAttributeNode(a)}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{a} : \mathrm{M}, \mathrm{R} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A} \odot \mathrm{S}'_{\mathrm{M}}[\mathrm{T}]_{\mathrm{F}'}, \gamma]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \varnothing_g\right\}$$

- `r:=n.removeAttributeNode(a)`: when `a` identifies an attribute node in the attribute set of `n`, `a` is removed from the attribute set of `n` and is returned in `r`. This operation fails if `a` is not in the attribute set of `n`.

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{a}:\mathrm{M},\mathtt{r}:\mathrm{R})*\alpha\mapsto \mathrm{S}_{\mathrm{N}}[\beta\odot \mathrm{S}'_{\mathrm{M}}[\mathrm{T}]_{\mathrm{F}'},\gamma]^{\mathrm{E}}_{\mathrm{F}}*\delta\mapsto\varnothing_g\right\}$$
$$\mathtt{r := n.removeAttributeNode(a)}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{a}:\mathrm{M},\mathtt{r}:\mathrm{M})*\alpha\mapsto \mathrm{S}_{\mathrm{N}}[\beta,\gamma]^{\mathrm{E}}_{\mathrm{F}}*\delta\mapsto \mathrm{S}'_{\mathrm{M}}[\mathrm{T}]_{\mathrm{F}'}\right\}$$

- `r:=n.getElementsByTagName(s)`: when `s` holds a string, it searches the child list of `n` (using depth-first, left-to-right search), compiles a NodeList containing the identifiers of those element nodes whose names match `s` and returns the identifier of this NodeList in `r`.

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:\mathrm{R})*\alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\beta,\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}}*\mathsf{srch}(\mathrm{T},\mathrm{S},\mathrm{L})\right\}$$
$$\mathtt{r := n.getElementsByTagName(s)}$$
$$\left\{\exists \mathrm{E}',\mathrm{F}'.\ \mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:\mathrm{F}')*\alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\beta,\mathrm{T}]^{\mathrm{E}'}_{\mathrm{F}}*\mathrm{E}\dot{\subseteq}\mathrm{E}'*(\mathrm{S},\mathrm{F}')\dot{\in}\mathrm{E}'\right\}$$

A.4. Attribute Node Axioms

When `n` identifies an attribute node, then

- `r:=n.name`: returns the name of `n` in `r`.

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R})*\alpha\mapsto \mathrm{S}_{\mathrm{N}}[\beta]_{\mathrm{F}}\right\}$$
$$\mathtt{r := n.name}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{S})*\alpha\mapsto \mathrm{S}_{\mathrm{N}}[\beta]_{\mathrm{F}}\right\}$$

- `r:=n.value`: returns the value of `n` in `r`.

$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{R})*\alpha\mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}*\mathsf{val}(\mathrm{T},\mathrm{S}')\right\}$$
$$\mathtt{r := n.value}$$
$$\left\{\mathsf{vars}(\mathtt{n}:\mathrm{N},\mathtt{r}:\mathrm{S}')*\alpha\mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}}\right\}$$

A.5. Document Node Axioms

When `n` identifies a document node, then

- `r:=n.documentElement`: returns the identifier of the document element in `r` when it exists; otherwise `null` is returned.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \delta\right\}$$
$$\mathtt{r := n.documentElement}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{S}_{\mathrm{M}}[\beta, \gamma]^{\mathrm{E}'}_{\mathrm{F}'}]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \delta\right\}$$

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \delta\right\}$$
$$\mathtt{r := n.documentElement}$$
$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \delta\right\}$$

- `r:=n.createElement(s)`: when `s` holds a *safe* DOM string, the DOM grove is extended with a new element node named `s`. The identifier of the new element node is returned in `r`. A DOM string is safe if it does not contain the invalid '#' character. The new element has no attributes and no children. This operation fails if `s` holds an unsafe string (one containing the '#' character).

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \gamma * \mathsf{safe}(\mathrm{S})\right\}$$
$$\mathtt{r := n.createElement(s)}$$
$$\left\{\exists \mathrm{M}, \mathrm{F}', \mathrm{E}'.\ \mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : \mathrm{M}) * \ \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \gamma \oplus \mathrm{S}_{\mathrm{M}}[\varnothing_a, \varnothing_f]^{\mathrm{E}'}_{\mathrm{F}'}\right\}$$

- `r:=n.createTextNode(s)`: when `s` holds a string, the DOM grove is extended with a new text node with value `s`. The identifier of the new text node is returned in `r`.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \gamma\right\}$$
$$\mathtt{r := n.createTextNode(s)}$$
$$\left\{\exists \mathrm{M}, \mathrm{F}'.\ \mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \mathbin{\&} \gamma \oplus \#\mathrm{text}_{\mathrm{M}}[\mathrm{S}]_{\mathrm{F}'}\right\}$$

- `r:=n.createAttribute(s)`: when `s` holds a *safe* string, the DOM grove is extended with a new attribute node named `s`. The identifier of the new attribute node is returned in `r`. This operation fails if `s` holds an unsafe string (one containing the '#' character).

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \gamma * \mathsf{safe}(\mathrm{S})\right\}$$
$$\mathtt{r := n.createAttribute(s)}$$
$$\left\{\exists \mathrm{M}, \mathrm{F}'.\ \mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\beta]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \gamma \oplus \mathrm{S}_{\mathrm{M}}[\varnothing_{tf}]_{\mathrm{F}'}\right\}$$

- `r:=n.getElementsByTagName(s)`: behaves the same way as the element node operation with the same name.

$$\left\{\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}} \;\&\; \beta * \mathsf{srch}(\mathrm{T}, \mathrm{S}, \mathrm{L})\right\}$$
$$\mathtt{r := n.getElementsByTagName(s)}$$
$$\left\{\exists \mathrm{E}', \mathrm{F}'.\ \mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : \mathrm{F}') * \alpha \mapsto \#\mathrm{doc}_{\mathrm{N}}[\mathrm{T}]^{\mathrm{E}'}_{\mathrm{F}} \;\&\; \beta * \mathrm{E} \dot{\subseteq} \mathrm{E} * (\mathrm{S}, \mathrm{F}') \dot{\in} \mathrm{E}'\right\}$$

A.6. NodeList Axioms

When `f` identifies a NodeList, then

- `r:=f.length()`: returns the length of `f` in `r`.

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} \;\&\; \delta * \mathsf{TIDs}(\mathrm{T}, \mathrm{L}) * \mathrm{F} \dot{\in} \mathrm{F}'\right\}$$
$$\mathtt{r := f.length()}$$
$$\left\{\exists \mathrm{M}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} \;\&\; \delta * \mathrm{M} \doteq |\mathrm{L}|\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathsf{TIDs}(\mathrm{T}, \mathrm{L}) * \mathrm{F} \dot{\in} \mathrm{F}'\right\}$$
$$\mathtt{r := f.length()}$$
$$\left\{\exists \mathrm{M}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathrm{M} \doteq |\mathrm{L}|\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}'} * \mathrm{F} \dot{\in} \mathrm{F}'\right\}$$
$$\mathtt{r := f.length()}$$
$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : 0) * \alpha \mapsto \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{T}]_{\mathrm{F}'} * \mathsf{TIDs}(\mathrm{T}, \mathrm{L}) * \mathrm{F} \dot{\in} \mathrm{F}'\right\}$$
$$\mathtt{r := f.length()}$$
$$\left\{\exists \mathrm{M}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{S}']_{\mathrm{F}'} * \mathrm{M} \doteq |\mathrm{L}|\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} \mathbin{\&} \gamma * \mathsf{srch}(\mathrm{T}, \mathrm{S}, \mathrm{L}) * (\mathrm{S}, \mathrm{F}) \in \mathrm{E}\right\}$$

$$\mathtt{r := f.length()}$$

$$\left\{\exists \mathrm{M}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} \mathbin{\&} \gamma * \mathrm{M} \doteq |\mathrm{L}|\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f}{:}\mathrm{F}, \mathtt{r}{:}\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathsf{srch}(\mathrm{T}, \mathrm{S}', \mathrm{L}) * (\mathrm{S}', \mathrm{F}) \dot{\in} \mathrm{E}\right\}$$

$$\mathtt{r := f.length()}$$

$$\left\{\exists \mathrm{M}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathrm{M} \doteq |\mathrm{L}|\right\}$$

- `r:=f.item(i)`: when `i` holds an integer, the `i`th item of `f` (indexed from 0) is returned in `r`. If `i` holds an out-of-bounds value (i.e. negative or greater than or equal to the length of `f`) then `null` is returned.

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) \\ * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E1}}_{\mathrm{F1}}]^{\mathrm{E2}}_{\mathrm{F2}} \mathbin{\&} \delta * \mathrm{F} \dot{\in} \mathrm{F}_2 * \mathrm{I} \doteq 0\end{array}\right\}$$

$$\mathtt{r := f.item(i)}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{N}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E1}}_{\mathrm{F1}}]^{\mathrm{E2}}_{\mathrm{F2}} \mathbin{\&} \delta\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\beta]^{\mathrm{E}}_{\mathrm{F}'} \mathbin{\&} \gamma * \mathrm{F} \dot{\in} \mathrm{F}' * \mathrm{I} \dot{\neq} 0\right\}$$

$$\mathtt{r := f.item(i)}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\beta]^{\mathrm{E}}_{\mathrm{F}'} \mathbin{\&} \gamma\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}'} \mathbin{\&} \gamma * \mathrm{F} \dot{\in} \mathrm{F}'\right\}$$

$$\mathtt{r := f.item(i)}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\varnothing_e]^{\mathrm{E}}_{\mathrm{F}'} \mathbin{\&} \gamma\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \mathrm{I} \dot{<} 0 * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\alpha, \beta]^{\mathrm{E}}_{\mathrm{F}'} * \mathrm{F} \dot{\in} \mathrm{F}'\right\}$$

$$\mathtt{r := f.item(i)}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\alpha, \beta]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S_N}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathsf{TIDs}(\mathrm{T}, \mathrm{L}) * \mathrm{I} \dot{\geq} |\mathrm{L}| * \mathrm{F} \dot{\in} \mathrm{F}'\end{array}\right\}$$
$$\mathtt{r := f.item(i)}$$
$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S_N}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S_N}[\beta, \mathrm{T} \otimes \gamma]^{\mathrm{E}}_{\mathrm{F}'} * \mathsf{TIDs}(\mathrm{T}, \mathrm{L}) * 0 \dot{\leq} \mathrm{I} \dot{<} |\mathrm{L}| * \mathrm{F} \dot{\in} \mathrm{F}'\end{array}\right\}$$
$$\mathtt{r := f.item(i)}$$
$$\left\{\exists \mathrm{M}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S_N}[\beta, \mathrm{T} \otimes \gamma]^{\mathrm{E}}_{\mathrm{F}'} * \mathrm{M} \dot{=} |\mathrm{L}|^{\mathrm{I}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{text_N}[\mathrm{S}]_{\mathrm{F}'} * \mathrm{F} \dot{\in} \mathrm{F}'\right\}$$
$$\mathtt{r := f.item(i)}$$
$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \ \alpha \mapsto \#\mathrm{text_N}[\mathrm{S}]_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \mathrm{I} \dot{<} 0 * \alpha \mapsto \mathrm{S_N}[\alpha]_{\mathrm{F}'} * \mathrm{F} \dot{\in} \mathrm{F}'\right\}$$
$$\mathtt{r := f.item(i)}$$
$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S_N}[\alpha]_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S_N}[\mathrm{T}]_{\mathrm{F}'} * \mathsf{TIDs}(\mathrm{T}, \mathrm{L}) * \mathrm{I} \dot{\geq} |\mathrm{L}| * \mathrm{F} \dot{\in} \mathrm{F}'\right\}$$
$$\mathtt{r := f.item(i)}$$
$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S_N}[\mathrm{T}]_{\mathrm{F}'}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S_N}[\mathrm{T} \oslash \beta]_{\mathrm{F}'} * \mathsf{TIDs}(\mathrm{T}, \mathrm{L}) * 0 \dot{\leq} \mathrm{I} \dot{<} |\mathrm{L}| * \mathrm{F} \dot{\in} \mathrm{F}'\end{array}\right\}$$
$$\mathtt{r := f.item(i)}$$
$$\left\{\exists \mathrm{M}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S_N}[\mathrm{T} \oslash \beta]_{\mathrm{F}'} * \mathrm{M} \dot{=} |\mathrm{L}|^{\mathrm{I}}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \#\mathrm{doc_U}[\beta]^{\mathrm{E}}_{\mathrm{F}'} \ \& \ \gamma * (\mathrm{S}, \mathrm{F}) \dot{\in} \mathrm{E} * \mathrm{I} \dot{<} 0\right\}$$
$$\mathtt{r := f.item(i)}$$
$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc_U}[\beta]^{\mathrm{E}}_{\mathrm{F}'} \ \& \ \gamma\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} \, \& \, \beta * \mathsf{srch}(\mathrm{T}, \mathrm{S}, \mathrm{L}) * (\mathrm{S}, \mathrm{F}) \dot{\in} \mathrm{E} * 0 \dot{\leq} \mathrm{I} \dot{<} |\mathrm{L}|\end{array}\right\}$$

$$\mathtt{r := f.item(i)}$$

$$\left\{\exists \mathrm{M}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} \, \& \, \beta * \mathrm{M} \dot{=} |\mathrm{L}|^{\mathrm{I}}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} \, \& \, \beta * \mathsf{srch}(\mathrm{T}, \mathrm{S}, \mathrm{L}) * (\mathrm{S}, \mathrm{F}) \dot{\in} \mathrm{E} * \mathrm{I} \dot{\geq} |\mathrm{L}|\end{array}\right\}$$

$$\mathtt{r := f.item(i)}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \#\mathrm{doc}_{\mathrm{U}}[\mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} \, \& \, \beta\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathsf{srch}(\mathrm{T}, \mathrm{S}', \mathrm{L}) * (\mathrm{S}', \mathrm{F}) \dot{\in} \mathrm{E} * 0 \dot{\leq} \mathrm{I} \dot{<} |\mathrm{L}|\end{array}\right\}$$

$$\mathtt{r := f.item(i)}$$

$$\left\{\exists \mathrm{M}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{M}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathrm{M} \dot{=} |\mathrm{L}|^{\mathrm{I}}\right\}$$

$$\left\{\begin{array}{l}\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) \\ * \, \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'} * \mathsf{srch}(\mathrm{T}, \mathrm{S}', \mathrm{L}) * (\mathrm{S}', \mathrm{F}) \dot{\in} \mathrm{E} * \mathrm{I} \dot{\geq} |\mathrm{L}|\end{array}\right\}$$

$$\mathtt{r := f.item(i)}$$

$$\left\{\exists \mathrm{M}.\ \mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \mathrm{T}]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}'} * (\mathrm{S}', \mathrm{F}) \dot{\in} \mathrm{E} * \mathrm{I} \dot{<} 0\right\}$$

$$\mathtt{r := f.item(i)}$$

$$\left\{\mathsf{vars}(\mathtt{f} : \mathrm{F}, \mathtt{i} : \mathrm{I}, \mathtt{r} : \mathtt{null}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma]^{\mathrm{E}}_{\mathrm{F}'}\right\}$$

B. DOM Implementation Correctness

Definition 128 (JS assertion transformation)**.** Given the implementation function $[\![.]\!]$ (Def. 87), and the JSLogic logical expressions JSLExp (Def. 74), the JSLogic *expression transformation function*, $\llfloor . \rrfloor_e : \text{JSLExp} \rightarrow \text{JSLExp}$, is defined over the structure of JSLogic expressions as follows:

$$\llfloor w \rrfloor_e \triangleq w \qquad \llfloor L \rrfloor_e \triangleq L \qquad \llfloor S \rrfloor_e \triangleq S \qquad \llfloor \mathsf{e} \rrfloor_e \triangleq [\![\mathsf{e}]\!] \qquad \llfloor \varnothing \rrfloor_e \triangleq \varnothing$$

$$\llfloor \mathrm{X} \rrfloor_e \triangleq \mathrm{X} \qquad \llfloor \mathbf{l} \rrfloor_e \triangleq \mathbf{l} \qquad \llfloor E_1 \ominus E_2 \rrfloor_e \triangleq \llfloor E_1 \rrfloor_e \ominus \llfloor E_2 \rrfloor_e$$

$$\llfloor E_1 \bullet E_2 \rrfloor_e \triangleq \llfloor E_1 \rrfloor_e \bullet \llfloor E_2 \rrfloor_e \qquad \llfloor E_1 : E_2 \rrfloor_e \triangleq \llfloor E_1 \rrfloor_e : \llfloor E_2 \rrfloor_e$$

$$\llfloor E_1 \oplus E_2 \rrfloor_e \triangleq \llfloor E_1 \rrfloor_e \oplus \llfloor E_2 \rrfloor_e \qquad \llfloor E_1 . E_2 \rrfloor_e \triangleq \llfloor E_1 \rrfloor_e . \llfloor E_2 \rrfloor_e$$

$$\llfloor \lambda E_1 . E_2 \rrfloor_e \triangleq \lambda \llfloor E_1 \rrfloor_e . \llfloor E_2 \rrfloor_e$$

Given the JSLogic assertions JSAst (Def. 75), the JSLogic *assertion transformation function*, $\llfloor . \rrfloor_a : \text{JSAst} \rightarrow \text{JSAst}$, is defined over the structure of JSLogic assertions as follows:

$$\llfloor \mathsf{false} \rrfloor_a \triangleq \mathsf{false} \qquad \llfloor P \Rightarrow Q \rrfloor_a \triangleq \llfloor P \rrfloor_a \Rightarrow \llfloor Q \rrfloor_a \qquad \llfloor \exists \mathrm{X}.\ P \rrfloor_a \triangleq \exists \mathrm{X}.\ \llfloor P \rrfloor_a$$

$$\llfloor E_1 \ominus E_2 \rrfloor_a \triangleq \llfloor E_1 \rrfloor_e \ominus \llfloor E_2 \rrfloor_e \qquad \llfloor \mathsf{emp} \rrfloor_a \triangleq \mathsf{emp}$$

$$\llfloor (E_1, E_2) \mapsto E \rrfloor_a \triangleq (\llfloor E_1 \rrfloor_e , \llfloor E_2 \rrfloor_e) \mapsto \llfloor E \rrfloor_e \qquad \llfloor P * Q \rrfloor_a \triangleq \llfloor P \rrfloor_a * \llfloor Q \rrfloor_a$$

$$\llfloor P \mathrel{-\!\!*} Q \rrfloor_a \triangleq \llfloor P \rrfloor_a \mathrel{-\!\!*} \llfloor Q \rrfloor_a \qquad \llfloor P \uplus Q \rrfloor_a \triangleq \llfloor P \rrfloor_a \uplus \llfloor Q \rrfloor_a$$

Lemma 17 (JS expression translation)**.** *Given the expression transformation function* $\llfloor . \rrfloor_e$ *(Def. 128), the* JSLogic *expressions* Exp*, the* JSLogic *evaluation environments* Env *and the* JSLogic *evaluation function* $(\!|.|\!)^{(.)}$ *(Def. 74), for all* $E \in \text{Exp}$ *and* $\epsilon \in \text{Env}$*:*

$$(\!| \llfloor E \rrfloor_e |\!)^{\epsilon} = (\!| E |\!)^{\epsilon} \;\vee\; \exists \mathsf{x}, \mathsf{e}.\ (\!| E |\!)^{\epsilon} = \lambda \mathsf{x}.\mathsf{e} \wedge (\!| \llfloor E \rrfloor_e |\!)^{\epsilon} = \lambda \mathsf{x}.\, [\![\mathsf{e}]\!]$$

Proof. By induction on the structure of JSLogic expressions. The full proof is straightforward and is omitted here. Informally, the proof of all base cases but $\lambda E_1.E_2$ follows trivially from the definition of $\llbracket . \rrbracket_e$ (i.e. the $\llbracket . \rrbracket_e$ behaves as the identity function in these cases). The proof of inductive cases follow from the inductive hypotheses and the definition of the evaluation function. The proof of $\lambda E_1.E_2$ follows from the definitions of $(\!|.|\!)^\epsilon$ and $\llbracket . \rrbracket_e$.

Lemma 18 (JS assertion translation)**.** *Given the translation function* $\llbracket . \rrbracket$ *(Def. 89), the interface functions* $\mathfrak{I}$ *(Def. 80), the* JSLogic *evaluation environments* Env *(Def. 74), and the* JSLogic *assertions* JSAst *(Def. 75), for all* $\epsilon \in$ Env, $I \in \mathfrak{I}$ *and* $P \in$ JSAst*:*

$$\llbracket |P|_\epsilon \rrbracket^I = |\llbracket P \rrbracket_a|_\epsilon$$

Proof. By induction on the structure of JSLogic assertions.
Case false

$$\llbracket |\mathsf{false}|_\epsilon \rrbracket^I = \llbracket \emptyset \rrbracket^I = \emptyset = |\mathsf{false}|_\epsilon = |\llbracket \mathsf{false} \rrbracket_a|_\epsilon$$

Case emp

$$\llbracket |\mathsf{emp}|_\epsilon \rrbracket^I = \llbracket \{\mathbf{0}\} \rrbracket^I = \{\mathbf{0}\} = |\mathsf{emp}|_\epsilon = |\llbracket \mathsf{emp} \rrbracket_a|_\epsilon$$

where $\mathbf{0}$ denotes the empty JavaScript heap.

Case $(E_1, E_2) \mapsto E$

$$\begin{array}{rl}
& \llbracket |(E_1, E_2) \mapsto E|_\epsilon \rrbracket^I \\
(\llbracket . \rrbracket \text{ Def. }) & = \left\{ [(l, \mathsf{x}) \mapsto w] \;\middle|\; \begin{array}{l} (\!|E_1|\!)^\epsilon = l \land (\!|E_2|\!)^\epsilon = \mathsf{x} \land (\!|E|\!)^\epsilon = v \\ \land \big((\exists \mathsf{y}, \mathsf{e}.\ v = \lambda \mathsf{y}.\mathsf{e} \land w = \lambda \mathsf{y}.\ \llbracket \mathsf{e} \rrbracket) \lor v = w \big) \end{array} \right\} \\
(\text{Lemma17}) & = \left\{ [(l, \mathsf{x}) \mapsto w] \;\middle|\; (\!|\llbracket E_1 \rrbracket_e|\!)^\epsilon = l \land (\!|\llbracket E_2 \rrbracket_e|\!)^\epsilon = \mathsf{x} \land (\!|\llbracket E \rrbracket_e|\!)^\epsilon = w \right\} \\
(\llbracket . \rrbracket_e \text{ Def. }) & = |(\llbracket E_1 \rrbracket_e, \llbracket E_2 \rrbracket_e) \mapsto \llbracket E \rrbracket_e|_\epsilon \\
(\llbracket . \rrbracket \text{ Def. }) & = |\llbracket (E_1, E_2) \mapsto E \rrbracket_a|_\epsilon
\end{array}$$

Case $E_1 \ominus E_2$
The proof of this case is analogous and omitted here.

Case $\exists \textsc{x}.\ P$
Let $\epsilon{=}(\Gamma, L)$, then we have:

$$\begin{aligned}
& \llbracket |\exists \textsc{x}.\ P|_\epsilon \rrbracket^I \\
& = \left\llbracket \bigcup_{v \in \text{JSLV}\textsc{al}_{\mathbb{DOM}}} |P|_{([\Gamma|\textsc{x}\mapsto v], L)} \right\rrbracket^I \\
& = \bigcup_{v \in \text{JSLV}\textsc{al}_{\mathbb{DOM}}} \left\llbracket |P|_{([\Gamma|\textsc{x}\mapsto v], L)} \right\rrbracket^I \\
\text{(I.H.)} \quad & = \bigcup_{v \in \text{JSLV}\textsc{al}_{\mathbb{DOM}}} |\llbracket P \rrbracket_a|_{([\Gamma|\textsc{x}\mapsto v], L)} \\
& = |\exists \textsc{x}.\ \llbracket P \rrbracket_a|_\epsilon \\
(\llbracket . \rrbracket \text{ Def. }) \quad & = |\llbracket \exists \textsc{x}.\ P \rrbracket_a|_\epsilon
\end{aligned}$$

Case $P * Q$

$$\begin{aligned}
& \llbracket |P * Q|_\epsilon \rrbracket^I \\
(\llbracket . \rrbracket, |.| \text{ Def. }) \quad & = \left\{ h_1 \circ h_2 \,\middle|\, \begin{array}{r} \exists h_1' \in |P|_\epsilon .\ \exists h_2' \in |Q|_\epsilon . \\ h_1{=}\langle\!\langle h_1' \rangle\!\rangle_{\text{JS}} \wedge h_2{=}\langle\!\langle h_2' \rangle\!\rangle_{\text{JS}} \end{array} \right\} \\
& = \left\{ h_1 \,\middle|\, \begin{array}{r} \exists h_1' \in |P|_\epsilon . \\ h_1{=}\langle\!\langle h_1' \rangle\!\rangle_{\text{JS}} \end{array} \right\} * \left\{ h_2 \,\middle|\, \begin{array}{r} \exists h_2' \in |Q|_\epsilon . \\ h_2{=}\langle\!\langle h_2' \rangle\!\rangle_{\text{JS}} \end{array} \right\} \\
(\llbracket . \rrbracket, |.| \text{ Def. }) \quad & = \llbracket |P|_\epsilon \rrbracket^I * \llbracket |Q|_\epsilon \rrbracket^I \\
\text{(I.H.)} \quad & = |\llbracket P \rrbracket_a|_\epsilon * |\llbracket Q \rrbracket_a|_\epsilon \\
(\llbracket . \rrbracket \text{ Def. }) \quad & = |\llbracket P * Q \rrbracket_a|_\epsilon
\end{aligned}$$

Case $P \dagger Q$ where $\dagger \in \{ -\!\!*, \uplus\!\!\!\!*\,, \Rightarrow \}$
The proof of these cases are analogous to that of $P * Q$ and are omitted here.

Theorem 5 (Correct refinement (with proof)). *For all* $P, Q \in \text{JSA}\textsc{st}_{\mathbb{DOM}}$

(Def. 76) and $\mathtt{C} \in \mathrm{JSOP}_{\mathbb{DOM}}$ *(Def. 70):*

$$\{P\}\ \mathtt{C}\ \{Q\} \implies \tau : \{P\}\ \mathtt{C}\ \{Q\}$$

Proof. By induction on the structure of triples $\{P\}\ \mathtt{C}\ \{Q\}$. In each case we assume, as the inductive hypothesis, that the translated premises of each rule are sound. We show how to derive a proof of translated conclusions from these translated premises.

Case $\mathtt{e} \in \mathrm{OP}_{\mathbb{DOM}}$
This follows immediately from Lemma 23.

Case (Definition)

$$\dfrac{\dfrac{\dfrac{}{\tau : \{P\}\ \mathtt{e}\ \{Q\}}\ \text{(I.H.)}}{\forall I,\epsilon,r.\{\llbracket |P|_\epsilon * r\rrbracket^I\}\ \llbracket \mathtt{e}\rrbracket\ \{\llbracket |Q|_\epsilon * r\rrbracket^I\}}\ \tau\text{: Def.} \qquad \dfrac{\dfrac{}{\mathbf{r} \notin \mathsf{fv}(Q)}}{\mathbf{r} \notin \mathsf{fv}(|Q|_\epsilon)}}{\dfrac{\forall I,\epsilon,r.\ \{\llbracket |P|_\epsilon * r\rrbracket^I\}\ \mathtt{var}\ \llbracket \mathtt{e}\rrbracket\ \{\llbracket |Q|_\epsilon * r\rrbracket^I * |\mathbf{r}\doteq\mathtt{undefined}|_\epsilon\}}{\dfrac{\forall I,\epsilon,r.\ \{\llbracket |P|_\epsilon * r\rrbracket^I\}\ \llbracket \mathtt{var\ e}\rrbracket\ \{\llbracket |Q * \mathbf{r}\doteq\mathtt{undefined}|_\epsilon * r\rrbracket^I\}}{\tau : \{P\}\ \mathtt{var\ e}\ \{Q * \mathbf{r}\doteq\mathtt{undefined}\}}\ \tau\text{: Def.}}\ \llbracket.\rrbracket,\llbracket.\rrbracket\ \text{Def.}}\ \text{(Definition)}$$

Case (Value)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall\epsilon.\ \{|\mathsf{emp}|_\epsilon\}\ \mathtt{v}\ \{|\mathbf{r}\doteq\mathtt{v}|_\epsilon\}}\ \text{(Value)}}{\forall I,\epsilon,r.\ \{|\mathsf{emp}|_\epsilon * \llbracket r\rrbracket^I\}\ \mathtt{v}\ \{|\mathbf{r}\doteq\mathtt{v}|_\epsilon * \llbracket r\rrbracket^I\}}\ \text{(Frame)}}{\forall I,\epsilon,r.\ \{\llbracket |\mathsf{emp}|_\epsilon * r\rrbracket^I\}\ \llbracket \mathtt{v}\rrbracket\ \{\llbracket |\mathbf{r}\doteq\mathtt{v}|_\epsilon * r\rrbracket^I\}}\ \llbracket.\rrbracket,\llbracket.\rrbracket\ \text{Def.}}{\tau : \{\mathsf{emp}\}\ \mathtt{v}\ \{\mathbf{r}\doteq\mathtt{v}\}}\ \tau\text{: Def.}$$

Case (Variable)
Let $P \triangleq \sigma(Ls_1, \mathbf{l}, \mathrm{x}, L) \uplus \gamma(Ls_2, L.\mathrm{x}, V)$. Given an evaluation environment ϵ, an interface function I, and a set of states r, from Lemma 18 and the definition of $\llbracket . \rrbracket^I$ we have $\llbracket |P|_\epsilon * r\rrbracket^I = |P'|_\epsilon * \llbracket r\rrbracket^I$ and $\llbracket |P * \mathbf{r}\doteq L.\mathrm{x}|_\epsilon * r\rrbracket^I = |P' * \mathbf{r}\doteq \llbracket L\rrbracket_e .\mathrm{x}|_\epsilon * \llbracket r\rrbracket^I$, where

$$P' \triangleq \sigma(\llbracket Ls_1\rrbracket_e, \mathbf{l}, \mathrm{x}, \llbracket L\rrbracket_e) \uplus \gamma(\llbracket Ls_2\rrbracket_e, \llbracket L\rrbracket_e .\mathrm{x}, \llbracket V\rrbracket_e)$$

We then have:

$$
\begin{array}{c}
\hline
\forall\epsilon.\ \{|P'|_\epsilon\}\ \mathtt{x}\ \{|P' * \mathbf{r}\doteq \llbracket L\rrbracket_e .\mathtt{x}|_\epsilon\} \quad \text{(Variable)}\\
\hline
\forall I,\epsilon,r.\ \{|P'|_\epsilon * \llbracket r\rrbracket^I\}\ \mathtt{x}\ \{|P' * \mathbf{r}\doteq \llbracket L\rrbracket_e .\mathtt{x}|_\epsilon * \llbracket r\rrbracket^I\} \quad \text{(Frame)}\\
\hline\hline
\forall I,\epsilon,r.\ \{\llbracket |P|_\epsilon * r\rrbracket^I\}\ [\![\mathtt{x}]\!]\ \{\llbracket |P * \mathbf{r}\doteq L.\mathtt{x}|_\epsilon * r\rrbracket^I\} \quad \llbracket.\rrbracket, [\![.]\!]\ \text{Def.}\\
\hline\hline
\tau : \{P\}\ \mathtt{x}\ \{P * \mathbf{r}\doteq L.\mathtt{x}\} \quad \tau\text{: Def.}
\end{array}
$$

Case (Variable Null)
Let $P \triangleq \sigma(Ls, \mathbf{l}, \mathtt{x}, \mathtt{null})$. Given an evaluation environment ϵ, an interface function I, and a set of states r, from Lemma 18 and the definition of $\llbracket.\rrbracket^I$ we have $\llbracket |P|_\epsilon * r\rrbracket^I = |P'|_\epsilon * \llbracket r\rrbracket^I$ and $\llbracket |P * \mathbf{r}\doteq\mathtt{null.x}|_\epsilon * r\rrbracket^I = |P' * \mathbf{r}\doteq\mathtt{null.x}|_\epsilon * \llbracket r\rrbracket^I$, where $P' = \sigma(\llbracket Ls\rrbracket_e, \mathbf{l}, \mathtt{x}, \mathtt{null})$.
We then have:

$$
\begin{array}{c}
\hline
\forall\epsilon.\ \{|P'|_\epsilon\}\ \mathtt{x}\ \{|P' * \mathbf{r}\doteq\mathtt{null.x}|_\epsilon\} \quad \text{(Variable Null)}\\
\hline
\forall I,\epsilon,r.\ \{|P'|_\epsilon * \llbracket r\rrbracket^I\}\ \mathtt{x}\ \{|P' * \mathbf{r}\doteq\mathtt{null.x}|_\epsilon * \llbracket r\rrbracket^I\} \quad \text{(Frame)}\\
\hline\hline
\forall I,\epsilon,r.\ \{\llbracket |P|_\epsilon * r\rrbracket^I\}\ [\![\mathtt{x}]\!]\ \{\llbracket |P * \mathbf{r}\doteq\mathtt{null.x}|_\epsilon * r\rrbracket^I\} \quad \llbracket.\rrbracket, [\![.]\!]\ \text{Def.}\\
\hline\hline
\tau : \{P\}\ \mathtt{x}\ \{P * \mathbf{r}\doteq\mathtt{null.x}\} \quad \tau\text{: Def.}
\end{array}
$$

Case (Member Access)
Let $Q \triangleq R * \gamma(Ls, V, L) * L \dot{\neq} \mathtt{null}$. Given an evaluation ϵ, an interface function I, and a set of states r, from Lemma 18 and the definition of $\llbracket.\rrbracket$ we have

$$
\begin{aligned}
&\llbracket |Q|_\epsilon * r\rrbracket^I = \llbracket |R|_\epsilon * r\rrbracket^I * |S|_\epsilon\\
&\llbracket |Q * \mathbf{r}\doteq L.\mathtt{x}|_\epsilon\rrbracket^I = \llbracket |R|_\epsilon * r\rrbracket^I * |S * \mathbf{r}\doteq \llbracket L\rrbracket_e .\mathtt{x}|_\epsilon\\
&\llbracket |Q * \mathbf{r}\doteq V|_\epsilon\rrbracket^I = \llbracket |R|_\epsilon * r\rrbracket^I * |S * \mathbf{r}\doteq \llbracket V\rrbracket_e|_\epsilon
\end{aligned}
$$

where $S \triangleq \gamma(\llbracket Ls\rrbracket_e, \llbracket V\rrbracket_e, \llbracket L\rrbracket_e) * \llbracket L\rrbracket_e \dot{\neq} \mathtt{null}$.

We then have:

$$
\begin{array}{cl}
\hline
\tau : \{P\}\ \mathtt{e}\ \{Q * \mathbf{r}\dot{=}V\} & \text{(I.H.)}\\
\hline\hline
\forall I,\epsilon,r.\ \{\llfloor |P|_\epsilon * r \rrfloor^I\}\ \llbracket \mathtt{e} \rrbracket\ \{\llfloor |Q * \mathbf{r}\dot{=}V|_\epsilon * r \rrfloor^I\} & \tau\text{: Def.}\\
\hline\hline
\forall I,\epsilon,r. \{\llfloor |P|_\epsilon * r \rrfloor^I\}\ \llbracket \mathtt{e} \rrbracket\ \{\llfloor |R|_\epsilon * r \rrfloor^I * |S * \mathbf{r}\dot{=} \llfloor V \rrfloor_e|_\epsilon\} & \text{Def. of } \llfloor . \rrfloor\\
\hline
\begin{array}{c}
\forall I,\epsilon,r. \qquad \{\llfloor |P|_\epsilon * r \rrfloor^I\}\\
\llbracket \mathtt{e} \rrbracket\,\mathtt{.x}\\
\left\{\llfloor |R|_\epsilon * r \rrfloor^I * |S * \mathbf{r}\dot{=} \llfloor L \rrfloor_e .\mathbf{x}|_\epsilon\right\}
\end{array} & \text{(Member Access)}\\
\hline\hline
\forall I,\epsilon,r. \{\llfloor |P|_\epsilon * r \rrfloor^I\}\ \llbracket \mathtt{e.x} \rrbracket\ \{\llfloor |Q * \mathbf{r}\dot{=}L.\mathbf{x}|_\epsilon * r \rrfloor^I\} & \llfloor . \rrfloor, \llbracket . \rrbracket\ \text{Def.}\\
\hline\hline
\tau : \{P\}\ \mathtt{e.x}\ \{Q * \mathbf{r}\dot{=}L.\mathbf{x}\} & \tau\text{: Def.}
\end{array}
$$

Case (Computed Access)
Let $Q \triangleq S_2 * \gamma(Ls_2, V_2, X)$ and $R \triangleq S_1 * \gamma(Ls_1, V_1, L) * L \dot{\neq} \mathtt{null}$. Given an evaluation environment ϵ, an interface function I, and a set of states r, from Lemma 18 and the definition of $\llfloor . \rrfloor$ we have $\llfloor |R|_\epsilon * r \rrfloor^I = \llfloor |S_1|_\epsilon * r \rrfloor^I * |T_1|_\epsilon$ and $\llfloor |R * \mathbf{r}\dot{=}V_1|_\epsilon * r \rrfloor^I = \llfloor |S_1|_\epsilon * r \rrfloor^I * |T_1 * \mathbf{r}\dot{=} \llfloor V_1 \rrfloor_e|_\epsilon$, where

$$T_1 \triangleq \gamma(\llfloor Ls_1 \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor L \rrfloor_e) * \llfloor L \rrfloor_e \dot{\neq} \mathtt{null}$$

Similarly, we know

$$
\begin{aligned}
\llfloor |Q|_\epsilon * r \rrfloor^I &= \llfloor |S_2|_\epsilon * r \rrfloor^I * |T_2|_\epsilon\\
\llfloor \big|Q * X \dot{\in} \chi^{\mathrm{U}} * \mathbf{r}\dot{=}V_2\big|_\epsilon * r \rrfloor^I &= \llfloor |S_2|_\epsilon * r \rrfloor^I * \big|T_2 * \llfloor X \rrfloor_e \dot{\in} \chi^{\mathrm{U}} * \mathbf{r}\dot{=} \llfloor V_2 \rrfloor_e\big|_\epsilon\\
\llfloor |Q * \mathbf{r}\dot{=}L\,.\mathtt{x}|_\epsilon * r \rrfloor^I &= \llfloor |S_2|_\epsilon * r \rrfloor^I * |T_2 * \mathbf{r}\dot{=} \llfloor L \rrfloor_e .\mathtt{x}|_\epsilon\\
T_2 &= \gamma(\llfloor Ls_2 \rrfloor_e, \llfloor V_2 \rrfloor_e, \llfloor X \rrfloor_e)
\end{aligned}
$$

We then have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\tau : \{P\}\ \texttt{e1}\ \{R * \mathbf{r}\doteq V_1\}}\ \text{(I.H.)}}{\forall I, \epsilon, r.\ \{\llbracket |P|_\epsilon * r \rrbracket^I\}\ \llbracket\texttt{e1}\rrbracket\ \{\llbracket |R * \mathbf{r}\doteq V_1|_\epsilon * r \rrbracket^I\}}\ \tau\text{: Def.}}{\forall I,\epsilon,r. \left\{\llbracket |P|_\epsilon * r \rrbracket^I\right\} \llbracket\texttt{e1}\rrbracket \left\{\llbracket |S_1|_\epsilon * r \rrbracket^I * |T_1 * \mathbf{r}\doteq \llbracket V_1 \rrbracket_e|_\epsilon\right\}}\ \llbracket.\rrbracket\text{ Def.} \qquad (\ddagger)}{\forall I,\epsilon,r. \quad \begin{array}{c} \{\llbracket |P|_\epsilon * r \rrbracket^I\} \\ \llbracket\texttt{e1}\rrbracket\,\texttt{[}\llbracket\texttt{e2}\rrbracket\texttt{]} \\ \left\{\llbracket |S_2|_\epsilon * r \rrbracket^I * |\gamma(\llbracket Ls_2 \rrbracket_e, \llbracket V_2 \rrbracket_e, \llbracket X \rrbracket_e) * \mathbf{r}\doteq \llbracket L \rrbracket_e .\mathbf{x}|_\epsilon\right\} \end{array}}\ (\dagger)$$

$$\dfrac{\dfrac{\forall I,\epsilon,r. \quad \begin{array}{c} \{\llbracket |P|_\epsilon * r \rrbracket^I\} \\ \llbracket\texttt{e1}\rrbracket\,\texttt{[}\llbracket\texttt{e2}\rrbracket\texttt{]} \\ \left\{\llbracket |S_2|_\epsilon * r \rrbracket^I * |\gamma(\llbracket Ls_2 \rrbracket_e, \llbracket V_2 \rrbracket_e, \llbracket X \rrbracket_e) * \mathbf{r}\doteq \llbracket L \rrbracket_e .\mathbf{x}|_\epsilon\right\} \end{array}}{\{\llbracket |P|_\epsilon * r \rrbracket^I\}\ \llbracket\texttt{e1[e2]}\rrbracket\ \{\llbracket |Q * \mathbf{r}\doteq L.\mathbf{x}|_\epsilon * r \rrbracket^I\}}\ \llbracket.\rrbracket, \llbracket.\rrbracket\text{ Def.}}{\tau : \{P\}\ \texttt{e1[e2]}\ \{Q * \mathbf{r}\doteq \mathrm{L}.\mathbf{x}\}}\ \tau\text{: Def.}$$

where (†) denotes the application of the (Computed Access) rule, and

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\tau : \{R\}\ \texttt{e2}\ \{Q * X \dot{\in} \chi^{\mathrm{U}} * \mathbf{r}\doteq V_2\}}\ \text{(I.H.)}}{\forall I, \epsilon, r. \quad \begin{array}{c} \{\llbracket |R|_\epsilon * r \rrbracket^I\} \\ \llbracket\texttt{e2}\rrbracket \\ \left\{\llbracket |Q * X \dot{\in} \chi^{\mathrm{U}} * \mathbf{r}\doteq V_2|_\epsilon * r \rrbracket^I\right\} \end{array}}\ \tau\text{: Def.}}{\forall I, \epsilon, r. \quad \begin{array}{c} \left\{\llbracket |S_1|_\epsilon * r \rrbracket^I * |T_1|_\epsilon\right\} \\ \llbracket\texttt{e2}\rrbracket \\ \left\{\llbracket |S_2|_\epsilon * r \rrbracket^I * \left|\gamma(\llbracket Ls_2 \rrbracket_e, \llbracket V_2 \rrbracket_e, \llbracket X \rrbracket_e) * \llbracket X \rrbracket_e \dot{\in} \chi^{\mathrm{U}} * \mathbf{r}\doteq \llbracket V_2 \rrbracket_e\right|_\epsilon\right\} \end{array}}\ \llbracket.\rrbracket\text{ Def.}}{(\ddagger)}$$

Case (Object)
Let $P_i \triangleq R_i * \gamma(Ls_i, Y_i, X_i)$; $\{P_{i-1}\}\texttt{ei}\{P_i * \mathbf{r}\doteq Y_i\}$ for $i \in 1 \cdots n$ and $Q \triangleq P_n * R$; $R \triangleq \exists \mathrm{L}.\ \mathsf{newobj}(\mathrm{L}, @\mathit{proto}, \texttt{x1}, \cdots \texttt{xn}) * (\mathrm{L}, \texttt{x1}) \mapsto X_1 * \cdots * (\mathrm{L}, \texttt{xn}) \mapsto X_n * (\mathrm{L}, @\mathit{proto}) \mapsto l_{op} * \mathbf{r}\doteq \mathrm{L}$. Assume $\texttt{x1} \neq \cdots \neq \texttt{xn}$ and that $\mathbf{r} \notin \mathsf{fv}(P_n)$. Given an evaluation environment ϵ, an interface function I, a set of states r, and $i \in 1..n$, from from Lemma 18 and the definition of $\llbracket . \rrbracket^I$ we know $\llbracket |P_i|_\epsilon * r \rrbracket^I = \llbracket |R_i|_\epsilon * r \rrbracket^I * |T_i|_\epsilon$ and $\llbracket |P_i|_\epsilon * \mathbf{r}\doteq Y_i * r \rrbracket^I = \llbracket |R_i|_\epsilon * r \rrbracket^I * |T_i * \mathbf{r}\doteq \llbracket Y_i \rrbracket_e|_\epsilon$, where $T_i \triangleq \gamma(\llbracket Ls_i \rrbracket_e, \llbracket Y_i \rrbracket_e, \llbracket X_i \rrbracket_e)$.

Similarly, we know $\llfloor |Q * r|_\epsilon \rrfloor^I = \llfloor |R_n|_\epsilon * r \rrfloor^I * |T_n * R'|_\epsilon$, where

$$\begin{aligned} R' \triangleq\ & \exists \textsc{l}.\ \mathsf{newobj}(\textsc{l}, @\mathit{proto}, \texttt{x1}, \cdots \texttt{xn}) \\ & *(\textsc{l}, \texttt{x1}) \mapsto \llfloor X_1 \rrfloor_e * \cdots * (\textsc{l}, \texttt{xn}) \mapsto \llfloor X_n \rrfloor_e \\ & *(\textsc{l}, @\mathit{proto}) \mapsto l_{op} * \mathbf{r} \doteq \textsc{l} \end{aligned}$$

We then have:

$$\begin{array}{cl}
\hline
\tau : \{P_{i-1}\}\ \texttt{ei}\ \{P_i\} \quad \forall i \in 2..n & \text{(I.H.)} \\
\hline\hline
\forall I, \epsilon, r.\ \{\llfloor |P_{i-1}|_\epsilon * r \rrfloor^I\}\ \llbracket \texttt{ei} \rrbracket\ \{\llfloor |P_i|_\epsilon * r \rrfloor^I\}\ \forall i \in 2..n & \tau\text{: Def.} \\
\hline\hline
\forall I, \epsilon, r. \quad \{\llfloor |R_{i-1}|_\epsilon * r \rrfloor^I * |T_i|_\epsilon\} & \llfloor . \rrfloor\ \text{Def.} \\
(\ddagger) \qquad \llbracket \texttt{ei} \rrbracket \qquad \forall i \in 2..n & \\
\{\llfloor |R_i|_\epsilon * r \rrfloor^I * |T_i * \mathbf{r} \doteq \llfloor Y_i \rrfloor_e|_\epsilon\} & \\
\hline
\forall I, \epsilon, r. \quad \{\llfloor |P_0|_\epsilon * r \rrfloor^I\} & \text{(Object)} \\
\texttt{x1} : \llbracket \texttt{e1} \rrbracket, \ldots,\ \texttt{xn} : \llbracket \texttt{en} \rrbracket & \\
\{\llfloor |R_n|_\epsilon * r \rrfloor^I * |T_n * R'|_\epsilon\} & \\
\hline\hline
\forall I, \epsilon, r.\ \{\llfloor |P_0|_\epsilon * r \rrfloor^I\}\ \llbracket \texttt{x1}:\texttt{e1}, \ldots,\ \texttt{xn}:\texttt{en} \rrbracket\ \{\llfloor |Q|_\epsilon * r \rrfloor^I\} & \llfloor . \rrfloor, \llbracket . \rrbracket\ \text{Def.} \\
\hline\hline
\tau : \{P_0\}\ \texttt{x1}:\texttt{e1}, \ldots, \texttt{xn}:\texttt{en}\ \{Q\} & \tau\text{: Def.}
\end{array}$$

$$\begin{array}{cl}
\hline
\tau : \{P_0\}\ \texttt{e1}\ \{P_1\} & \text{(I.H.)} \\
\hline\hline
\forall I, \epsilon, r.\ \{\llfloor |P_0|_\epsilon * r \rrfloor^I\}\ \llbracket \texttt{e1} \rrbracket\ \{\llfloor |P_1|_\epsilon * r \rrfloor^I\} & \tau\text{: Def.} \\
\hline\hline
\forall I, \epsilon, r.\ \left\{\llfloor |P_0|_\epsilon * r \rrfloor^I\right\}\ \llbracket \texttt{e1} \rrbracket\ \left\{\llfloor |R_1|_\epsilon * r \rrfloor^I * |T_i * \mathbf{r} \doteq \llfloor Y_1 \rrfloor_e|_\epsilon\right\} & \llfloor . \rrfloor\ \text{Def.} \\
\hline
(\ddagger) &
\end{array}$$

Case (Binary Operators)

Let $Q \triangleq S_2 * \gamma(Ls_2, V_2, V_4)$; $R \triangleq S_1 * \gamma(Ls_1, V_1, V_3)$, and $V{=}V_3\overline{\oplus}V_4$.

Given an evaluation environment ϵ, an interface function I and a set of states r, from Lemma 18 and the definition of $\llbracket . \rrbracket^I$ we know

$$\begin{aligned}
\llbracket |Q * \mathbf{r}{\doteq}V_2|_\epsilon * r \rrbracket^I &= \llbracket |S_2|_\epsilon * r \rrbracket^I * |\gamma(\llbracket Ls_2 \rrbracket_e , \llbracket V_2 \rrbracket_e , \llbracket V_4 \rrbracket_e) * \mathbf{r}{\doteq} \llbracket V_2 \rrbracket_e|_\epsilon \\
\llbracket |R|_\epsilon * r \rrbracket^I &= \llbracket |S_1|_\epsilon * r \rrbracket^I * |\gamma(\llbracket Ls_1 \rrbracket_e , \llbracket V_1 \rrbracket_e , \llbracket V_3 \rrbracket_e)|_\epsilon \\
\llbracket |R * \mathbf{r}{\doteq}V_1|_\epsilon * r \rrbracket^I &= \llbracket |S_1|_\epsilon * r \rrbracket^I * |\gamma(\llbracket Ls_1 \rrbracket_e , \llbracket V_1 \rrbracket_e , \llbracket V_3 \rrbracket_e) * \mathbf{r}{\doteq} \llbracket V_1 \rrbracket_e|_\epsilon \\
\llbracket V \rrbracket_e &= \llbracket V_3 \rrbracket_e \overline{\oplus} \llbracket V_4 \rrbracket_e
\end{aligned}$$

Then we have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\tau : \{P\}\ \mathtt{e1}\ \{R * \mathbf{r}{\doteq}V_1\}}\ \text{(I.H.)}}{\forall I,\epsilon,r.\ \begin{array}{c}\{\llbracket |P|_\epsilon * r \rrbracket^I\}\\ \llbracket \mathtt{e1} \rrbracket \\ \{\llbracket |S_1 * \gamma(Ls_1,V_1,V_3) * \mathbf{r}{\doteq}V_1|_\epsilon * r \rrbracket^I\}\end{array}}\ \tau\text{: Def.}}{\forall I,\epsilon,r.\ \begin{array}{c}\{\llbracket |P|_\epsilon * r \rrbracket^I\}\\ \llbracket \mathtt{e1} \rrbracket \\ \left\{\begin{array}{l}\llbracket |S_1|_\epsilon * r \rrbracket^I\\ * |\gamma(\llbracket Ls_1 \rrbracket_e , \llbracket V_1 \rrbracket_e , \llbracket V_3 \rrbracket_e) * \mathbf{r}{\doteq} \llbracket V_1 \rrbracket_e|_\epsilon\end{array}\right\}\end{array}}\ \llbracket . \rrbracket \text{ Def.} \quad (\dagger) \quad (\ddagger)}{\forall I,\epsilon,r.\ \begin{array}{c}\{\llbracket |P|_\epsilon * r \rrbracket^I\}\\ \llbracket \mathtt{e1} \rrbracket \oplus \llbracket \mathtt{e2} \rrbracket \\ \left\{\begin{array}{l}\llbracket |S_2|_\epsilon * r \rrbracket^I\\ * |\gamma(\llbracket Ls_2 \rrbracket_e , \llbracket V_2 \rrbracket_e , \llbracket V_4 \rrbracket_e) * \mathbf{r}{\doteq} \llbracket V \rrbracket_e|_\epsilon\end{array}\right\}\end{array}}\ (*)}{\dfrac{\forall I,\epsilon,r.\ \{\llbracket |P|_\epsilon * r \rrbracket^I\}\ \llbracket \mathtt{e1} \oplus \mathtt{e2} \rrbracket\ \{\llbracket |Q * \mathbf{r}{\doteq}V|_\epsilon * r \rrbracket^I\}}{\tau : \{P\}\ \mathtt{e1} \oplus \mathtt{e2}\ \{Q * \mathbf{r}{\doteq}\mathrm{V}\}}\ \tau\text{: Def.}}\ \llbracket . \rrbracket, \llbracket . \rrbracket \text{ Def.}$$

where $(*)$ denotes the application of the (Binary Operators) rule, and

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\tau : \{R\}\ \mathsf{e2}\ \{Q * \mathbf{r}\dot{=}V_2\}}\ \text{(I.H.)}
}{
\forall I, \epsilon, r.\ \{\llbracket |R|_\epsilon * r \rrbracket^I\}\ \llbracket \mathsf{e2} \rrbracket\ \{\llbracket |Q * \mathbf{r}\dot{=}V_2|_\epsilon * r \rrbracket^I\}
}\ \tau\text{: Def.}
}{
\begin{array}{ll}
\forall I, \epsilon, r. & \{\llbracket |S_1 * \gamma(Ls_1, V_1, V_3)|_\epsilon * r \rrbracket^I\} \\
& \llbracket \mathsf{e2} \rrbracket \\
& \{\llbracket |S_2 * \gamma(Ls_2, V_2, V_4) * \mathbf{r}\dot{=}V_2|_\epsilon * r \rrbracket^I\}
\end{array}
}\ R,\ Q\ \text{Defs.}
}{
\begin{array}{ll}
\forall I, r, \epsilon. & \{\llbracket |S_1|_\epsilon * r \rrbracket^I * |\gamma(\llbracket Ls_1 \rrbracket_e, \llbracket V_1 \rrbracket_e, \llbracket V_3 \rrbracket_e)|_\epsilon\} \\
& \llbracket \mathsf{e2} \rrbracket \\
& \{\llbracket |S_2|_\epsilon * r \rrbracket^I * |\gamma(\llbracket Ls_2 \rrbracket_e, \llbracket V_2 \rrbracket_e, \llbracket V_4 \rrbracket_e) * \mathbf{r}\dot{=}\llbracket V_2 \rrbracket_e|_\epsilon\}
\end{array}
}\ \llbracket . \rrbracket\ \text{Def.}
$$

$$(\dagger)$$

$$
\dfrac{
\dfrac{}{V = V_3 \overline{\oplus} V_4}
}{
\llbracket V \rrbracket_e = \llbracket V_3 \rrbracket_e \overline{\oplus} \llbracket V_4 \rrbracket_e
}
$$

$$(\ddagger)$$

Case (Assign Global)
Let $Q \triangleq S * \gamma(Ls, V_1, V_2)$.
Given an evaluation environment ϵ, an interface function I and a set of states r, from Lemma 18 and the definition of $\llbracket . \rrbracket^I$ we know

$$
\begin{array}{l}
\llbracket |Q * (l_g, X) \mapsto \varnothing * \mathbf{r}\dot{=}V_1|_\epsilon * r \rrbracket^I = \llbracket |S|_\epsilon * r \rrbracket^I * \\
\qquad\qquad \left| \begin{array}{l} \gamma(\llbracket Ls \rrbracket_e, \llbracket V_1 \rrbracket_e, \llbracket V_2 \rrbracket_e) \\ *(l_g, \llbracket X \rrbracket_e) \mapsto \varnothing * \mathbf{r}\dot{=} \llbracket V_1 \rrbracket_e \end{array} \right|_\epsilon \\
\llbracket |Q * (l_g, X) \mapsto V_2 * \mathbf{r}\dot{=}V_2|_\epsilon * r \rrbracket^I = \llbracket |S|_\epsilon * r \rrbracket^I * \\
\qquad\qquad \left| \begin{array}{l} \gamma(\llbracket Ls \rrbracket_e, \llbracket V_1 \rrbracket_e, \llbracket V_2 \rrbracket_e) \\ *(l_g, \llbracket X \rrbracket_e) \mapsto \llbracket V_2 \rrbracket_e * \mathbf{r}\dot{=} \llbracket V_2 \rrbracket_e \end{array} \right|_\epsilon
\end{array}
$$

We then have:

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{}{\tau : \{P\}\ \mathtt{e1}\ \{R * \mathbf{r} \doteq \mathtt{null}.X\}}\ \text{(I.H.)}
    }{
      \forall I, \epsilon, r.\quad \begin{array}{c} \left\{ \llbracket |P|_{\epsilon} * r \rrbracket^{I} \right\} \\ \llbracket \mathtt{e1} \rrbracket \\ \left\{ \llbracket |R * \mathbf{r} \doteq \mathtt{null}.X|_{\epsilon} * r \rrbracket^{I} \right\} \end{array}
    }\ \tau\text{: Def.}
  }{
    \forall I, \epsilon, r.\quad \begin{array}{c} \{ \llbracket |P|_{\epsilon} * r \rrbracket^{I} \} \\ \llbracket \mathtt{e1} \rrbracket \\ \{ \llbracket |R|_{\epsilon} * r \rrbracket^{I} * |\mathbf{r} \doteq \mathtt{null}.\, \llbracket X \rrbracket_{e}|_{\epsilon} \} \end{array} \qquad (\ddagger)
  }\ \llbracket . \rrbracket\ \text{Def.}
}{
  \cfrac{
    \cfrac{
      \forall I,\epsilon,r.\ \left\{ \llbracket |P|_{\epsilon} * r \rrbracket^{I} \right\} \llbracket \mathtt{e1} \rrbracket \mathtt{=} \llbracket \mathtt{e2} \rrbracket \left\{ \begin{array}{l} \llbracket |S|_{\epsilon} * r \rrbracket^{I} * \\ \left| \begin{array}{l} \gamma(\llbracket Ls \rrbracket_{e}, \llbracket V_1 \rrbracket_{e}, \llbracket V_2 \rrbracket_{e}) \\ *(l_g, X) \mapsto \llbracket V_2 \rrbracket_{e} \\ * \mathbf{r} \doteq \llbracket V_2 \rrbracket_{e} \end{array} \right|_{\epsilon} \end{array} \right\}
    }{
      \forall I,\epsilon,r. \left\{ \llbracket |P|_{\epsilon} * r \rrbracket^{I} \right\} \llbracket \mathtt{e1=e2} \rrbracket \left\{ \left\llbracket \begin{array}{l} |Q * (l_g, X) \mapsto V_2 * \mathbf{r} \doteq V_2|_{\epsilon} \\ * r \end{array} \right\rrbracket^{I} \right\}
    }\ \llbracket . \rrbracket, \llbracket . \rrbracket\ \text{Def.}
  }{
    \tau : \{P\}\ \mathtt{e1 = e2}\ \{Q * (l_g, X) \mapsto V_2 * \mathbf{r} \doteq V_2\}
  }\ \tau\text{: Def.}
}\ (\dagger)
$$

where (†) denotes the application of the (Assign Global) rule, and

$$
\cfrac{
  \cfrac{
    \cfrac{}{\tau : \{R\}\ \mathtt{e2}\ \{Q * (l_g, X) \mapsto \varnothing * \mathbf{r} \doteq V_1\}}\ \text{(I.H.)}
  }{
    \forall I,\epsilon,r. \{ \llbracket |R|_{\epsilon} * r \rrbracket^{I} \}\ \llbracket \mathtt{e2} \rrbracket\ \{ \llbracket |Q * (l_g, X) \mapsto \varnothing * \mathbf{r} \doteq V_1|_{\epsilon} * r \rrbracket^{I} \}
  }\ \tau\text{: Def.}
}{
  \cfrac{
    \forall I, \epsilon, r.\quad \begin{array}{c} \{ \llbracket |R|_{\epsilon} * r \rrbracket^{I} \} \\ \llbracket \mathtt{e2} \rrbracket \\ \left\{ \begin{array}{l} \llbracket |S|_{\epsilon} * r \rrbracket^{I} * \\ |\gamma(\llbracket Ls \rrbracket_{e}, \llbracket V_1 \rrbracket_{e}, \llbracket V_2 \rrbracket_{e}) * (l_g, \llbracket X \rrbracket_{e}) \mapsto \varnothing * \mathbf{r} \doteq \llbracket V_1 \rrbracket_{e}|_{\epsilon} \end{array} \right\} \end{array}
  }{
    (\ddagger)
  }
}\ \llbracket . \rrbracket\ \text{Def.}
$$

Case (Assign Local)

Let $Q \triangleq S * \gamma(Ls, V_1, V_2)$.

Given an evaluation environment ϵ, an interface function I and a set of states r, from Lemma 18 and the definition of $\llbracket . \rrbracket^I$ we know

$$\llbracket |Q * (L, X) \mapsto V_3 * \mathbf{r}\doteq V_1|_\epsilon * r \rrbracket^I = \llbracket |S|_\epsilon * r \rrbracket^I * \left| \begin{array}{l} \gamma(\llbracket Ls \rrbracket_e, \llbracket V_1 \rrbracket_e, \llbracket V_2 \rrbracket_e) \\ *(\llbracket L \rrbracket_e, \llbracket X \rrbracket_e) \mapsto \llbracket V_3 \rrbracket_e * \mathbf{r}\doteq \llbracket V_1 \rrbracket_e \end{array} \right|_\epsilon$$

$$\llbracket |Q * (L, X) \mapsto V_2 * \mathbf{r}\doteq V_2|_\epsilon * r \rrbracket^I = \llbracket |S|_\epsilon * r \rrbracket^I * \left| \begin{array}{l} \gamma(\llbracket Ls \rrbracket_e, \llbracket V_1 \rrbracket_e, \llbracket V_2 \rrbracket_e) \\ *(\llbracket L \rrbracket_e, \llbracket X \rrbracket_e) \mapsto \llbracket V_2 \rrbracket_e * \mathbf{r}\doteq \llbracket V_2 \rrbracket_e \end{array} \right|_\epsilon$$

We then have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\tau : \{P\}\ \texttt{e1}\ \{R * \mathbf{r}\doteq L.X\}}\ \text{(I.H.)}}{\forall I, \epsilon, r.\quad \begin{array}{c} \{\llbracket |P|_\epsilon * r \rrbracket^I\} \\ \llbracket \texttt{e1} \rrbracket \\ \{\llbracket |R * \mathbf{r}\doteq L.X|_\epsilon * r \rrbracket^I\} \end{array}}\ \tau\text{: Def.}}{\forall I, \epsilon, r.\quad \begin{array}{c} \{\llbracket |P|_\epsilon * r \rrbracket^I\} \\ \llbracket \texttt{e1} \rrbracket \\ \{\llbracket |R|_\epsilon * r \rrbracket^I * |\mathbf{r}\doteq \llbracket L \rrbracket_e . \llbracket X \rrbracket_e|_\epsilon\} \end{array}}\ \llbracket . \rrbracket\ \text{Def.} \qquad (\ddagger)}{\forall I, \epsilon, r.\ \left\{\llbracket |P|_\epsilon * r \rrbracket^I\right\}\ \llbracket \texttt{e1} \rrbracket \texttt{=} \llbracket \texttt{e2} \rrbracket\ \left\{ \begin{array}{l} \llbracket |S|_\epsilon * r \rrbracket^I \\ * \left| \begin{array}{l} \gamma(\llbracket Ls \rrbracket_e, \llbracket V_1 \rrbracket_e, \llbracket V_2 \rrbracket_e) * \\ (\llbracket L \rrbracket_e, \llbracket X \rrbracket_e) \mapsto \llbracket V_2 \rrbracket_e \\ * \mathbf{r}\doteq \llbracket V_2 \rrbracket_e \end{array} \right|_\epsilon \end{array} \right\}}\ (\dagger)}{\dfrac{\forall I, \epsilon, r. \left\{\llbracket |P|_\epsilon * r \rrbracket^I\right\}\ \llbracket \texttt{e1=e2} \rrbracket\ \left\{ \left\llbracket \begin{array}{l} |Q * (L, X) \mapsto V_2 * \mathbf{r}\doteq V_2|_\epsilon \\ * r \end{array} \right\rrbracket^I \right\}}{\tau : \{P\}\ \texttt{e1 = e2}\ \{Q * (L, X) \mapsto V_2 * \mathbf{r}\doteq V_2\}}\ \tau\text{: Def.}}\ \llbracket . \rrbracket, \llbracket . \rrbracket\ \text{Def.}$$

where ($\dagger$) denotes the application of the (Assign Local) rule, and

$$
\begin{array}{c}
\hline
\tau : \{R\}\ \mathsf{e2}\ \{Q * (L,X) \mapsto V_3 * \mathbf{r} \doteq V_1\} \quad \text{(I.H.)} \\
\hline\hline
\forall I,\epsilon,r. \{\llfloor |R|_\epsilon * r \rrfloor^I\}\ \llbracket \mathsf{e2} \rrbracket\ \{\llfloor |Q * (L,X) \mapsto V_3 * \mathbf{r} \doteq V_1|_\epsilon * r \rrfloor^I\} \quad \tau\text{: Def.} \\
\hline\hline
\forall I,\epsilon,r. \quad \begin{array}{c}
\{\llfloor |R|_\epsilon * r \rrfloor^I\} \\
\llbracket \mathsf{e2} \rrbracket \\
\left\{ \begin{array}{l}
\llfloor |S|_\epsilon * r \rrfloor^I * \\
\left| \begin{array}{l}
\gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor V_2 \rrfloor_e) \\
* (\llfloor L \rrfloor_e, \llfloor X \rrfloor_e) \mapsto \llfloor V_3 \rrfloor_e * \mathbf{r} \doteq \llfloor V_1 \rrfloor_e
\end{array} \right|_\epsilon
\end{array} \right\}
\end{array} \quad \llfloor . \rrfloor \text{ Def.} \\
\hline
(\ddagger)
\end{array}
$$

Case (Function)

Let

$$
Q = \left(\begin{array}{l}
\exists \mathrm{L}_1, \mathrm{L}_2.\ \mathsf{newobj}(\mathrm{L}_1, @proto) * (\mathrm{L}_1, @proto) \mapsto l_{op} \\
* \mathsf{newobj}(\mathrm{L}_2, @proto, \mathtt{prototype}, @scope, @body) \\
* \mathsf{fun}(\mathrm{L}_2, \mathbf{l}, \mathtt{x}, \mathsf{e}, \mathrm{L}_1) * \mathbf{r} \doteq \mathrm{L}_2
\end{array} \right)
$$

Given an evaluation environment ϵ, an interface function I and a set of states r, from Lemma 18 and the definition of $\llfloor . \rrfloor^I$ we know $\llfloor |Q|_\epsilon * r \rrfloor^I = |Q'|_\epsilon * \llfloor r \rrfloor^I$, where

$$
Q' = \left(\begin{array}{l}
\exists \mathrm{L}_1, \mathrm{L}_2.\ \mathsf{newobj}(\mathrm{L}_1, @proto) * (\mathrm{L}_1, @proto) \mapsto l_{op} \\
* \mathsf{newobj}(\mathrm{L}_2, @proto, \mathtt{prototype}, @scope, @body) \\
* \mathsf{fun}(\mathrm{L}_2, \mathbf{l}, \mathtt{x}, \llbracket \mathsf{e} \rrbracket, \mathrm{L}_1) * \mathbf{r} \doteq \mathrm{L}_2
\end{array} \right)
$$

We then have:

$$
\begin{array}{c}
\hline
\forall \epsilon.\ \{|\mathsf{emp}|_\epsilon\}\ \mathtt{function(x)\{}\llbracket \mathsf{e} \rrbracket\mathtt{\}}\ \{|Q'|_\epsilon\} \quad \text{(Function)} \\
\hline
\forall I,\epsilon,r.\ \{|\mathsf{emp}|_\epsilon * \llfloor r \rrfloor^I\}\ \mathtt{function(x)\{}\llbracket \mathsf{e} \rrbracket\mathtt{\}}\ \{|Q'|_\epsilon * \llfloor r \rrfloor^I\} \quad \text{(Frame)} \\
\hline\hline
\forall I,\epsilon,r.\ \{\llfloor |\mathsf{emp}|_\epsilon * r \rrfloor^I\}\ \llbracket \mathtt{function(x)\{e\}} \rrbracket\ \{\llfloor |Q|_\epsilon * r \rrfloor^I\} \quad \llfloor . \rrfloor, \llbracket . \rrbracket \text{ Def.} \\
\hline\hline
\tau : \{\mathsf{emp}\}\ \mathtt{function(x)\{e\}}\ \{Q\} \quad \tau\text{: Def.}
\end{array}
$$

Case (Named Function)

The proof of this case is analogous to that (Function) and is omitted here.

Case (While)

Let $S \triangleq R * \gamma(Ls, V_1, V_2)$, $Q \triangleq S * \mathsf{False}(V_2) * \mathbf{r} \doteq \texttt{undefined}$.

Given an evaluation environment ϵ, an interface function I and a set of states r, from Lemma 18 and the definition of $\llfloor . \rrfloor^I$ we know

$$\llfloor |S * \mathbf{r} \doteq V_1|_\epsilon * r \rrfloor^I = \llfloor |R|_\epsilon * r \rrfloor^I * \left| \gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor V_2 \rrfloor_e) * \mathbf{r} \doteq \llfloor V_1 \rrfloor_e \right|_\epsilon$$

$$\llfloor |S * \mathsf{True}(V_2)|_\epsilon * r \rrfloor^I = \llfloor |R|_\epsilon * r \rrfloor^I * \left| \gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor V_2 \rrfloor_e) * \mathsf{True}(\llfloor V_2 \rrfloor_e) \right|_\epsilon$$

$$\llfloor |Q|_\epsilon * r \rrfloor^I = \llfloor |R|_\epsilon * r \rrfloor^I * \left| \begin{array}{l} \gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor V_2 \rrfloor_e) \\ * \mathsf{False}(\llfloor V_2 \rrfloor_e) * \mathbf{r} \doteq \texttt{undefined} \end{array} \right|_\epsilon$$

We then have:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\tau : \{P\}\ \texttt{e1}\ \{S * \mathbf{r} \doteq V_1\}}\ \text{(I.H.)}}{\forall I, \epsilon, r. \{\llfloor |P|_\epsilon * r \rrfloor^I\}\ \llbracket \texttt{e1} \rrbracket\ \{\llfloor |S * \mathbf{r} \doteq V_1|_\epsilon * r \rrfloor^I\}}\ \tau\text{: Def.}}{\forall I, \epsilon, r. \left\{ \llfloor |P|_\epsilon * r \rrfloor^I \right\} \llbracket \texttt{e1} \rrbracket \left\{ \begin{array}{l} \llfloor |R|_\epsilon * r \rrfloor^I * \\ \left| \begin{array}{l} \gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor V_2 \rrfloor_e) \\ * \mathbf{r} \doteq \llfloor V_1 \rrfloor_e \end{array} \right|_\epsilon \end{array} \right\}}\ \llfloor . \rrfloor\text{ Def.} \qquad (\dagger)}{\dfrac{\dfrac{\begin{array}{c} \forall I, \epsilon, r. \qquad \{\llfloor |P|_\epsilon * r \rrfloor^I\} \\ \texttt{while(}\llbracket \texttt{e1} \rrbracket\texttt{)\{}\llbracket \texttt{e2} \rrbracket\texttt{\}} \\ \left\{ \llfloor |R|_\epsilon * r \rrfloor^I * \left| \begin{array}{l} \gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor 2 \rrfloor_e) \\ * \mathsf{False}(\llfloor V_2 \rrfloor_e) * \mathbf{r} \doteq \texttt{undefined} \end{array} \right|_\epsilon \right\} \end{array}}{\forall I, \epsilon, r.\ \{\llfloor |P|_\epsilon * r \rrfloor^I\}\ \llbracket \texttt{while(e1)\{e2\}} \rrbracket\ \{\llfloor |Q|_\epsilon * r \rrfloor^I\}}\ \llfloor . \rrfloor, \llbracket . \rrbracket\text{ Def.}}{\tau : \{P\}\ \texttt{while(e1)\{e2\}}\ \{Q\}}\ \tau\text{: Def.}}\ \text{(While)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\tau : \{S * \mathsf{True}(V_2)\}\ \texttt{e2}\ \{P\}}\ \text{(I.H.)}}{\forall I, \epsilon, r.\ \{\llfloor |S * \mathsf{True}(V_2)|_\epsilon * r \rrfloor^I\}\ \llbracket \texttt{e2} \rrbracket\ \{\llfloor |P|_\epsilon * r \rrfloor^I\}}\ \tau\text{: Def.}}{\forall I, \epsilon, r. \left\{ \begin{array}{l} \llfloor |R|_\epsilon * r \rrfloor^I * \\ |\gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor V_2 \rrfloor_e) * \mathsf{True}(\llfloor V_2 \rrfloor_e))|_\epsilon \end{array} \right\} \llbracket \texttt{e2} \rrbracket \left\{ \llfloor |P|_\epsilon * r \rrfloor^I \right\}}\ \llfloor . \rrfloor\text{ Def.}}{(\dagger)}$$

Case If

This case is analogous to that of (While) and is omitted here.

Case (With)

Let $S \triangleq R * \gamma(Ls, V_1, L_1)$.

Given an evaluation environment ϵ, an interface function I, a set of states r and assertions P and Q, from Lemma 18 and the definition of $\llfloor . \rrfloor^I$ we know

$$
\begin{aligned}
\llfloor |S * \mathbf{l}\doteq L * \mathbf{r}\doteq V_1|_\epsilon * r \rrfloor^I &= \llfloor |R|_\epsilon * r \rrfloor^I * \left| \begin{array}{l} \gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor L_1 \rrfloor_e) \\ * \mathbf{l}\doteq \llfloor L \rrfloor_e * \mathbf{r}\doteq \llfloor V_1 \rrfloor_e \end{array} \right|_\epsilon \\
\llfloor |S * \mathbf{l}\doteq L|_\epsilon * r \rrfloor^I &= \llfloor |R|_\epsilon * r \rrfloor^I * \left| \begin{array}{l} \gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor L_1 \rrfloor_e) \\ * \mathbf{l}\doteq \llfloor L \rrfloor_e \end{array} \right|_\epsilon \\
\llfloor |S * \mathbf{l}\doteq L_1 : L|_\epsilon * r \rrfloor^I &= \llfloor |R|_\epsilon * r \rrfloor^I * \left| \begin{array}{l} \gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor L_1 \rrfloor_e) \\ * \mathbf{l}\doteq \llfloor L_1 \rrfloor_e : \llfloor L \rrfloor_e \end{array} \right|_\epsilon \\
\llfloor |P * \mathbf{l}\doteq L|_\epsilon * r \rrfloor^I &= \llfloor |P|_\epsilon * r \rrfloor^I * \left| \mathbf{l}\doteq \llfloor L \rrfloor_e \right|_\epsilon \\
\llfloor |Q * \mathbf{l}\doteq L|_\epsilon * r \rrfloor^I &= \llfloor |Q|_\epsilon * r \rrfloor^I * \left| \mathbf{l}\doteq \llfloor L \rrfloor_e \right|_\epsilon \\
\llfloor |P * \mathbf{l}\doteq L_1 : L|_\epsilon * r \rrfloor^I &= \llfloor |P|_\epsilon * r \rrfloor^I * \left| \mathbf{l}\doteq \llfloor L_1 \rrfloor_e : \llfloor L \rrfloor_e \right|_\epsilon
\end{aligned}
$$

We then have:

$$
\begin{array}{cl}
\hline
\tau : \{P * \mathbf{l}\doteq \mathrm{L}\}\ \texttt{e1}\ \{S * \mathbf{l}\doteq \mathrm{L} * \mathbf{r}\doteq \mathrm{V}_1\} & \text{(I.H.)} \\
\hline\hline
\forall I,\epsilon,r. \quad \{\llfloor |P * \mathbf{l}\doteq L|_\epsilon * r \rrfloor^I\} & \tau\text{: Def.} \\
\llbracket \texttt{e1} \rrbracket & \\
\{\llfloor |S * \mathbf{l}\doteq L * \mathbf{r}\doteq V_1|_\epsilon * r \rrfloor^I\} & \\
\hline\hline
\forall I,\epsilon,r. \quad \{\llfloor |P|_\epsilon * r \rrfloor^I * |\mathbf{l}\doteq \llfloor L \rrfloor_e|_\epsilon\} & \llfloor . \rrfloor \text{ Def.} \\
\llbracket \texttt{e1} \rrbracket & (\dagger) \\
\left\{ \begin{array}{l} \llfloor |R|_\epsilon * r \rrfloor^I * \\ |\gamma(\llfloor Ls \rrfloor_e, \llfloor V_1 \rrfloor_e, \llfloor L_1 \rrfloor_e) * \mathbf{l}\doteq \llfloor L \rrfloor_e * \mathbf{r}\doteq \llfloor V_1 \rrfloor_e|_\epsilon \end{array} \right\} & \\
\hline
\forall I,\epsilon,r. \left\{ \begin{array}{l} \llfloor |P|_\epsilon * r \rrfloor^I \\ * |\mathbf{l}\doteq \llfloor L \rrfloor_e|_\epsilon \end{array} \right\} \texttt{with(}\llbracket \texttt{e1} \rrbracket\texttt{)\{}\llbracket \texttt{e2} \rrbracket\texttt{\}} \left\{ \begin{array}{l} \llfloor |Q|_\epsilon * r \rrfloor^I \\ * |\mathbf{l}\doteq \llfloor L \rrfloor_e|_\epsilon \end{array} \right\} & \text{(With)} \\
\hline\hline
\forall I,\epsilon,r. \{\llfloor |P * \mathbf{l}\doteq L|_\epsilon * r \rrfloor^I\}\ \llbracket \texttt{with(e1)\{e2\}} \rrbracket\ \{\llfloor |Q * \mathbf{l}\doteq L|_\epsilon * r \rrfloor^I\} & \llfloor . \rrfloor, \llbracket . \rrbracket \text{ Def.} \\
\hline\hline
\tau : \{P * \mathbf{l}\doteq L\}\ \texttt{with(e1)\{e2\}}\ \{Q * \mathbf{l}\doteq L\} & \tau\text{: Def.}
\end{array}
$$

where (†) denotes the application of the (Assign Local) rule, and

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\tau : \{S * \mathbf{l} \doteq L_1{:}L\} \ \mathsf{e2} \ \{Q * \mathbf{l} \doteq L_1{:}L\}}\ \text{(I.H.)}}{\forall I,\epsilon,r.\{\llbracket |S * \mathbf{l} \doteq L_1{:}L|_\epsilon * r \rrbracket^I\} \ \llbracket \mathsf{e2} \rrbracket \ \{\llbracket |Q * \mathbf{l} \doteq L_1{:}L|_\epsilon * r \rrbracket^I\}}\ \tau\text{: Def.}}{\forall I,\epsilon,r. \left\{ \begin{array}{l} \llbracket |R|_\epsilon * r \rrbracket^I * \\ \left| \begin{array}{l} \gamma(\llbracket Ls \rrbracket_e , \llbracket V_1 \rrbracket_e , \llbracket L_1 \rrbracket_e) \\ * \, \mathbf{l} \doteq \llbracket L_1 \rrbracket_e : \llbracket L \rrbracket_e \end{array} \right|_\epsilon \end{array} \right\} \llbracket \mathsf{e2} \rrbracket \left\{ \begin{array}{l} \llbracket |Q|_\epsilon * r \rrbracket^I \\ * \, |\mathbf{l} \doteq \llbracket L_1 \rrbracket_e : \llbracket L \rrbracket_e|_\epsilon \end{array} \right\}}\ \llbracket . \rrbracket\ \text{Def.}}{(\dagger)}$$

Case (Function Call)
Let

$$\begin{aligned}
R_1 &\triangleq S_1 * \mathsf{This}(F_1, T) \uplus \gamma(Ls_1, F_1, F_2) \\
&\quad * (F_2, @\mathit{body}) \mapsto \lambda \mathrm{X}.\mathsf{e}_3 * (F_2, @\mathit{scope}) \mapsto Ls_2 \\
R_2 &\triangleq S_2 * \gamma(Ls_4, V_1, V_2) \\
R_3 &\triangleq R_2 * \exists \mathrm{L}.\ \mathbf{l} \doteq \mathrm{L} : Ls_2 * (\mathrm{L}, X) \mapsto V_2 * (\mathrm{L}, @\mathit{this}) \mapsto T \\
&\quad * (\mathrm{L}, @\mathit{proto}) \mapsto \mathtt{null} * \mathsf{defs}(X, \mathrm{L}, \mathsf{e3}) \\
&\quad * \mathsf{newobj}(\mathrm{L}, @\mathit{proto}, @\mathit{this}, X, \mathsf{decls}(X, \mathrm{L}, \mathsf{e3}))
\end{aligned}$$

Given an evaluation environment $\epsilon {=} (\Gamma, L')$, an interface function I, a set of states r and assertions P and Q, from Lemma 18 and the definition of $\llbracket . \rrbracket^I$ we know

$$\begin{aligned}
\llbracket |R_1|_\epsilon * r \rrbracket^I &= \llbracket |S_1|_\epsilon * r \rrbracket^I * |T_1|_\epsilon \\
\llbracket |R_1 * \mathbf{r} \doteq F_1|_\epsilon * r \rrbracket^I &= \llbracket |S_1|_\epsilon * r \rrbracket^I * |T_1 * \mathbf{r} \doteq \llbracket F_1 \rrbracket_e|_\epsilon \\
& T_1 {=} \mathsf{This}(\llbracket F_1 \rrbracket_e , \llbracket T \rrbracket_e) \uplus \gamma(\llbracket Ls_1 \rrbracket_e , \llbracket F_1 \rrbracket_e , \llbracket F_2 \rrbracket_e) \\
& \qquad * \ (\llbracket F_2 \rrbracket_e , @\mathit{body}) \mapsto \lambda \mathrm{X}. \llbracket \mathsf{e}_3 \rrbracket \\
& \qquad * \ (\llbracket F_2 \rrbracket_e , @\mathit{scope}) \mapsto \llbracket Ls_2 \rrbracket_e \\
\llbracket |R_2 * \mathbf{l} \doteq Ls_3 * \mathbf{r} \doteq V_1|_\epsilon * r \rrbracket^I &= \llbracket |S_2|_\epsilon * r \rrbracket^I * \left| \begin{array}{l} \gamma(\llbracket Ls_4 \rrbracket_e , \llbracket V_1 \rrbracket_e , \llbracket V_2 \rrbracket_e) \\ * \, \mathbf{l} \doteq \llbracket Ls_3 \rrbracket_e * \mathbf{r} \doteq \llbracket V_1 \rrbracket_e \end{array} \right|_\epsilon \\
\llbracket |Q * \mathbf{l} \doteq L{:}Ls_2|_\epsilon * r \rrbracket^I &= \llbracket |Q|_\epsilon * r \rrbracket^I * \left| \mathbf{l} \doteq \llbracket L \rrbracket_e : \llbracket Ls_2 \rrbracket_e \right|_\epsilon
\end{aligned}$$

$$\begin{aligned}
\lfloor\!\lfloor |Q * \mathbf{l} \dot{=} Ls_3|_\epsilon * r \rfloor\!\rfloor^I &= \lfloor\!\lfloor |Q|_\epsilon * r \rfloor\!\rfloor^I * \left| \mathbf{l} \dot{=} \lfloor\!\lfloor Ls_3 \rfloor\!\rfloor_e \right|_\epsilon \\
\lfloor\!\lfloor |R_3|_\epsilon * r \rfloor\!\rfloor^I &= \lfloor\!\lfloor |S_2|_\epsilon * r \rfloor\!\rfloor^I * |T_3|_\epsilon \\
T_3 =& \gamma(\lfloor\!\lfloor Ls_4 \rfloor\!\rfloor_e, \lfloor\!\lfloor V_1 \rfloor\!\rfloor_e, \lfloor\!\lfloor V_2 \rfloor\!\rfloor_e) \\
& * \exists \textsc{l}.\ \mathbf{l} \dot{=} \textsc{l}\text{: } \lfloor\!\lfloor Ls_2 \rfloor\!\rfloor_e * (\textsc{l}, \lfloor\!\lfloor X \rfloor\!\rfloor_e) \mapsto \lfloor\!\lfloor V_2 \rfloor\!\rfloor_e \\
& * (\textsc{l}, @\mathit{this}) \mapsto \lfloor\!\lfloor T \rfloor\!\rfloor_e * (\textsc{l}, @\mathit{proto}) \mapsto \texttt{null} \\
& * \mathsf{defs}(\lfloor\!\lfloor X \rfloor\!\rfloor_e, \textsc{l}, \llbracket \mathsf{e3} \rrbracket) \\
& * \mathsf{newobj}\begin{pmatrix} \textsc{l}, @\mathit{proto}, @\mathit{this}, \lfloor\!\lfloor X \rfloor\!\rfloor_e, \\ \mathsf{decls}(\lfloor\!\lfloor X \rfloor\!\rfloor_e, \textsc{l}, \llbracket \mathsf{e3} \rrbracket) \end{pmatrix}
\end{aligned}$$

We then have:

$$\begin{array}{c}
(\dagger) \quad (\ddagger) \quad (\dagger\dagger) \quad (\ddagger\ddagger) \\ \hline
\forall I, \Gamma, r. \quad \{\lfloor\!\lfloor |P|_\epsilon * r \rfloor\!\rfloor^I\} \\
\llbracket \mathsf{e1(e2)} \rrbracket \\
\left\{ \bigcup_{v \in \text{JSLVal}_{\text{DOM}}} \left(\left\lfloor\!\left\lfloor |Q|_{([\Gamma|\textsc{l} \mapsto v], L')} * r \right\rfloor\!\right\rfloor^I * |\mathbf{l} \dot{=} Ls_3|_{([\Gamma|\textsc{l} \mapsto v], L')} \right) \right\} \\ \hline\hline
\forall I, \Gamma, r. \{\lfloor\!\lfloor |P|_\epsilon * r \rfloor\!\rfloor^I\} \ \llbracket \mathsf{e1(e2)} \rrbracket \ \{\lfloor\!\lfloor |\exists \textsc{l}.\ Q * \mathbf{l} \dot{=} Ls_3|_\epsilon * r \rfloor\!\rfloor^I\} \\ \hline\hline
\tau : \{P\}\ \mathsf{e1(e2)}\ \{\exists \textsc{l}.\ Q * \mathbf{l} \dot{=} Ls_3\}
\end{array}
\begin{array}{l} \text{(Function Call)} \\ \\ \\ \\ \\ \tau\text{: Def.} \end{array}$$

$$\begin{array}{c}
\\ \hline
\tau : \{P\}\ \mathsf{e1}\ \{R_1 * \mathbf{r} \dot{=} F_1\} \\ \hline\hline
\forall I, \Gamma, r. \{\lfloor\!\lfloor |P|_\epsilon * r \rfloor\!\rfloor^I\} \ \llbracket \mathsf{e1} \rrbracket \ \{\lfloor\!\lfloor |R_1 * \mathbf{r} \dot{=} F_1|_\epsilon * r \rfloor\!\rfloor^I\} \\ \hline\hline
\forall I, \Gamma, r. \quad \{\lfloor\!\lfloor |P|_\epsilon * r \rfloor\!\rfloor^I\} \\
\llbracket \mathsf{e1} \rrbracket \\
\{\lfloor\!\lfloor |S_1|_\epsilon * r \rfloor\!\rfloor^I * |T_1 * \mathbf{r} \dot{=} F_1|_\epsilon\} \\ \hline
(\dagger)
\end{array}
\begin{array}{l} \text{(I.H.)} \\ \tau\text{: Def.} \end{array}$$

$$\begin{array}{c}
\\ \hline
\tau : \{R_1\}\ \mathsf{e2}\ \{R_2 * \mathbf{l} \dot{=} Ls_3 * \mathbf{r} \dot{=} V_1\} \\ \hline\hline
\forall I, \Gamma, r. \{\lfloor\!\lfloor |R_1|_\epsilon * r \rfloor\!\rfloor^I\} \ \llbracket \mathsf{e2} \rrbracket \ \{\lfloor\!\lfloor |R_2 * \mathbf{l} \dot{=} Ls_3 * \mathbf{r} \dot{=} V_1|_\epsilon * r \rfloor\!\rfloor^I\} \\ \hline\hline
\forall I, \Gamma, r. \quad \{\lfloor\!\lfloor |S_1|_\epsilon * r \rfloor\!\rfloor^I * |T_1|_\epsilon\} \\
\llbracket \mathsf{e2} \rrbracket \\
\left\{ \lfloor\!\lfloor |S_2|_\epsilon * r \rfloor\!\rfloor^I * \left| \begin{array}{l} \gamma(\lfloor\!\lfloor Ls_4 \rfloor\!\rfloor_e, \lfloor\!\lfloor V_1 \rfloor\!\rfloor_e, \lfloor\!\lfloor V_2 \rfloor\!\rfloor_e) \\ * \mathbf{l} \dot{=} \lfloor\!\lfloor Ls_3 \rfloor\!\rfloor_e * \mathbf{r} \dot{=} \lfloor\!\lfloor V_1 \rfloor\!\rfloor_e \end{array} \right|_\epsilon \right\} \\ \hline
(\ddagger)
\end{array}
\begin{array}{l} \text{(I.H.)} \\ \tau\text{: Def.} \end{array}$$

$$\begin{array}{c}
\hline
\tau : \{R_3\}\ \mathsf{e3}\ \{\exists \mathrm{L}.\ Q * \mathbf{l} \doteq \mathrm{L}{:}Ls_2\} \quad (\text{I.H.}) \\
\hline\hline
\forall I, \Gamma, r.\ \{\llbracket |R_3|_\epsilon * r \rrbracket^I\}\ \llbracket \mathsf{e3} \rrbracket\ \{\llbracket |\exists \mathrm{L}.\ Q * \mathbf{l} \doteq \mathrm{L}{:}Ls_2|_\epsilon * r \rrbracket^I\} \quad \tau\text{: Def.} \\
\hline\hline
\forall I, \Gamma, r. \qquad \{|T_3|_\epsilon * \llbracket |S_2|_\epsilon * r \rrbracket^I\} \\
\llbracket \mathsf{e3} \rrbracket \\
\left\{ \bigcup_{v \in \mathrm{JSLV\small{AL}}_{\mathbb{DOM}}} \left(\left\llbracket |Q|_{[\Gamma|\mathrm{L} \mapsto v]} * r \right\rrbracket^I * |\mathbf{l} \doteq \mathrm{L}{:}Ls_2|_{([\Gamma|\mathrm{L} \mapsto v], L')} \right) \right\} \\
\hline
(\dagger\dagger)
\end{array}$$

$$\begin{array}{c}
\hline
\mathbf{l} \notin \mathsf{fv}(Q,\ R_2) \\
\hline
\forall I, \Gamma, r.\ \mathbf{l} \notin \mathsf{fv} \left(\llbracket |Q|_\epsilon * r \rrbracket^I,\ \llbracket |S_2|_\epsilon * r \rrbracket^I * |\gamma(\llbracket Ls_4 \rrbracket_e, \llbracket V_1 \rrbracket_e, \llbracket V_2 \rrbracket_e)|_\epsilon \right) \\
\hline
(\ddagger\ddagger)
\end{array}$$

Case (Frame)

$$\begin{array}{c}
\hline
\tau : \{P\}\ \mathsf{e}\ \{Q\} \quad (\text{I.H.}) \\
\hline\hline
\forall I, \epsilon, r.\ \{\llbracket |P|_\epsilon * r \rrbracket^I\}\ \llbracket \mathsf{e} \rrbracket\ \{\llbracket |Q|_\epsilon * r \rrbracket^I\} \quad \tau\text{: Def.} \\
\hline
\forall I, \epsilon, r.\ \{\llbracket |P|_\epsilon * |R|_\epsilon * r \rrbracket^I\}\ \llbracket \mathsf{e} \rrbracket\ \{\llbracket |Q|_\epsilon\, |R|_\epsilon * r \rrbracket^I\} \\
\hline\hline
\forall I, \epsilon, r.\ \{\llbracket |P * R|_\epsilon * r \rrbracket^I\}\ \llbracket \mathsf{e} \rrbracket\ \{\llbracket |Q * R|_\epsilon * r \rrbracket^I\} \\
\hline\hline
\tau : \{P * R\}\ \mathsf{e}\ \{Q * R\} \quad \tau\text{: Def.}
\end{array}$$

Case (Consequence)

$$\begin{array}{c}
\hline
\tau : \{P'\}\ \mathsf{e}\ \{Q'\} \quad (\text{I.H.}) \\
\hline\hline
(\dagger) \quad \forall I, \epsilon, r. \left\{ \llbracket |P'|_\epsilon * r \rrbracket^I \right\}\ \llbracket \mathsf{e} \rrbracket\ \left\{ \llbracket |Q'|_\epsilon * r \rrbracket^I \right\} \quad \tau\text{: Def.} \quad (\ddagger) \\
\hline
\forall I, \epsilon, r.\ \left\{ \llbracket |P|_\epsilon * r \rrbracket^I \right\}\ \llbracket \mathsf{e} \rrbracket\ \left\{ \llbracket |Q|_\epsilon * r \rrbracket^I \right\} \quad (*) \\
\hline\hline
\tau : \{P\}\ \mathsf{e}\ \{Q\} \quad \tau\text{: Def.}
\end{array}$$

where $(*)$ denotes the application of the (Consequence) rule, and

$$\begin{array}{c}
\hline
P \vdash P' \\
\hline
\forall I, \epsilon, r.\ \left\{ \llbracket |P|_\epsilon * r \rrbracket^I \right\} \subseteq \left\{ \llbracket |P'|_\epsilon * r \rrbracket^I \right\} \quad \text{Lemma 19} \\
\hline
(\dagger)
\end{array}$$

$$
\begin{array}{cl}
\overline{Q' \vdash Q} & \\
\hline
\forall I, \epsilon, r.\ \left\{ \llfloor |Q'|_{\epsilon} * r \rrfloor^{I} \right\} \subseteq \left\{ \llfloor |Q|_{\epsilon} * r \rrfloor^{I} \right\} & \text{Lemma 19} \\
\hline
(\ddagger) &
\end{array}
$$

Case (Elimination)

$$
\begin{array}{cl}
\overline{\tau : \{P\}\ \mathsf{e}\ \{Q\}} & \text{(I.H.)} \\
\hline\hline
\forall I, \epsilon, r.\ \left\{ \llfloor |P|_{\epsilon} * r \rrfloor^{I} \right\}\ \llbracket \mathsf{e} \rrbracket\ \left\{ \llfloor |P|_{\epsilon} * r \rrfloor^{I} \right\} & \tau\text{: Def.} \\
\hline
\forall I, \epsilon, r. \left\{ \exists x.\ \llfloor |P|_{\epsilon} * r \rrfloor^{I} \right\}\ \llbracket \mathsf{e} \rrbracket\ \left\{ \exists x.\ \llfloor |Q|_{\epsilon} * r \rrfloor^{I} \right\} & \text{(Elimination)} \\
\hline
\begin{array}{c}
\forall I, \epsilon, r.\ \left\{ \bigcup\limits_{v \in \mathrm{JSLV{\small AL}}_{\mathbb{DOM}}} \left(\left\llfloor \left\{ \mathsf{w}_1 + \mathsf{w}_2 \,\middle|\, \begin{array}{l} \mathsf{w}_1 \in |P|_{\epsilon} \land \\ \mathsf{w}_2 \in r \land x = v \end{array} \right\} \right\rrfloor^{I} \right) \right\} \\
\llbracket \mathsf{e} \rrbracket \\
\left\{ \bigcup\limits_{v \in \mathrm{JSLV{\small AL}}_{\mathbb{DOM}}} \left(\left\llfloor \left\{ \mathsf{w}_1 + \mathsf{w}_2 \,\middle|\, \begin{array}{l} \mathsf{w}_1 \in |Q|_{\epsilon} \land \\ \mathsf{w}_2 \in r \land x = v \end{array} \right\} \right\rrfloor^{I} \right) \right\}
\end{array} & \\
\hline
\begin{array}{c}
\forall I, \epsilon, r.\ \left\{ \bigcup\limits_{v \in \mathrm{JSLV{\small AL}}_{\mathbb{DOM}}} \left(\llfloor \{ \mathsf{w} \mid \mathsf{w} \in |P|_{\epsilon} \land x = v \} * r \rrfloor^{I} \right) \right\} \\
\llbracket \mathsf{e} \rrbracket \\
\left\{ \bigcup\limits_{v \in \mathrm{JSLV{\small AL}}_{\mathbb{DOM}}} \left(\llfloor \{ \mathsf{w} \mid \mathsf{w} \in |Q|_{\epsilon} \land x = v \} * r \rrfloor^{I} \right) \right\}
\end{array} & * \text{ Def.} \\
\hline
\forall I, \epsilon, r. \left\{ \llfloor |\exists \mathrm{x}.\ P|_{\epsilon} * r \rrfloor^{I} \right\}\ \llbracket \mathsf{e} \rrbracket\ \left\{ \llfloor |\exists \mathrm{x}.\ Q|_{\epsilon} * r \rrfloor^{I} \right\} & \exists, \llfloor . \rrfloor \text{ Def.} \\
\hline\hline
\tau : \{\exists \mathrm{x}.\ P\}\ \mathsf{e}\ \{\exists \mathrm{x}.\ Q\} & \tau\text{: Def.}
\end{array}
$$

Case (Disjunction)

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{\tau : \{P_1\}\ \mathsf{e}\ \{Q_1\}}\ \text{(I.H.)}
      }{
        \begin{array}{c}\forall I, \epsilon, r.\ \{\llfloor |P_1|_\epsilon * r \rrbracket^I\} \\ \llbracket \mathsf{e} \rrbracket \\ \{\llfloor |Q_1|_\epsilon * r \rrbracket^I\}\end{array}
      }
      \qquad
      \dfrac{
        \dfrac{}{\tau : \{P_2\}\ \mathsf{e}\ \{Q_2\}}\ \text{(I.H.)}
      }{
        \begin{array}{c}\forall I, \epsilon, r.\ \{\llfloor |P_2|_\epsilon * r \rrbracket^I\} \\ \llbracket \mathsf{e} \rrbracket \\ \{\llfloor |Q_2|_\epsilon * r \rrbracket^I\}\end{array}
      }
    }{
      \begin{array}{c}\forall I, \epsilon, r.\ \{\llfloor |P_1|_\epsilon * r \rrbracket^I \lor \llfloor |P_2|_\epsilon * r \rrbracket^I\} \\ \llbracket \mathsf{e} \rrbracket \\ \{\llfloor |Q_1|_\epsilon * r \rrbracket^I \lor \llfloor |Q_2|_\epsilon * r \rrbracket^I\}\end{array}
    }\ (\ddagger)
  }{
    \begin{array}{c}\forall I, \epsilon, r.\ \{\llfloor (|P_1|_\epsilon * r) \cup (|P_2|_\epsilon * r) \rrbracket^I\} \\ \llbracket \mathsf{e} \rrbracket \\ \{\llfloor (|Q_1|_\epsilon * r) \cup (|Q_2|_\epsilon * r) \rrbracket^I\}\end{array}
  }\ \llfloor . \rrbracket\ \text{Def.}
}{
  \dfrac{
    \forall I, \epsilon, r.\ \left\{\llfloor |(P_1 \lor P_2)|_\epsilon * r \rrbracket^I\right\}\ \llbracket \mathsf{e} \rrbracket\ \left\{\llfloor |(Q_1 \lor Q_2)|_\epsilon * r \rrbracket^I\right\}
  }{
    \tau : \{P_1 \lor P_2\}\ \mathsf{e}\ \{Q_1 \lor Q_2\}
  }\ \tau\text{: Def.}
}\ (\dagger)
$$

where (†) follows from the semantics of $\lor$; and (‡) denotes the application of the Disjunction rule.

□

Lemma 19 (Implication preservation)**.** *For all evaluation environments* ϵ*, interface functions* $I \in \mathcal{I}$*,* JSLogic$_{\mathbb{DOM}}$ *assertions* $P, Q \in$ JSAst$_{\mathbb{DOM}}$*, and set of* JSLogic$_{\mathbb{DOM}}$ *statesr* $\in$ JSLHeap$_{\mathbb{DOM}}$*:*

$$\vdash P \Rightarrow Q \qquad \Longrightarrow \qquad \llfloor |P|_\epsilon * r \rrbracket^I \subseteq \llfloor |Q|_\epsilon * r \rrbracket^I$$

Proof. Pick arbitrary ϵ, $I \in \mathcal{I}$, $P, Q \in$ JSAst$_{\mathbb{DOM}}$, $r \in$ JSLHeap$_{\mathbb{DOM}}$ and h such that

$$\vdash P \Rightarrow Q \tag{B.1}$$
$$h \in \llfloor |P|_\epsilon * r \rrbracket^I \tag{B.2}$$

We are then required to show:

$$h \in \llfloor |Q|_\epsilon * r \rrbracket^I \tag{B.3}$$

From (B.2) and the definitions of $\llfloor . \rrbracket$ and $|.|_\epsilon$ we know there exists h_p, h_r,

$\mathbf{h}_p$, $\mathbf{h}_r$, h_1, h_2 and I_1 such that:

$$\epsilon, (h_p, \mathbf{h}_p) \models P \text{ and } (h_r, \mathbf{h}_r) \in r \tag{B.4}$$

$$h_1 = \langle\!\langle h_p \circ h_r \rangle\!\rangle_{\text{JS}} \text{ and } h_2 \in \langle\!\langle \mathbf{h}_p \bullet \mathbf{h}_r \rangle\!\rangle_{\text{DOM}}^{I \uplus I_1} \tag{B.5}$$

$$\mathit{dom}(I_1^{\mathsf{in}}) = (\mathbf{h}_p \bullet \mathbf{h}_r)^{\mathsf{in}} \land \mathit{dom}(I_1^{\mathsf{out}}) = (\mathbf{h}_p \bullet \mathbf{h}_r)^{\mathsf{out}} \tag{B.6}$$

$$h = h_1 \circ h_2 \tag{B.7}$$

From (B.1) and (B.4) we know there exists $\mathbf{h}'$ such that

$$\mathbf{h}_p \approx \mathbf{h}' \text{ and } \epsilon, (h_p, \mathbf{h}') \models Q \tag{B.8}$$

On the other hand, from the definition of $\approx$ and (B.8) we have:

$$\mathbf{h}_p \bullet \mathbf{h}_r \approx \mathbf{h}' \bullet \mathbf{h}_r \tag{B.9}$$

From (B.5), (B.6), (B.9) and Lemma 20 we then know there exists I_2 such that:

$$h_2 \in \left\langle\!\!\left\langle \mathbf{h}' \bullet \mathbf{h}_r \right\rangle\!\!\right\rangle_{\text{DOM}}^{I \uplus I_2} \land \mathit{dom}(I_2^{\mathsf{in}}) = (\mathbf{h}' \bullet \mathbf{h}_r)^{\mathsf{in}} \land \mathit{dom}(I_2^{\mathsf{out}}) = (\mathbf{h}' \bullet \mathbf{h}_r)^{\mathsf{out}} \tag{B.10}$$

From (B.4), (B.8) and the definition of ι we know $(h_p \circ h_r, \mathbf{h}' \bullet \mathbf{h}_r) \in |Q|_\epsilon * r$. Consequently, from (B.5), (B.7), (B.10) and the definition of $\llbracket . \rrbracket$ we have:

$$h \in \llbracket |Q|_\epsilon * r \rrbracket^I$$

as required by (B.3). □

Lemma 20 (Abstract (de)allocation preservation). *For all $h \in$ JSLHeap, $\mathbf{h}_1, \mathbf{h}_2 \in \text{LHeap}_{\mathbb{DOM}}$, $I_1 \in \mathcal{I}$:*

$$\begin{aligned} h \in \langle\!\langle \mathbf{h}_1 \rangle\!\rangle_{\text{DOM}}^{I \uplus I_1} \land \mathit{dom}(I_1^{\mathsf{in}}) = \mathbf{h}_1^{\mathsf{in}} \land \mathit{dom}(I_1^{\mathsf{out}}) = \mathbf{h}_1^{\mathsf{out}} \land \mathbf{h}_1 \approx \mathbf{h}_2 \implies \\ \exists I_2.\ h \in \langle\!\langle \mathbf{h}_2 \rangle\!\rangle_{\text{DOM}}^{I \uplus I_2} \land \mathit{dom}(I_2^{\mathsf{in}}) = \mathbf{h}_2^{\mathsf{in}} \land \mathit{dom}(I_2^{\mathsf{out}}) = \mathbf{h}_2^{\mathsf{out}} \end{aligned}$$

Proof. Given a natural number n, let $\approx^n$ denote the number of abstract (de)allocation transitions taken in $\approx$. That is, $\mathbf{h}_1 \approx^0 \mathbf{h}_2$ when $\mathbf{h}_1 = \mathbf{h}_2$; $\mathbf{h}_1 \approx^1 \mathbf{h}_2$ when $\mathbf{h}_2$ can be obtained from $\mathbf{h}_1$ by a single abstract allocation/deallocation; and $\mathbf{h}_1 \approx^{n+1} \mathbf{h}_2 \Leftrightarrow \exists \mathbf{h}_3.\ \mathbf{h}_1 \approx^1 \mathbf{h}_3 \land \mathbf{h}_3 \approx^n \mathbf{h}_2$. Since

$\approx$ denotes the reflexive transitive closure of the abstract (de)allocation transitions, it suffices to show that for all $n \in \mathbb{N}$:

$$\begin{array}{l}\forall h \in \text{JSLHeap}, \mathbf{h}_1, \mathbf{h}_2 \in \text{LHeap}_{\mathbb{DOM}}, I_1 \in \mathfrak{I}. \\ \quad h \in \langle\!\langle \mathbf{h}_1 \rangle\!\rangle^{I \uplus I_1}_{\text{DOM}} \land \mathit{dom}(I_1^{\mathsf{in}}){=}\mathbf{h}_1^{\mathsf{in}} \land \mathit{dom}(I_1^{\mathsf{out}}){=}\mathbf{h}_1^{\mathsf{out}} \land \mathbf{h}_1 \approx^n \mathbf{h}_2 \implies \\ \qquad\qquad \exists I_2.\ h \in \langle\!\langle \mathbf{h}_2 \rangle\!\rangle^{I \uplus I_2}_{\text{DOM}} \land \mathit{dom}(I_2^{\mathsf{in}}){=}\mathbf{h}_2^{\mathsf{in}} \land \mathit{dom}(I_2^{\mathsf{out}}){=}\mathbf{h}_2^{\mathsf{out}}\end{array}$$

We proceed by induction on n.
The base case, when $\mathbf{h}_1 \approx^0 \mathbf{h}_2$, holds trivially since we have $\mathbf{h}_1{=}\mathbf{h}_2$ and can despatch the proof obligation by picking the witness $I_2{=}I_1$.

Inductive case ($n{=}k{+}1$)
Pick arbitrary $h \in \text{JSLHeap}$, $\mathbf{h}_1, \mathbf{h}_2 \in \text{LHeap}_{\mathbb{DOM}}$, $I_1 \in \mathfrak{I}$, such that:

$$h \in \langle\!\langle \mathbf{h}_1 \rangle\!\rangle^{I \uplus I_1}_{\text{DOM}} \land \mathit{dom}(I_1^{\mathsf{in}}){=}\mathbf{h}_1^{\mathsf{in}} \land \mathit{dom}(I_1^{\mathsf{out}}){=}\mathbf{h}_1^{\mathsf{out}} \land \mathbf{h}_1 \approx^{k+1} \mathbf{h}_2 \tag{B.11}$$

$$\begin{array}{l}\forall m \leq k.\ \forall h \in \text{JSLHeap}, \mathbf{h}_1, \mathbf{h}_2 \in \text{LHeap}_{\mathbb{DOM}}, I_1 \in \mathfrak{I}. \\ \quad h \in \langle\!\langle \mathbf{h}_1 \rangle\!\rangle^{I \uplus I_1}_{\text{DOM}} \land \mathit{dom}(I_1^{\mathsf{in}}){=}\mathbf{h}_1^{\mathsf{in}} \land \mathit{dom}(I_1^{\mathsf{out}}){=}\mathbf{h}_1^{\mathsf{out}} \land \mathbf{h}_1 \approx^m \mathbf{h}_2 \implies \\ \qquad\qquad \exists I_2.\ h \in \langle\!\langle \mathbf{h}_2 \rangle\!\rangle^{I \uplus I_2}_{\text{DOM}} \land \mathit{dom}(I_2^{\mathsf{in}}){=}\mathbf{h}_2^{\mathsf{in}} \land \mathit{dom}(I_2^{\mathsf{out}}){=}\mathbf{h}_2^{\mathsf{out}}\end{array} \tag{I.H.}$$

We are then required to show:

$$\exists I_2.\ h \in \langle\!\langle \mathbf{h}_2 \rangle\!\rangle^{I \uplus I_2}_{\text{DOM}} \land \mathit{dom}(I_2^{\mathsf{in}}){=}\mathbf{h}_2^{\mathsf{in}} \land \mathit{dom}(I_2^{\mathsf{out}}){=}\mathbf{h}_2^{\mathsf{out}} \tag{B.12}$$

From (B.11) and the definitions of $\langle\!\langle . \rangle\!\rangle_{\text{DOM}}$ and $\approx^{k+1}$ we know there exist $h_1, h_2, h_3, h_4 \in \text{JSLHeap}$, and $\mathbf{h}_3$ such that:

$$h{=}h_1 \circ h_2 \circ h_3 \circ h_4 \tag{B.13}$$

$$\begin{array}{l} h_1 \in \mathtt{Protos} \land h_2 \in \mathsf{Crust}\,(\mathbf{h}_1, I \uplus I_1) \\ \land\, h_3 \in \mathsf{H}\,(\mathbf{h}_1, I \uplus I_1) \land h_4 \in \mathsf{true}\end{array} \tag{B.14}$$

$$\mathbf{h}_1 \approx^1 \mathbf{h}_3 \land \mathbf{h}_3 \approx^k \mathbf{h}_2 \tag{B.15}$$

From the definition of the abstract allocation relation $\approx^1$ and (B.15), there are now two cases consider.

Case 1 There exists $a, \mathbf{x}, \mathbf{d}_1, \mathbf{d}_2, \mathbf{h}_0$ such that

$$\begin{array}{l}\mathbf{h}_1{=}[a \mapsto \mathbf{d}_1] \bullet [\mathbf{x} \mapsto \mathbf{d}_2] \bullet \mathbf{h}_0 \ \text{ and } \ \mathbf{x} \in \text{addr}(\mathbf{d}_1) \\ \mathbf{h}_3{=}[a \mapsto \mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2] \bullet \mathbf{h}_0\end{array}$$

Let $I_3{=}(I_1 \setminus \mathbf{x})$, where $(I_1 \setminus \mathbf{x})$ denotes I_1 with $\mathbf{x}$ removed from the domains of I_1^{in} and I_1^{out}. From (B.11), (B.14) and Lemma 21(B.24) we then have:

$$h_3 \in \mathsf{H}\left(\mathbf{h}_3, I \uplus I_3\right) \tag{B.16}$$

From the definitions of I_1, I_3, $\mathbf{h}_1$ and $\mathbf{h}_3$ we have:

$$dom(I_3^{\mathsf{in}}){=}\mathbf{h}_3^{\mathsf{in}} \land dom(I_3^{\mathsf{out}}){=}\mathbf{h}_3^{\mathsf{out}} \tag{B.17}$$

On the other hand, from the definitions of $\mathbf{h}_1$, $\mathbf{h}_3$ and the crust set (Def. 81) we have $\mathsf{cset}\left(\mathbf{h}_1, I \uplus I_1\right){=}\mathsf{cset}\left(\mathbf{h}_3, I \uplus I_3\right)$. Consequently, from the definition of crust child list (Def. 83) and crust (Def. 84) we have $\mathsf{Crust}\left(\mathbf{h}_1, I \uplus I_1\right){=}\mathsf{Crust}\left(\mathbf{h}_3, I \uplus I_3\right)$. As such, from (B.14) we have

$$h_2 \in \mathsf{Crust}\left(\mathbf{h}_3, I \uplus I_3\right) \tag{B.18}$$

From (B.13), (B.14), (B.16), (B.18) and the definition of $\langle\!\langle . \rangle\!\rangle_{\text{DOM}}$ we then have:

$$h \in \langle\!\langle \mathbf{h}_3 \rangle\!\rangle_{\text{DOM}}^{I \uplus I_3} \tag{B.19}$$

Lastly, from (I.H.), (B.15), (B.17) and (B.19) we have:

$$\exists I_2.\ h \in \langle\!\langle \mathbf{h}_2 \rangle\!\rangle_{\text{DOM}}^{I \uplus I_2} \land dom(I_2^{\mathsf{in}}){=}\mathbf{h}_2^{\mathsf{in}} \land dom(I_2^{\mathsf{out}}){=}\mathbf{h}_2^{\mathsf{out}}$$

as required by (B.12).

Case 2 There exists $a, \mathbf{x}, \mathbf{d}_1, \mathbf{d}_2, \mathbf{h}_0$ such that

$$\begin{aligned}
&\mathbf{h}_1{=}[a \mapsto \mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2] \bullet \mathbf{h}_0 \\
&\mathbf{h}_3{=}[a \mapsto \mathbf{d}_1] \bullet [\mathbf{x} \mapsto \mathbf{d}_2] \bullet \mathbf{h}_0 \quad \text{and} \quad \mathbf{x} \in \text{addr}(\mathbf{d}_1)
\end{aligned}$$

From (B.11), (B.14) and Lemma 21(B.24) we know there exist (L, u) and I_3 such that $I_3{=}(I_1 \uplus [\mathbf{x} \mapsto (L, u)])$, where $(I_1 \uplus [\mathbf{x} \mapsto (L, u)]) \triangleq (I_1^{\mathsf{in}} \uplus [\mathbf{x} \mapsto L], I_1^{\mathsf{out}} \uplus [\mathbf{x} \mapsto u])$, and:

$$h_3 \in \mathsf{H}\left(\mathbf{h}_3, I \uplus I_3\right) \tag{B.20}$$

From the definitions of I_1, I_3, $\mathbf{h}_1$ and $\mathbf{h}_3$ we have:

$$dom(I_3^{\mathsf{in}})=\mathbf{h}_3^{\mathsf{in}} \wedge dom(I_3^{\mathsf{out}})=\mathbf{h}_3^{\mathsf{out}} \tag{B.21}$$

On the other hand, from the definitions of $\mathbf{h}_1$, $\mathbf{h}_3$ and the crust set (Def. 81) we have $\mathsf{cset}\left(\mathbf{h}_1, I \uplus I_1\right)=\mathsf{cset}\left(\mathbf{h}_3, I \uplus I_3\right)$. Consequently, from the definition of crust child list (Def. 83) and crust (Def. 84) we have $\mathsf{Crust}\left(\mathbf{h}_1, I \uplus I_1\right)=\mathsf{Crust}\left(\mathbf{h}_3, I \uplus I_3\right)$. As such, from (B.14) we have

$$h_2 \in \mathsf{Crust}\left(\mathbf{h}_3, I \uplus I_3\right) \tag{B.22}$$

From (B.13), (B.14), (B.20), (B.22) and the definition of $\langle\!\langle . \rangle\!\rangle_{\text{DOM}}$ we then have:

$$h \in \langle\!\langle \mathbf{h}_3 \rangle\!\rangle_{\text{DOM}}^{I \uplus I_3} \tag{B.23}$$

Lastly, from (I.H.), (B.15), (B.21) and (B.23) we have:

$$\exists I_2.\ h \in \langle\!\langle \mathbf{h}_2 \rangle\!\rangle_{\text{DOM}}^{I \uplus I_2} \wedge dom(I_2^{\mathsf{in}})=\mathbf{h}_2^{\mathsf{in}} \wedge dom(I_2^{\mathsf{out}})=\mathbf{h}_2^{\mathsf{out}}$$

as required by (B.12). $\square$

Lemma 21 (Abstract (de)allocation preservation (auxiliary))**.** *For all* $\mathbf{h}_0$, $\mathbf{h}_1, \mathbf{h}_2 \in \text{LHeap}_{\mathbb{DOM}}$*,* $I \in \mathcal{I}$*,* $a \in \text{Add}_{\mathbb{DOM}}$*,* $\mathbf{d}_1, \mathbf{d}_2 \in \text{LData}_{\mathbb{DOM}}$ *and* $\mathbf{x} \in \text{AAdd}$*:*

$$\begin{aligned}\mathbf{h}_1=[a \mapsto \mathbf{d}_1] \bullet [\mathbf{x} \mapsto \mathbf{d}_2] \wedge \mathbf{h}_2=[a \mapsto \mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2] \wedge \mathbf{x} \in \text{addr}(\mathbf{d}_1) \\ \implies \exists (L,u).\ \mathsf{H}\left(\mathbf{h}_1 \bullet \mathbf{h}_0, I \uplus [\mathbf{x} \mapsto (L,u)]\right) = \mathsf{H}\left(\mathbf{h}_2 \bullet \mathbf{h}_0, I\right)\end{aligned} \tag{B.24}$$

where $I \uplus [\mathbf{x} \mapsto (L,u)] \triangleq (I^{\mathsf{in}} \uplus [\mathbf{x} \mapsto L], I^{\mathsf{out}} \uplus [\mathbf{x} \mapsto u])$*.*
For all $I \in \mathcal{I}$*,* $\iota \in \text{Inter}$*,* $\mathbf{d}_1, \mathbf{d}_2 \in \text{LData}_{\mathbb{DOM}}$ *and* $\mathbf{x} \in \text{AAdd}$*:*

$$\begin{aligned}\mathbf{x} \in \text{addr}(\mathbf{d}_1) \wedge (\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2) \quad \textit{defined} \implies \\ \mathsf{D}\left(\mathbf{d}_1\right)_I^{\iota} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) = \mathsf{D}\left(\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2\right)_I^{\iota}\end{aligned} \tag{B.25}$$

For all $\mathbf{h} \in \text{LHeap}_{\mathbb{DOM}}$*,* $I_0, I \in \mathcal{I}$*:*

$$\forall \mathbf{x} \in \mathbf{h}^{\mathsf{in}} \cup \mathbf{h}^{\mathsf{out}}.\ I(\mathbf{x}) \ \textit{defined} \implies \mathsf{H}\left(\mathbf{h}, I_0 \uplus I\right)=\mathsf{H}\left(\mathbf{h}, I\right) \tag{B.26}$$

For all $\mathbf{h} \in \textsc{LHeap}_{\mathbb{DOM}}$, $I \in \mathcal{I}$:

$$\mathsf{H}(\mathbf{h}, I) \neq \emptyset \implies \forall \mathbf{x} \in \mathbf{h}^{\mathsf{in}} \cup \mathbf{h}^{\mathsf{out}}.\ I(\mathbf{x})\ \textit{defined} \qquad \text{(B.27)}$$

Proof (B.24). Pick arbitrary $\mathbf{h}_0, \mathbf{h}_1, \mathbf{h}_2 \in \textsc{LHeap}_{\mathbb{DOM}}$, $I \in \mathcal{I}$, $a \in \textsc{Add}_{\mathbb{DOM}}$, $\mathbf{d}_1, \mathbf{d}_2 \in \textsc{LData}_{\mathbb{DOM}}$ and $\mathbf{x} \in \textsc{AAdd}$ such that

$$\mathbf{h}_1 = [a \mapsto \mathbf{d}_1] \bullet [\mathbf{x} \mapsto \mathbf{d}_2] \land \mathbf{h}_2 = [a \mapsto \mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2] \land \mathbf{x} \in \operatorname{addr}(\mathbf{d}_1) \qquad \text{(B.28)}$$

Let:

$$\iota \triangleq \begin{cases} ([d], \texttt{null}) & \text{if } a = \mathcal{R}_d \\ I(a) & \text{otherwise} \end{cases}$$

From the definitions of $\mathbf{h}_1$, $\mathbf{h}_2$ and $\mathbf{h}_0$ we have

$$\begin{array}{ll} (\mathbf{h}_2 \bullet \mathbf{h}_0)^{\mathsf{in}} = \mathbf{h}_0^{\mathsf{in}} \uplus (\{a\} \cap \textsc{AAdd}) & (\mathbf{h}_1 \bullet \mathbf{h}_0)^{\mathsf{in}} = (\mathbf{h}_2 \bullet \mathbf{h}_0)^{\mathsf{in}} \uplus \{\mathbf{x}\} \\ (\mathbf{h}_2 \bullet \mathbf{h}_0)^{\mathsf{out}} = \mathbf{h}_0^{\mathsf{out}} \uplus \operatorname{addr}(\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2) & (\mathbf{h}_1 \bullet \mathbf{h}_0)^{\mathsf{out}} = (\mathbf{h}_2 \bullet \mathbf{h}_0)^{\mathsf{out}} \uplus \{\mathbf{x}\} \end{array} \qquad \text{(B.29)}$$

There are two cases to consider.

Case 1. $\neg \forall \mathbf{y} \in (\mathbf{h}_2 \bullet \mathbf{h}_0)^{\mathsf{in}} \cup (\mathbf{h}_2 \bullet \mathbf{h}_0)^{\mathsf{out}}.\ I(\mathbf{y})$ defined
From the assumption of the case and Lemma 21(B.27) we have:

$$\mathsf{H}(\mathbf{h}_2 \bullet \mathbf{h}_0, I) = \emptyset$$

On the other hand, from (B.29), the assumption of the case and the definition of $I \uplus [\mathbf{x} \mapsto (L, u)]$ we then know

$$\neg \forall \mathbf{y} \in (\mathbf{h}_1 \bullet \mathbf{h}_0)^{\mathsf{in}} \cup (\mathbf{h}_1 \bullet \mathbf{h}_0)^{\mathsf{out}}.\ (I \uplus [\mathbf{x} \mapsto (L, u)])(\mathbf{y}) \text{ defined}$$

Consequently, from Lemma 21(B.27) we have $\mathsf{H}(\mathbf{h}_1 \bullet \mathbf{h}_0, I \uplus [\mathbf{x} \mapsto (L, u)]) = \emptyset$. As such we have,

$$\mathsf{H}(\mathbf{h}_2 \bullet \mathbf{h}_0, I) = \mathsf{H}(\mathbf{h}_1 \bullet \mathbf{h}_0, I \uplus [\mathbf{x} \mapsto (L, u)]) = \emptyset$$

as required.

Case 2. $\forall \mathbf{y} \in (\mathbf{h}_2 \bullet \mathbf{h}_0)^{\mathsf{in}} \cup (\mathbf{h}_2 \bullet \mathbf{h}_0)^{\mathsf{out}}.\ I(\mathbf{y})$ defined
From (B.29), the assumption of the case and the definition of $I \uplus [\mathbf{x} \mapsto$

$(L, u)]$ we have

$$\forall \mathbf{y} \in (\mathbf{h}_1 \bullet \mathbf{h}_0)^{\mathsf{in}} \cup (\mathbf{h}_2 \bullet \mathbf{h}_0)^{\mathsf{out}}.\ (I \uplus [\mathbf{x} \mapsto (L, u)])(\mathbf{y}) \text{ defined} \tag{B.30}$$

We then have:

$$\begin{array}{lll}
 & & \exists (L, u).\ \mathsf{H}\left(\mathbf{h}_1 \bullet \mathbf{h}_0, I \uplus [\mathbf{x} \mapsto (L, u)]\right) \\
 & = & \bigcup\limits_{(L,u) \in \textsc{Inter}} \mathsf{H}\left(\mathbf{h}_1 \bullet \mathbf{h}_0, I \uplus [\mathbf{x} \mapsto (L, u)]\right) \\
(\mathsf{H}\ \text{Def.}\ ) & = & \bigcup\limits_{(L,u) \in \textsc{Inter}} \begin{pmatrix} \mathsf{H}\left([a \mapsto \mathbf{d}_1], I \uplus [\mathbf{x} \mapsto (L, u)]\right) \\ * \mathsf{H}\left([\mathbf{x} \mapsto \mathbf{d}_2], I \uplus [\mathbf{x} \mapsto (L, u)]\right) \\ * \mathsf{H}\left(\mathbf{h}_0, I \uplus [\mathbf{x} \mapsto (L, u)]\right) \end{pmatrix} \\
(\mathsf{H}\ \text{Def.}\ ) & = & \bigcup\limits_{(L,u) \in \textsc{Inter}} \begin{pmatrix} \mathsf{D}\left(\mathbf{d}_1\right)^{\iota}_{I \uplus [\mathbf{x} \mapsto (L,u)]} \\ * \mathsf{H}\left([\mathbf{x} \mapsto \mathbf{d}_2], I \uplus [\mathbf{x} \mapsto (L, u)]\right) \\ * \mathsf{H}\left(\mathbf{h}_0, I \uplus [\mathbf{x} \mapsto (L, u)]\right) \end{pmatrix} \\
(*) & = & \bigcup\limits_{(L,u) \in \textsc{Inter}} \begin{pmatrix} \mathsf{D}\left(\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2\right)^{\iota}_{I \uplus [\mathbf{x} \mapsto (L,u)]} \\ * \mathsf{H}\left(\mathbf{h}_0, I \uplus [\mathbf{x} \mapsto (L, u)]\right) \end{pmatrix} \\
(\mathsf{H}\ \text{Def.}\ ) & = & \bigcup\limits_{(L,u) \in \textsc{Inter}} \begin{pmatrix} \mathsf{H}\left([a \mapsto \mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2], I \uplus [\mathbf{x} \mapsto (L, u)]\right) \\ * \mathsf{H}\left(\mathbf{h}_0, I \uplus [\mathbf{x} \mapsto (L, u)]\right) \end{pmatrix} \\
(**) & = & \mathsf{H}\left([a \mapsto \mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2], I\right) * \mathsf{H}\left(\mathbf{h}_0, I\right) \\
(\mathsf{H}\ \text{Def.}\ ) & = & \mathsf{H}\left([a \mapsto \mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2] \bullet \mathbf{h}_0, I\right) \\
 & = & \mathsf{H}\left(\mathbf{h}_2 \bullet \mathbf{h}_0, I\right)
\end{array}$$

where the step in $(*)$ follows from Lemma 21(B.25), and the step in $(**)$ follows from Lemma 21(B.26) and (B.30). □

Proof (B.25). Pick arbitrary $I \in \mathcal{I}, \iota \in \textsc{Inter}$, $\mathbf{d}_1, \mathbf{d}_2 \in \textsc{LData}_{\mathbb{DOM}}$ and $\mathbf{x} \in \textsc{AAdd}$ such that

$$\mathbf{x} \in \text{addr}(\mathbf{d}_1) \text{ and } (\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2) \quad \text{defined} \tag{B.31}$$

We proceed by induction on the structure of $\mathbf{d}_1$.

Case $\mathbf{d}_1 = \varnothing_\dagger \in \{\varnothing_e, \varnothing_f, \varnothing_a, \varnothing_{tf}, \varnothing_g\}$
This case holds vacuously since the $\mathbf{x} \in \text{addr}(\mathbf{d}_1)$ assumption in (B.31) is contradicted.

Case $\mathbf{d}_1{=}\mathbf{d}_3 \ddagger \mathbf{d}_4$ with $\ddagger \in \{\otimes, \odot, \oslash, \oplus\}$
Pick arbitrary L and u such that $\iota{=}(L, u)$. From (B.31) and the definition of $\diamond_{\mathbf{x}}$ we know that either $\mathbf{x} \in \mathrm{addr}(\mathbf{d}_3)$, $\mathbf{x} \notin \mathrm{addr}(\mathbf{d}_4)$ and $\mathbf{d}_3 \diamond_{\mathbf{x}} \mathbf{d}_2$ is defined, or $\mathbf{x} \notin \mathrm{addr}(\mathbf{d}_3)$, $\mathbf{x} \in \mathrm{addr}(\mathbf{d}_4)$ and $\mathbf{d}_4 \diamond_{\mathbf{x}} \mathbf{d}_2$ is defined.

Case 1. $\mathbf{x} \in \mathrm{addr}(\mathbf{d}_3)$, $\mathbf{x} \notin \mathrm{addr}(\mathbf{d}_4)$ and $\mathbf{d}_3 \diamond_{\mathbf{x}} \mathbf{d}_2$ defined
We then have:

$$\begin{array}{ll}
 & \mathsf{D}\left(\mathbf{d}_1\right)_I^{\iota} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
 & = \mathsf{D}\left(\mathbf{d}_3 \ddagger \mathbf{d}_4\right)_I^{(L,u)} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
(\mathsf{D}\left(.\right) \text{ Def. }) & = \exists L_1, L_2.\ L{\doteq}L_1{+\!\!+}L_2 * \mathsf{D}\left(\mathbf{d}_3\right)_I^{(L_1,u)} * \mathsf{D}\left(\mathbf{d}_4\right)_I^{(L_2,u)} \\
 & \quad * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
(\text{I.H.}) & = \exists L_1, L_2.\ L{\doteq}L_1{+\!\!+}L_2 * \mathsf{D}\left(\mathbf{d}_3 \diamond_{\mathbf{x}} \mathbf{d}_2\right)_I^{(L_1,u)} * \mathsf{D}\left(\mathbf{d}_4\right)_I^{(L_2,u)} \\
(\mathsf{D}\left(.\right) \text{ Def. }) & = \mathsf{D}\left(\left(\mathbf{d}_3 \diamond_{\mathbf{x}} \mathbf{d}_2\right) \ddagger \mathbf{d}_2\right)_I^{(L,u)} \\
(\diamond_{\mathbf{x}} \text{ Def. }) & = \mathsf{D}\left(\left(\mathbf{d}_3 \ddagger \mathbf{d}_4\right) \diamond_{\mathbf{x}} \mathbf{d}_2\right)_I^{(L,u)} \\
 & = \mathsf{D}\left(\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2\right)_I^{\iota}
\end{array}$$

as required.

Case 2. $\mathbf{x} \notin \mathrm{addr}(\mathbf{d}_3)$, $\mathbf{x} \in \mathrm{addr}(\mathbf{d}_4)$ and $\mathbf{d}_4 \diamond_{\mathbf{x}} \mathbf{d}_2$ defined
We then have:

$$\begin{array}{ll}
 & \mathsf{D}\left(\mathbf{d}_1\right)_I^{\iota} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
 & = \mathsf{D}\left(\mathbf{d}_3 \ddagger \mathbf{d}_4\right)_I^{(L,u)} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
(\mathsf{D}\left(.\right) \text{ Def. }) & = \exists L_1, L_2.\ L{\doteq}L_1{+\!\!+}L_2 * \mathsf{D}\left(\mathbf{d}_3\right)_I^{(L_1,u)} * \mathsf{D}\left(\mathbf{d}_4\right)_I^{(L_2,u)} \\
 & \quad * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
(\text{I.H.}) & = \exists L_1, L_2.\ L{\doteq}L_1{+\!\!+}L_2 * \mathsf{D}\left(\mathbf{d}_3\right)_I^{(L_1,u)} * \mathsf{D}\left(\mathbf{d}_4 \diamond_{\mathbf{x}} \mathbf{d}_2\right)_I^{(L_2,u)} \\
(\mathsf{D}\left(.\right) \text{ Def. }) & = \mathsf{D}\left(\mathbf{d}_3 \ddagger \left(\mathbf{d}_4 \diamond_{\mathbf{x}} \mathbf{d}_2\right)\right)_I^{(L,u)} \\
(\diamond_{\mathbf{x}} \text{ Def. }) & = \mathsf{D}\left(\left(\mathbf{d}_3 \ddagger \mathbf{d}_4\right) \diamond_{\mathbf{x}} \mathbf{d}_2\right)_I^{(L,u)} \\
 & = \mathsf{D}\left(\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2\right)_I^{\iota}
\end{array}$$

as required.

Case $\mathbf{d}_1{=}\mathrm{s}_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs}$
Pick arbitrary L and u such that $\iota{=}(L, u)$. From (B.31) and the definition of $\diamond_{\mathbf{x}}$ we know that either $\mathbf{x} \in \mathrm{addr}(\mathbf{as})$, $\mathbf{x} \notin \mathrm{addr}(\mathbf{f})$ and $\mathbf{as} \diamond_{\mathbf{x}} \mathbf{d}_2$ is defined, or $\mathbf{x} \notin \mathrm{addr}(\mathbf{as})$, $\mathbf{x} \in \mathrm{addr}(\mathbf{f})$ and $\mathbf{f} \diamond_{\mathbf{x}} \mathbf{d}_2$ is defined.

Case 1. $\mathbf{x} \in \mathrm{addr}(\mathbf{as})$, $\mathbf{x} \notin \mathrm{addr}(\mathbf{f})$ and $\mathbf{as} \diamond_{\mathbf{x}} \mathbf{d}_2$ defined
We then have:

$$\begin{array}{ll}
 & \mathsf{D}\left(\mathbf{d}_1\right)^{\iota}_{I} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
 & = \mathsf{D}\left(\mathrm{s}_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs}\right)^{(L,u)}_{I} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
(\mathsf{D}\left(.\right)\ \text{Def.}) & = L \doteq [n] * \exists \mathit{fid}, l_a, L_f, L_a.\ \mathit{fs} \doteq \{\mathit{fid}\} * \mathsf{ENode}\left(n, \mathrm{s}, l_a, \mathit{fid}, u\right) \\
 & \quad * \mathsf{FL}\left(\mathit{fid}, n, L_f\right) * \mathsf{TLs}\left(n, ts\right) * l_a \Mapsto \left\{\mathrm{s}'{:}m \mid (\mathrm{s}', m) \in L_a\right\} \\
 & \quad * \mathsf{D}\left(\mathbf{f}\right)^{(L_f,n)}_{I} * \mathsf{D}\left(\mathbf{as}\right)^{(L_a,\mathtt{null})}_{I} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
(\text{I.H.}) & = L \doteq [n] * \exists \mathit{fid}, l_a, L_f, L_a.\ \mathit{fs} \doteq \{\mathit{fid}\} * \mathsf{ENode}\left(n, \mathrm{s}, l_a, \mathit{fid}, u\right) \\
 & \quad * \mathsf{FL}\left(\mathit{fid}, n, L_f\right) * \mathsf{TLs}\left(n, ts\right) * l_a \Mapsto \left\{\mathrm{s}'{:}m \mid (\mathrm{s}', m) \in L_a\right\} \\
 & \quad * \mathsf{D}\left(\mathbf{f}\right)^{(L_f,n)}_{I} * \mathsf{D}\left(\mathbf{as} \diamond_{\mathbf{x}} \mathbf{d}_2\right)^{(L_a,\mathtt{null})}_{I} \\
(\mathsf{D}\left(.\right)\ \text{Def.}) & = \mathsf{D}\left(\mathrm{s}_n[\mathbf{as} \diamond_{\mathbf{x}} \mathbf{d}_2, \mathbf{f}]^{ts}_{fs}\right)^{(L,u)}_{I} \\
(\diamond_{\mathbf{x}}\ \text{Def.}) & = \mathsf{D}\left(\left(\mathrm{s}_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs}\right) \diamond_{\mathbf{x}} \mathbf{d}_2\right)^{(L,u)}_{I} \\
 & = \mathsf{D}\left(\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2\right)^{\iota}_{I}
\end{array}$$

as required.

Case 2. $\mathbf{x} \notin \mathrm{addr}(\mathbf{as})$, $\mathbf{x} \in \mathrm{addr}(\mathbf{f})$ and $\mathbf{f} \diamond_{\mathbf{x}} \mathbf{d}_2$ defined
We then have:

$$\begin{array}{ll}
 & \mathsf{D}\left(\mathbf{d}_1\right)^{\iota}_{I} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
 & = \mathsf{D}\left(\mathrm{s}_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs}\right)^{(L,u)}_{I} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
(\mathsf{D}\left(.\right)\ \text{Def.}) & = L \doteq [n] * \exists \mathit{fid}, l_a, L_f, L_a.\ \mathit{fs} \doteq \{\mathit{fid}\} * \mathsf{ENode}\left(n, \mathrm{s}, l_a, \mathit{fid}, u\right) \\
 & \quad * \mathsf{FL}\left(\mathit{fid}, n, L_f\right) * \mathsf{TLs}\left(n, ts\right) * l_a \Mapsto \left\{\mathrm{s}'{:}m \mid (\mathrm{s}', m) \in L_a\right\} \\
 & \quad * \mathsf{D}\left(\mathbf{f}\right)^{(L_f,n)}_{I} * \mathsf{D}\left(\mathbf{as}\right)^{(L_a,\mathtt{null})}_{I} * \mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}_2, I\right) \\
(\text{I.H.}) & = L \doteq [n] * \exists \mathit{fid}, l_a, L_f, L_a.\ \mathit{fs} \doteq \{\mathit{fid}\} * \mathsf{ENode}\left(n, \mathrm{s}, l_a, \mathit{fid}, u\right) \\
 & \quad * \mathsf{FL}\left(\mathit{fid}, n, L_f\right) * \mathsf{TLs}\left(n, ts\right) * l_a \Mapsto \left\{\mathrm{s}'{:}m \mid (\mathrm{s}', m) \in L_a\right\} \\
 & \quad * \mathsf{D}\left(\mathbf{f} \diamond_{\mathbf{x}} \mathbf{d}_2\right)^{(L_f,n)}_{I} * \mathsf{D}\left(\mathbf{as}\right)^{(L_a,\mathtt{null})}_{I}
\end{array}$$

(D (.) Def.) $\quad = \mathsf{D}\left(\mathrm{s}_n[\mathbf{as}, \mathbf{f} \diamond_{\mathbf{x}} \mathbf{d}_2]^{ts}_{fs}\right)^{(L,u)}_I$

($\diamond_{\mathbf{x}}$ Def.) $\quad = \mathsf{D}\left((\mathrm{s}_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs}) \diamond_{\mathbf{x}} \mathbf{d}_2\right)^{(L,u)}_I$

$\quad = \mathsf{D}\left(\mathbf{d}_1 \diamond_{\mathbf{x}} \mathbf{d}_2\right)^{\iota}_I$

as required.

Case $\mathbf{d}_1$=#text$_n$[s]$_{fs}$
This case holds vacuously since the $\mathbf{x} \in \mathrm{addr}(\mathbf{d}_1)$ assumption in (B.31) is contradicted.

Case $\mathbf{d}_1$=s$_n[\mathbf{tf}]_{fs}$ or $\mathbf{d}_1$=#doc$_n[\mathbf{d}]^{fs}_{\mathbf{tf}}$ & $\mathbf{g}$
These two cases are analogous to the element case (where $\mathbf{d}_1$=s$_n[\mathbf{as}, \mathbf{f}]^{ts}_{fs}$) and are omitted here. □

Proof (B.26). The proof of this part is by induction on the structure of $\mathsf{H}(.,.)$ and $\mathsf{D}(.)^{(.)}_{(.)}$ definitions. The proof is straightforward and we provide an informal argument instead.
When $\mathbf{h}$=$\mathbf{0}$, the result holds trivially from the definition of $\mathsf{H}(\mathbf{0}, .)$. When $\mathbf{h}$=$\mathbf{h}_1 \bullet \mathbf{h}_2$, the result follows from the inductive hypotheses for $\mathbf{h}_1$ and $\mathbf{h}_2$.

Lastly, when $\mathbf{h}$=$a \mapsto \mathbf{d}$, then $\mathsf{H}(\mathbf{h}, I)$=$\mathsf{D}(\mathbf{d})^{\iota}_I$ where the choice of ι is decided by whether a=$\mathcal{R}_d$ or $a \in$ AAdd. When a=$\mathcal{R}_d$ then we must show that $\mathsf{D}(\mathbf{d})^{([d],\texttt{null})}_{I_0 \uplus I}$=$\mathsf{D}(\mathbf{d})^{([d],\texttt{null})}_I$. On the other hand, when $a \in$ AAdd, from the assumption we have $I(\mathbf{x})$ is defined, and consequently $I(\mathbf{x})$=$I_0 \uplus I(\mathbf{x})$ and thus it suffices to show $\mathsf{D}(\mathbf{d})^{I(\mathbf{x})}_{I_0 \uplus I}$=$\mathsf{D}(\mathbf{d})^{I(\mathbf{x})}_I$. Either way, it suffices to show that for all ι we have $\mathsf{D}(\mathbf{d})^{\iota}_{I_0 \uplus I}$=$\mathsf{D}(\mathbf{d})^{\iota}_I$. From the assumption we know that the range of $\mathbf{h}$ (i.e. $\mathbf{h}^{\mathsf{out}}$) is in the domain of I, and thus for all abstract addresses in $\mathbf{h}^{\mathsf{out}}$ (i.e. the abstract addresses present in $\mathbf{d}$) we have $I(\mathbf{x})$=$I_0 \uplus I(\mathbf{x})$. Observe that the definition of $\mathsf{D}(\mathbf{d})^{\iota}_I$ restricts I only with respect to those abstract addresses in $\mathbf{d}$. Since for any abstract address $\mathbf{x}$ in $\mathbf{d}$ the $I(\mathbf{x})$ and $I_0 \uplus I(\mathbf{x})$ agree (i.e. $I(\mathbf{x})$=$I_0 \uplus I(\mathbf{x})$), it is then straightforward to show that $\mathsf{D}(\mathbf{d})^{\iota}_{I_0 \uplus I}$=$\mathsf{D}(\mathbf{d})^{\iota}_I$.

Proof (B.27). The proof of this part is by induction on the structure of the $\mathsf{H}(.,.)$ and $\mathsf{D}(.)^{(.)}_{(.)}$ definitions. The formal proof is straightforward and omitted here. We provide an informal argument instead.

Observe that the $\mathsf{H}(.,.)$ is defined piecemeal as the composition of each cell in $\mathbf{h}$. For any abstract address in the domain of $\mathbf{h}$ (i.e. in $\mathbf{h}^{\mathsf{in}}$), the $\mathsf{H}\left(\mathbf{x} \mapsto \mathbf{d}, I\right)$ is defined as $\mathsf{D}\left(\mathbf{d}\right)_I^{I(\mathbf{x})}$ and is thus undefined if $I(\mathbf{x})$ is not defined. Similarly, for any abstract address $\mathbf{x}$ in the range of $\mathbf{h}$ (i.e. in $\mathbf{h}^{\mathsf{out}}$), the translation contains $\mathsf{D}\left(\mathbf{x}\right)_I^{\iota}$ (for some ι) with the definition of $\mathsf{D}\left(\mathbf{x}\right)_I^{\iota}$ in turn stipulating that $I(\mathbf{x}){=}\iota$. As such, $\mathsf{D}\left(\mathbf{x}\right)_I^{\iota}$ (and consequently $\mathsf{H}\left(\mathbf{h}, I\right)$) is undefined if $I(\mathbf{x})$ is not defined.

Auxiliary Lemmata

The lemmas presented in this section are used in establishing the correctness of the DOM implementation in [44]. In the proof derivations given in the remaining of this chapter, given a proof rule (Rule), we write (Rule)* to denote the application of (Rule) combined with an application of the frame rule. Similarly, we write (Rule)$^{\Rightarrow}$ to denote the application of (Rule) combined with an application of the rule of consequence. Finally, we write (Rule)$^{\Rightarrow *}$ to denote the application of (Rule) combined with applications of the frame rule and the rule of consequence. Lastly, we write $\mathsf{vars}(\overline{\mathtt{x}_i : v_i}^{i=1\ldots n})_{Ls}$, to describe the variable store at the model level (without logical variables), in the scope chain captured by Ls.

To verify our implementation, we assume the following axioms for the JavaScript array library operations. The specification below is partial and does not capture all behaviours admitted by the array library. We have specified those cases used in our implementation only.

Definition 129 (Array axioms).

$$\frac{\{P\}\ \mathtt{e}\ \{Q * \mathbf{r} \doteq V'\} \qquad Q = R * \gamma(Ls, V', V) * \mathsf{array}(V, L)}{\{P\}\ \mathtt{e.length}\ \{Q * \mathbf{r} \doteq |L|\}}$$

$$\frac{\begin{array}{ll}\{P\}\ \mathtt{e}\ \{\mathbf{r} \doteq V_1' * Q * S_1\} & Q = \gamma(Ls_1, V_1', V_1) * \mathsf{array}(V_1, -) \\ \{Q * S_1\}\ \mathtt{e'}\ \{\mathbf{r} \doteq V_2' * R * S_2\} & R = \gamma(Ls_2, V_2', V_2) * \mathsf{array}(V_1, L)\end{array}}{\begin{array}{c}\left\{P\right\} \\ \mathtt{e.item(e')} \\ \left\{R * S_2 * \left[\left(\left(V_2 \dot{<} 0 \lor V_2 \dot{\geq} |L|\right) * \mathbf{r} \doteq \mathtt{null}\right) \lor \left(0 \dot{\leq} V_2 \dot{<} |L| * \mathbf{r} \doteq |L|^{V_2}\right)\right]\right\}\end{array}}$$

$$\frac{\begin{array}{l}\{P\}\ \mathsf{e}\ \{\mathbf{r}\dot{=}V_1' * Q * S_1\} \qquad Q = \gamma(Ls_1, V_1', V_1) * \mathsf{array}(V_1, -)\\ \{Q * S_1\}\ \mathsf{e'}\ \{\mathbf{r}\dot{=}V_2' * R * S_2\}\\ R = \gamma(Ls_2, V_2', V_2) * \mathsf{array}(V_1, L_1{+\!\!+}[V_2]{+\!\!+}L_2) \qquad V_2 \notin L_1\end{array}}{\{P\}\ \mathtt{e.indexOf(e')}\ \{R * S_2 * \mathbf{r}\dot{=}|L_1|\}}$$

$$\frac{\begin{array}{l}\{P\}\ \mathsf{e}\ \{\mathbf{r}\dot{=}V_1' * Q * S_1\} \qquad Q = \gamma(Ls_1, V_1', V_1) * \mathsf{array}(V_1, -)\\ \{Q * S_1\}\ \mathsf{e'}\ \{\mathbf{r}\dot{=}V_2' * R * S_2\}\\ R = \gamma(Ls_2, V_2', V_2) * \mathsf{array}(V_1, L) \qquad V \notin L\end{array}}{\{P\}\ \mathtt{e.indexOf(e')}\ \{R * S_2 * \mathbf{r}\dot{=}-1\}}$$

$$\frac{\begin{array}{l}\{P\}\ \mathsf{e}\ \{\mathbf{r}\dot{=}V_1' * R_1 * S_1\} \qquad R_1 = \gamma(Ls_1, V_1', V_1) * \mathsf{array}(V_1, -)\\ \{R_1 * S_1\}\ \mathsf{e}_1\ \{\mathbf{r}\dot{=}V_2' * R_2\} \qquad \{R_2\}\ \mathsf{e}_2\ \{\mathbf{r}\dot{=}V_3' * R_3 * R_4 * S_3\}\\ R_3 = (\gamma(Ls_2, V_2', V_2) \uplus \gamma(Ls_3, V_3', V_3))\\ R_4 = \mathsf{array}(V_1, L_1{+\!\!+}L_2{+\!\!+}L_3) * |L_1|\dot{=}V_1 * |L_2|\dot{=}V_2\\ R_5 = \exists \mathrm{L}.\ \mathsf{array}(\mathrm{L}, L_2) * \mathsf{array}(V_1, L_1{+\!\!+}L_3)\end{array}}{\{P\}\ \mathtt{e.splice(e_1, e_2)}\ \{R_3 * R_5 * S_3 * \mathbf{r}\dot{=}\mathrm{L}\}}$$

Lemma 22 (Function call (auxiliary)). *For all* $\mathtt{n}$, $\mathtt{x}_1 \dots \mathtt{x}_m$, $\mathtt{x}_1' \dots \mathtt{x}_m'$, $\mathtt{y}_1 \dots \mathtt{y}_o$, $\mathtt{f}$, n, $x_1 \dots x_m$, $y_1 \dots y_o$, e, Ls, fid, e, s, l_a, u, l_{P}, *and* l_{P}, *and for all* $p_i \in \{p_1, \dots, p_6\}$,

if $p_i \iff p_i' * (p_i' \multimap\!\!* \; p_i)$
then the following proof rule is sound

$$\frac{\left\{p_i * \exists l.\ \mathbf{l}\dot{=}l{:}Ls * r\right\}\ \mathsf{e}\ \left\{q * \exists l.\ \mathbf{l}\dot{=}l{:}Ls\right\}}{\{vs * p_i\}\ \mathtt{n.f(x_1 \dots x_m)}\ \{vs * q * \mathsf{true}\}}$$

where

$$vs \triangleq \mathsf{vars}(\mathtt{n} : n, \overline{\mathtt{x}_j : x_j}^{1\dots m}, \overline{\mathtt{y}_j : y_j}^{1\dots o})_{Ls}$$

$$p_1 \triangleq \mathsf{DNode}\,(n, fid, e) * \mathtt{Protos} * F$$

$$p_2 \triangleq \mathsf{ENode}\,(n, \mathrm{s}, l_a, fid, u) * \mathtt{Protos} * F$$

$$p_3 \triangleq \mathsf{TNode}\,(n, \mathrm{s}, fid, u) * \mathtt{Protos} * F$$

$$p_4 \triangleq \mathsf{ANode}\,(n, \mathrm{s}, fid) * \mathtt{Protos} * F$$

$$p_5 \triangleq \mathsf{FL}\,(n, u, L) * \mathtt{Protos} * F$$

$$p_6 \triangleq \mathsf{TL}\,(n, \mathrm{s}, u) * \mathtt{Protos} * F$$

$$r \triangleq l \Mapsto \left\{\overline{\mathtt{x}'_j : x_j}^{1\ldots m}, @\mathit{this}: n, @\mathit{proto}: \mathtt{null}, \mathsf{defs}(\{\mathtt{x}'_1 \ldots \mathtt{x}'_m\}, l, \mathtt{e})\right\}$$

$$\begin{aligned} p'_i \triangleq\ & (n, @\mathit{this}) \mapsto \varnothing * l_f \mapsto \{@\mathit{body}: \lambda \mathtt{x}'_1 \ldots \mathtt{x}'_m.\mathtt{e}, @\mathit{scope}: Ls\} \\ & * \mathscr{F}([n, l_{\mathrm{P}}, l_{\mathrm{G}}], \mathtt{f}, l_f) \end{aligned}$$

and for all a, b, L

$$\begin{aligned} \mathscr{F}([a], \mathtt{f}, l_f) &\triangleq (a, \mathtt{f}) \mapsto l_f \\ \mathscr{F}(a{:}(b{:}L), \mathtt{f}, l_f) &\triangleq (a, \mathtt{f}) \mapsto l_f \vee (a \mapsto \{\mathtt{f}: \varnothing, @\mathit{proto}: b\} * \mathscr{F}(b{:}L, \mathtt{f}, l_f)) \end{aligned}$$

Proof.

$$\dfrac{(\dagger) \quad (\ddagger) \quad \dfrac{}{\left\{p_i * \exists l.\ \mathbf{l}\dot{=}l{:}Ls * r\right\}\ \mathtt{e}\ \left\{q * \exists l.\ \mathbf{l}\dot{=}l{:}Ls\right\}}}{\{vs * p_i\}\ \mathtt{n.f(x_1\ .\ .\ .\ x_m))}\ \{vs * q\}}\ (\text{Function Call})^{\Rightarrow *}$$

where

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{vs\}\ \mathtt{n}\ \{\gamma(Ls_0, L_0.\mathbf{n}, n) * [\gamma(Ls_0, L_0.\mathbf{n}, n) \mathrel{-\!\!*} vs] * \mathbf{r}\dot{=}L_0.\mathbf{n}\}}\ (\text{Variable})^{*}}{\{vs * p_i\}\ \mathtt{n.f}\ \{vs * p_i * \mathbf{r}\dot{=}n.\mathtt{f}\}}\ (\text{Member Access})^{\Rightarrow *}}{\{vs * p_i\}\ \mathtt{n.f}\ \{vs * p'_i * (p'_i \mathrel{-\!\!*} p_i) * \mathbf{r}\dot{=}n.\mathtt{f}\}}\ (\text{Consequence})}{(\dagger)}$$

and for $j \in \{1 \ldots m\}$

$$\dfrac{\dfrac{\dfrac{}{\{vs\}\ \mathtt{x}_j\ \{\gamma(Ls_j, L_j.\mathbf{x}_j, x_j) * [\gamma(Ls_j, L_j.\mathbf{x}_j, x_j) \mathrel{-\!\!*} vs] * \mathbf{r}\dot{=}L_j.\mathbf{x}\}}\ (\text{Variable})^{*}}{\begin{gathered}\left\{vs * p_i\right\} \\ \mathtt{x}_j \\ \left\{\gamma(Ls_j, L_j.\mathbf{x}_j, x_j) * [\gamma(Ls_j, L_j.\mathbf{x}_j, x_j) \mathrel{-\!\!*} vs] * p_i * \mathbf{r}\dot{=}L_j.\mathbf{x}_j\right\}\end{gathered}}\ (\text{Frame})}{(\ddagger)}$$

$\square$

Lemma 23 (Axiom correctness). *For all $(P, C, Q) \in \textsc{Axiom}_{\mathbb{DOM}}$,*

$$\tau : \{P\}\ C\ \{Q\}$$

Proof. By structural induction on the DOM axioms. We provide a proof of one of the axioms of `r := n.firstChild()` and `n.removeChild(o)` here. The proof of other cases are analogous and omitted here.

Case `r := n.firstChild()`
We prove the correctness of one of the `r := n.firstChild()` axioms; the correctness proof of the remaining axioms is analogous.
Given an evaluation environment $\epsilon{=}(\Gamma, Ls) \in \textsc{Env}$, pick an arbitrary $s \in \mathcal{P}(\text{JSLHeap}_{\mathbb{DOM}})$ and $I \in \mathcal{I}$ such that:

$$\Gamma(\alpha){=}\mathbf{x} \qquad \Gamma(\beta){=}\mathbf{y} \qquad \Gamma(\gamma){=}\mathbf{z} \qquad \Gamma(\textsc{n}){=}n \qquad \Gamma(\textsc{s}){=}\mathrm{s} \qquad \Gamma(\textsc{s}'){=}\mathrm{s}'$$

$$\Gamma(\textsc{m}){=}m \qquad \Gamma(\textsc{f}){=}fs \qquad \Gamma(\textsc{e}){=}ts \qquad \Gamma(\textsc{f}'){=}fs' \qquad \Gamma(\textsc{r}){=}r$$

and let

$$p \triangleq \left\llbracket \left| \mathsf{vars}(\mathtt{n} : \textsc{n}, \mathtt{r} : \textsc{r}) * \alpha \mapsto \textsc{s}_{\textsc{n}}[\beta, \#\text{text}_{\textsc{m}}[\textsc{s}']_{\textsc{f}'} \otimes \gamma]_{\textsc{f}}^{\textsc{e}} \right|_{\epsilon} * s \right\rrbracket^{I}$$
$$q \triangleq \left\llbracket \left| \mathsf{vars}(\mathtt{n} : \textsc{n}, \mathtt{r} : \textsc{m}) * \alpha \mapsto \textsc{s}_{\textsc{n}}[\beta, \#\text{text}_{\textsc{m}}[\textsc{s}']_{\textsc{f}'} \otimes \gamma]_{\textsc{f}}^{\textsc{e}} \right|_{\epsilon} * s \right\rrbracket^{I}$$

It then suffices to show that:

$$\{p\}\ \mathtt{r = n.firstChild()}\ \{q\}$$

From the definition of p we have:

$$p = \left\{ \llbracket |\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{s_N}[\beta, \#\mathrm{text_M}[\mathrm{s}']_{\mathrm{F}'} \otimes \gamma]^{\mathrm{E}}_{\mathrm{F}}|_{\epsilon} * s \rrbracket^{I} \right\}$$

// From the definition of $\llbracket . \rrbracket^{I}$ and Lemma 18

$$\left\{ \begin{array}{l}
\mathsf{vars}(\mathtt{n} : n, \mathtt{r} : r)_{Ls} * \mathsf{Protos} * \mathsf{true} \\
* \bigcup_{(h,\mathbf{h}) \in s} \langle\!\langle h \rangle\!\rangle_{\mathrm{JS}} * \mathsf{H}\,(\mathbf{h}, I) * \mathsf{Crust}\left(\mathbf{x} \mapsto \mathrm{s}_n[\mathbf{y}, \#\mathrm{text}_m[\mathrm{s}']_{fs'} \otimes \mathbf{z}]^{ts}_{fs} \bullet \mathbf{h}, I\right) \\
* \exists u, l_a, l_{cn}, L_a, L_z.\ I(\mathbf{x}) \doteq ([n], u) * fs \doteq \{l_{cn}\} * \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) \\
* \mathsf{FL}\,(l_{cn}, n, [m] {+\!\!+} L_z) * \mathsf{TLs}\,(n, ts) * l_a \Mapsto \{\mathrm{s}'' : l \mid (\mathrm{s}'', l) \in L_a\} \\
* \mathsf{D}\,(\mathbf{y})^{(L_a, \mathtt{null})}_{I} * \mathsf{D}\left(\#\mathrm{text}_m[\mathrm{s}']_{fs'} \otimes \mathbf{z}\right)^{([m] {+\!\!+} L_z, n)}_{I}
\end{array} \right\}$$

$$= \left\{ \begin{array}{l}
\mathsf{vars}(\mathtt{n} : n, \mathtt{r} : r)_{Ls} * \exists l_{\mathrm{E}}, l_{\mathrm{N}}.\ (\mathsf{EProto} * \mathsf{NProto} \mathrel{-\!\!*} \mathsf{Protos}) * \mathsf{true} \\
* \bigcup_{(h,\mathbf{h}) \in s} \langle\!\langle h \rangle\!\rangle_{\mathrm{JS}} * \mathsf{H}\,(\mathbf{h}, I) * \mathsf{Crust}\left(\mathbf{x} \mapsto \mathrm{s}_n[\mathbf{y}, \#\mathrm{text}_m[\mathrm{s}']_{fs'} \otimes \mathbf{z}]^{ts}_{fs} \bullet \mathbf{h}, I\right) \\
* \exists u, l_a, l_{cn}, L_a, L_z.\ I(\mathbf{x}) \doteq ([n], u) * fs \doteq \{l_{cn}\} * \mathsf{FL}\,(l_{cn}, n, [m] {+\!\!+} L_z) \\
* \mathsf{TLs}\,(n, ts) * l_a \Mapsto \{\mathrm{s}'' : l \mid (\mathrm{s}'', l) \in L_a\} \\
* \mathsf{D}\,(\mathbf{y})^{(L_a, \mathtt{null})}_{I} * \mathsf{D}\left(\#\mathrm{text}_m[\mathrm{s}']_{fs'} \otimes \mathbf{z}\right)^{([m] {+\!\!+} L_z, n)}_{I} \\
* (n, @\mathit{this}) \mapsto \varnothing * (n, \mathtt{firstChild}) \mapsto \varnothing * (n, @\mathit{proto}) \mapsto l_{\mathrm{E}} \\
* ((n, @\mathit{this}) \mapsto \varnothing * (n, \mathtt{firstChild}) \mapsto \varnothing * (n, @\mathit{proto}) \mapsto l_{\mathrm{E}} \\
\qquad \mathrel{-\!\!*} \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u)) \\
* (l_{\mathrm{E}}, \mathtt{firstChild}) \mapsto \varnothing * (l_{\mathrm{E}}, @\mathit{proto}) \mapsto l_{\mathrm{N}} \\
* ((l_{\mathrm{E}}, \mathtt{firstChild}) \mapsto \varnothing * (l_{\mathrm{E}}, @\mathit{proto}) \mapsto l_{\mathrm{N}} \mathrel{-\!\!*} \mathsf{EProto}) \\
* (l_{\mathrm{N}}, \mathtt{firstChild}) \mapsto l_{\mathrm{FC}} * l_{\mathrm{FC}} \mapsto \{@\mathit{body} : \lambda.\mathsf{e}_{\mathrm{FC}}, @\mathit{scope} : Ls\} \\
* ((l_{\mathrm{N}}, \mathtt{firstChild}) \mapsto l_{\mathrm{FC}} * l_{\mathrm{FC}} \mapsto \{@\mathit{body}{:}\lambda.\mathsf{e}_{\mathrm{FC}}, @\mathit{scope}{:}Ls\} \mathrel{-\!\!*} \mathsf{NProto})
\end{array} \right\}$$

$$= \left\{ \begin{array}{l}
\mathsf{vars}(\mathtt{n} : n, \mathtt{r} : r)_{Ls} * \exists l_{\mathrm{E}}, l_{\mathrm{N}}.\ (\mathsf{EProto} * \mathsf{NProto} \mathrel{-\!\!*} \mathsf{Protos}) * \mathsf{true} \\
* \bigcup_{(h,\mathbf{h}) \in s} \langle\!\langle h \rangle\!\rangle_{\mathrm{JS}} * \mathsf{H}\,(\mathbf{h}, I) * \mathsf{Crust}\left(\mathbf{x} \mapsto \mathrm{s}_n[\mathbf{y}, \#\mathrm{text}_m[\mathrm{s}']_{fs'} \otimes \mathbf{z}]^{ts}_{fs} \bullet \mathbf{h}, I\right) \\
* \exists u, l_a, l_{cn}, L_a, L_z.\ I(\mathbf{x}) \doteq ([n], u) * fs \doteq \{l_{cn}\} * \mathsf{FL}\,(l_{cn}, n, [m] {+\!\!+} L_z) \\
* \mathsf{TLs}\,(n, ts) * l_a \Mapsto \{\mathrm{s}'' : l \mid (\mathrm{s}'', l) \in L_a\} \\
* \mathsf{D}\,(\mathbf{y})^{(L_a, \mathtt{null})}_{I} * \mathsf{D}\left(\#\mathrm{text}_m[\mathrm{s}']_{fs'} \otimes \mathbf{z}\right)^{([m] {+\!\!+} L_z, n)}_{I} \\
\mathscr{F}\,([n, l_{\mathrm{E}}, l_{\mathrm{N}}], \mathtt{firstChild}, l_{\mathrm{FC}}) * l_{\mathrm{FC}} \mapsto \{@\mathit{body} : \lambda.\mathsf{e}_{\mathrm{FC}}, @\mathit{scope} : Ls\} \\
* (\mathscr{F}\,([n, l_{\mathrm{E}}, l_{\mathrm{N}}], \mathtt{firstChild}, l_{\mathrm{FC}}) * l_{\mathrm{FC}} \mapsto \{@\mathit{body} : \lambda.\mathsf{e}_{\mathrm{FC}}, @\mathit{scope} : Ls\} \\
\qquad \mathrel{-\!\!*} \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) * \mathsf{EProto} * \mathsf{NProto})
\end{array} \right\}$$

$$= \left\{ \begin{array}{l}
\exists l_{\mathrm{E}}, l_{\mathrm{N}}.\ \mathsf{vars}(\mathtt{n}{:}n, \mathtt{r}{:}r)_{Ls} * \mathscr{F}\,([n, l_{\mathrm{E}}, l_{\mathrm{N}}], \mathtt{firstChild}, l_{\mathrm{FC}}) \\
* l_{\mathrm{FC}} \mapsto \{@\mathit{body}{:}\lambda.\mathsf{e}_{\mathrm{FC}}, @\mathit{scope}{:}Ls\} \\
* (\mathsf{vars}(\mathtt{n}{:}n, \mathtt{r}{:}r)_{Ls} * \mathscr{F}\,([n, l_{\mathrm{E}}, l_{\mathrm{N}}], \mathtt{firstChild}, l_{\mathrm{FC}}) \\
\quad * l_{\mathrm{FC}} \mapsto \{@\mathit{body} : \lambda.\mathsf{e}_{\mathrm{FC}}, @\mathit{scope} : Ls\} \\
\qquad \mathrel{-\!\!*} \llbracket |\mathsf{vars}(\mathtt{n} : \mathrm{N}, \mathtt{r} : \mathrm{R}) * \alpha \mapsto \mathrm{s_N}[\beta, \#\mathrm{text_M}[\mathrm{s}']_{\mathrm{F}'} \otimes \gamma]^{\mathrm{E}}_{\mathrm{F}}|_{\epsilon} * s \rrbracket^{I})
\end{array} \right\} \qquad \text{(B.32)}$$

We then proceed as follows:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{c}\left\{p * \exists l.\, \mathbf{l}\dot{=}l{:}Ls * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}\}\right\}\\ \mathtt{this.childNodes.\_\_contents\_\_.length}\\ \left\{p * \mathbf{r}\dot{=}m * \exists l.\, \mathbf{l}\dot{=}l{:}Ls * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}\}\right\}\end{array}}\ \text{array axioms, (B.32)}}{\begin{array}{c}\left\{p\right\}\\ \mathtt{n.firstChild()}\\ \left\{p * \mathbf{r}\dot{=}m * \exists l.\, l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}\}\right\}\end{array}}\ \text{Lemma 22,(B.32)}}{\{p\}\ \mathtt{n.firstChild()}\ \{p * \mathbf{r}\dot{=}m * \mathsf{true}\}}\ \text{(Consequence)}}{\{p\}\ \mathtt{n.firstChild()}\ \{p * \mathbf{r}\dot{=}m\}}\ \llbracket . \rrbracket\ \text{Def.}}{\{p\}\ \mathtt{r\,=\,n.firstChild()}\ \{q\}}\ \text{(Assign Local), (B.32)}
$$

Case `n.removeChild(o)`
Given an evaluation environment $\epsilon{=}(\Gamma, Ls) \in \textsc{Env}$, pick an arbitrary $s \in \mathcal{P}\left(\text{JSLHEAP}_{\mathbb{DOM}}\right)$ and $I \in \mathcal{I}$, such that:

$$
\Gamma(\alpha){=}\mathbf{x} \qquad \Gamma(\beta){=}\mathbf{y} \qquad \Gamma(\gamma_1){=}\mathbf{z}_1 \qquad \Gamma(\gamma_2){=}\mathbf{z}_2 \qquad \Gamma(\delta){=}w \qquad \Gamma(\mathrm{N}){=}n
$$

$$
\Gamma(\mathrm{S}){=}\mathrm{s} \qquad \Gamma(\mathrm{S}'){=}\mathrm{s}' \qquad \Gamma(\mathrm{O}){=}o \qquad \Gamma(\mathrm{F}){=}\mathit{fs} \qquad \Gamma(\mathrm{E}){=}\mathit{ts} \qquad \Gamma(\mathrm{F}'){=}\mathit{fs}'
$$

and let

$$
\begin{aligned}
p &\triangleq \left\llbracket \left|\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma_1 \otimes \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \varnothing_g\right|_{\epsilon} * s \right\rrbracket^{I}\\
q' &\triangleq \left\llbracket \left|\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{R}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma_1 \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'}\right|_{\epsilon} * s \right\rrbracket^{I}\\
q &\triangleq \left\llbracket \left|\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{O}) * \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\beta, \gamma_1 \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}} * \delta \mapsto \#\mathrm{text}_{\mathrm{O}}[\mathrm{S}']_{\mathrm{F}'}\right|_{\epsilon} * s \right\rrbracket^{I}
\end{aligned}
$$

It then suffices to show that:

$$
\{p\}\ \mathtt{r\,=\,n.removeChild(o)}\ \{q\}
$$

From the definition of p we have:

$$p = \left\{ \llbracket |\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{R})_{Ls} * \alpha \mapsto \mathrm{s}_{\mathrm{N}}[\beta, \gamma_1 \otimes \#\mathrm{text}_{\mathrm{O}}[\mathrm{s}']_{\mathrm{F}'} \otimes \gamma|_2]^{\mathrm{E}}_{\mathrm{F}}|_{\epsilon} * s \rrbracket^{I} \right\}$$

// From the definition of $\llbracket . \rrbracket^{I}$ and Lemma 18

$$\left\{ \begin{array}{l} \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{R})_{Ls} * \mathsf{Protos} * \mathsf{true} \\ * \bigcup_{(h,\mathbf{h}) \in s} \langle\!\langle h \rangle\!\rangle_{\mathrm{JS}} * \mathsf{H}\,(\mathbf{h}, I) * \mathsf{Crust}\left(\mathbf{x} \mapsto \mathrm{s}_n[\mathbf{y}, \mathbf{z}_1 \otimes \#\mathrm{text}_o[\mathrm{s}']_{fs'} \otimes \mathbf{z}_2]^{ts}_{fs} \bullet \mathbf{h}, I\right) \\ * \exists u, l_a, l_{cn}, L_a, L_1, L_2.\ I(\mathbf{x}) \doteq ([n], u) * fs \doteq \{l_{cn}\} * \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) \\ * \mathsf{FL}\,(l_{cn}, n, L_1 {+\!\!+} [o] {+\!\!+} L_2) * \mathsf{TLs}\,(n, ts) * l_a \Mapsto \{\mathrm{s}'' : l \mid (\mathrm{s}'', l) \in L_a\} \\ * \mathsf{D}\,(\mathbf{y})^{(L_a, \mathtt{null})}_{I} * \mathsf{D}\left(\mathbf{z}_1 \otimes \#\mathrm{text}_o[\mathrm{s}']_{fs'} \otimes \mathbf{z}_2\right)^{(L_1 {+\!\!+} [o] {+\!\!+} L_2, n)}_{I} * o \dot{\notin} L_1 \end{array} \right\}$$

$$= \left\{ \begin{array}{l} \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{R})_{Ls} * \exists l_{\mathrm{E}}, l_{\mathrm{N}}.\ (n, @\mathit{this}) \mapsto \varnothing \\ * (n, \mathtt{removeChild}) \mapsto \varnothing * (n, @\mathit{proto}) \mapsto l_{\mathrm{E}} \\ * (l_{\mathrm{E}}, \mathtt{removeChild}) \mapsto \varnothing * (l_{\mathrm{E}}, @\mathit{proto}) \mapsto l_{\mathrm{N}} \\ * (l_{\mathrm{N}}, \mathtt{removeChild}) \mapsto l_{\mathrm{RM}} * l_{\mathrm{RM}} \mapsto \{@\mathit{body} : \lambda.\mathsf{e}_{\mathrm{RM}}, @\mathit{scope} : Ls\} \\ * (\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{R})_{Ls} * (n, @\mathit{this}) \mapsto \varnothing \\ * (n, \mathtt{removeChild}) \mapsto \varnothing * (n, @\mathit{proto}) \mapsto l_{\mathrm{E}} \\ \quad * (l_{\mathrm{E}}, \mathtt{removeChild}) \mapsto \varnothing * (l_{\mathrm{E}}, @\mathit{proto}) \mapsto l_{\mathrm{N}} \\ \quad * (l_{\mathrm{N}}, \mathtt{removeChild}) \mapsto l_{\mathrm{RM}} * l_{\mathrm{RM}} \mapsto \{@\mathit{body} : \lambda.\mathsf{e}_{\mathrm{RM}}, @\mathit{scope} : Ls\} \\ \qquad -\!\!* \llbracket |\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{R}) * \alpha \mapsto \mathrm{s}_{\mathrm{N}}[\beta, \gamma_1 \otimes \#\mathrm{text}_{\mathrm{O}}[\mathrm{s}']_{\mathrm{F}'} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}}|_{\epsilon} * s \rrbracket^{I}) \end{array} \right\} \tag{B.33}$$

Using a similar rewrite, we have:

$$p = \left\{ \begin{array}{l} \mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{R})_{Ls} * \exists l_{\mathrm{E}}, l_{\mathrm{N}}.\ (n, @\mathit{this}) \mapsto \varnothing \\ * (n, \mathtt{\_\_removeChild\_\_}) \mapsto \varnothing * (n, @\mathit{proto}) \mapsto l_{\mathrm{E}} \\ * (l_{\mathrm{E}}, \mathtt{\_\_removeChild\_\_}) \mapsto \varnothing * (l_{\mathrm{E}}, @\mathit{proto}) \mapsto l_{\mathrm{N}} \\ * (l_{\mathrm{N}}, \mathtt{\_\_removeChild\_\_}) \mapsto l_{\mathrm{rm}} * l_{\mathrm{rm}} \mapsto \{@\mathit{body} : \lambda.\mathsf{e}_{\mathrm{rm}}, @\mathit{scope} : Ls\} \\ * (\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{R})_{Ls} * (n, @\mathit{this}) \mapsto \varnothing \\ * (n, \mathtt{\_\_removeChild\_\_}) \mapsto \varnothing * (n, @\mathit{proto}) \mapsto l_{\mathrm{E}} \\ \quad * (l_{\mathrm{E}}, \mathtt{\_\_removeChild\_\_}) \mapsto \varnothing * (l_{\mathrm{E}}, @\mathit{proto}) \mapsto l_{\mathrm{N}} \\ \quad * (l_{\mathrm{N}}, \mathtt{\_\_removeChild\_\_}) \mapsto l_{\mathrm{rm}} * l_{\mathrm{rm}} \mapsto \{@\mathit{body} : \lambda.\mathsf{e}_{\mathrm{rm}}, @\mathit{scope} : Ls\} \\ \qquad -\!\!* \llbracket |\mathsf{vars}(\mathtt{n}{:}\mathrm{N}, \mathtt{o}{:}\mathrm{O}, \mathtt{r}{:}\mathrm{R}) * \alpha \mapsto \mathrm{s}_{\mathrm{N}}[\beta, \gamma_1 \otimes \#\mathrm{text}_{\mathrm{O}}[\mathrm{s}']_{\mathrm{F}'} \otimes \gamma_2]^{\mathrm{E}}_{\mathrm{F}}|_{\epsilon} * s \rrbracket^{I}) \end{array} \right\} \tag{B.34}$$

We then proceed as follows:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{(\dagger)}{
\begin{array}{c}
\left\{\begin{array}{l} p * \exists l, l'.\ \mathbf{l} \doteq l' : l : Ls \\ * \, l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o\} \\ * \, l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o, \mathsf{i}{:}\texttt{undefined}\} \end{array}\right\} \\
\mathsf{e}_{\mathrm{rm}} \\
\left\{q' * \exists l, l'.\ \mathbf{l} \doteq l' : l : Ls * (l', \mathsf{o}) \mapsto o * \mathbf{r} \doteq l'.\mathsf{o} * \mathsf{true}\right\}
\end{array}}
}{
\begin{array}{c}
\left\{p * \exists l.\ \mathbf{l} \doteq l : Ls * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o\}\right\} \\
\texttt{this.childNodes.\_\_removeChild\_\_(o)} \\
\left\{q' * \exists l, l'.\ \mathbf{l} \doteq l : Ls * (l', \mathsf{o}) \mapsto o * \mathbf{r} \doteq l'.\mathsf{o} * \mathsf{true}\right\}
\end{array}}
\ \text{(Function Call)}^{\Rightarrow}\text{, (B.34)}
}{
\begin{array}{c}
\left\{p\right\} \\
\texttt{n.removeChild(o)} \\
\left\{q' * \exists l'.\ (l', \mathsf{o}) \mapsto o * \mathbf{r} \doteq l'.\mathsf{o} * \mathsf{true}\right\}
\end{array}}
\ \text{Lemma 22,(B.33)}
}{
\begin{array}{c}
\left\{p\right\} \\
\texttt{r = n.removeChild(o)} \\
\left\{q * \exists l'.\ (l', \mathsf{o}) \mapsto o * \mathbf{r} \doteq l'.\mathsf{o} * \mathsf{true}\right\}
\end{array}}
\ \text{(Assign Local))}^{\Rightarrow}\text{, (B.33)}
}{
\{p\}\ \texttt{r = n.removeChild(o)}\ \{q * \mathsf{true}\}
}\ \text{Consequence}
}{
\{p\}\ \texttt{r = n.removeChild(o)}\ \{q\}
}\ (*)
$$

where in $(*)$ we i) unfold the definition of $[\![.]\!]$; ii) apply the rule of consequence to absorb true; and iii) fold the definition of $[\![.]\!]$, and $(\dagger)$ follows from the proof sketch below where we have unwrapped the implementation code denoted by e_{rm}:

$$\left\{\begin{array}{l} p * \mathbf{l} \doteq l'{:}l{:}Ls * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o\} \\ * \, l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o, \mathtt{i}{:}\texttt{undefined}\} \end{array}\right\}$$

// From the definition of $⟦.⟧^I$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathsf{n}{:}n, \mathsf{o}{:}o, \mathsf{r}{:}r)_{Ls} * \mathbf{l} \doteq l'{:}l{:}Ls * \mathsf{Protos} * \mathsf{true} \\ * \bigcup_{(h,\mathbf{h}) \in s} \langle\!\langle h \rangle\!\rangle_{\text{JS}} * \mathsf{H}\,(\mathbf{h}, I) * \mathsf{Crust}\left(\mathbf{x} \mapsto \mathrm{s}_n[\mathbf{y}, \mathbf{z}_1 \otimes \#\mathrm{text}_o[\mathrm{s}']_{fs'} \otimes \mathbf{z}_2]^{ts}_{fs} \bullet \mathbf{h}, I\right) \\ * \exists u, l_a, l_{cn}, L_a, L_1, L_2.\ I(\mathbf{x}) \doteq ([n], u) * fs \doteq \{l_{cn}\} * \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) \\ * \mathsf{FL}\,(l_{cn}, n, L_1 +\!\!+ [o] +\!\!+ L_2) * \mathsf{TLs}\,(n, ts) * l_a \Mapsto \{\mathrm{s}'' : l \mid (\mathrm{s}'', l) \in L_a\} \\ * \mathsf{D}\,(\mathbf{y})_I^{(L_a, \texttt{null})} * \mathsf{D}\,(\mathbf{z}_1)_I^{(L_1, n)} * \mathsf{D}\left(\#\mathrm{text}_o[\mathrm{s}']_{fs'}\right)_I^{([o],n)} * \mathsf{D}\,(\mathbf{z}_2)_I^{(L_2, n)} * o \dot{\notin} L_1 \\ * \, l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o\} \\ * \, l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o, \mathtt{i}{:}\texttt{undefined}\} \end{array}\right\}$$

// Frame off

$$\left\{\begin{array}{l} \mathsf{vars}(\mathsf{n}{:}n, \mathsf{o}{:}o, \mathsf{r}{:}r)_{Ls} * \mathbf{l} \doteq l'{:}l{:}Ls \\ * \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) * \mathsf{FL}\,(l_{cn}, n, L_1 +\!\!+ [o] +\!\!+ L_2) \\ * \mathsf{D}\left(\#\mathrm{text}_o[\mathrm{s}']_{fs'}\right)_I^{([o],n)} * o \dot{\notin} L_1 * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o\} \\ * \, l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o, \mathtt{i}{:}\texttt{undefined}\} \end{array}\right\}$$

`var i;` // Apply (Definition)$^{\Rightarrow}$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathsf{n}{:}n, \mathsf{o}{:}o, \mathsf{r}{:}r)_{Ls} * \mathbf{l} \doteq l'{:}l{:}Ls \\ * \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) * \mathsf{FL}\,(l_{cn}, n, L_1 +\!\!+ [o] +\!\!+ L_2) \\ * \mathsf{D}\left(\#\mathrm{text}_o[\mathrm{s}']_{fs'}\right)_I^{([o],n)} * o \dot{\notin} L_1 * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o\} \\ * \, l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o, \mathtt{i}{:}\texttt{undefined}\} \end{array}\right\}$$

```
i = this.childNodes.__contents__.indexOf(o);
```

// array `indexOf` axiom in Def. 129, (Assign Local)$^{\Rightarrow}$, (Member Access)$^{\Rightarrow}$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathsf{n}{:}n, \mathsf{o}{:}o, \mathsf{r}{:}r)_{Ls} * \mathbf{l} \doteq l'{:}l{:}Ls \\ * \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) * \mathsf{FL}\,(l_{cn}, n, L_1 +\!\!+ [o] +\!\!+ L_2) \\ * \mathsf{D}\left(\#\mathrm{text}_o[\mathrm{s}']_{fs'}\right)_I^{([o],n)} * o \dot{\notin} L_1 * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o\} \\ * \, l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o, \mathtt{i}{:}\,|L_1|\} \end{array}\right\}$$

```
if (i === -1) { throw NOT_FOUND_ERROR }
```

$$\left\{\begin{array}{l} \mathsf{vars}(\mathsf{n}{:}n, \mathsf{o}{:}o, \mathsf{r}{:}r)_{Ls} * \mathbf{l} \doteq l'{:}l{:}Ls \\ * \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) * \mathsf{FL}\,(l_{cn}, n, L_1 +\!\!+ [o] +\!\!+ L_2) \\ * \mathsf{D}\left(\#\mathrm{text}_o[\mathrm{s}']_{fs'}\right)_I^{([o],n)} * o \dot{\notin} L_1 * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o\} \\ * \, l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\texttt{null}, \mathsf{o}{:}o, \mathtt{i}{:}\,|L_1|\} \end{array}\right\}$$

```
this.contents.splice(i, 1);
```

// array `splice` axiom in Def. 129, (Member Access)$^{\Rightarrow}$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{n}{:}n, \mathtt{o}{:}o, \mathtt{r}{:}r)_{Ls} * \mathbf{l} \doteq l'{:}l{:}Ls \\ * \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) * \exists l''.\ \mathsf{FL}\,(l_{cn}, n, L_1 {+\!\!+} L_2) * \mathsf{array}(l'', [o]) \\ * \mathsf{D}\,(\#\mathrm{text}_o[\mathrm{s}']_{fs'})_I^{([o],n)} * o \dot{\notin} L_1 * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}, \mathtt{o}{:}o\} \\ * l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}, \mathtt{o}{:}o, \mathtt{i}{:}\,|L_1|\} \end{array}\right\}$$

```
o.parentChild = null;
```

// (Assign Local)$^{\Rightarrow}$, (Member Access)$^{\Rightarrow}$

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{n}{:}n, \mathtt{o}{:}o, \mathtt{r}{:}r)_{Ls} * \mathbf{l} \doteq l'{:}l{:}Ls \\ * \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) * \mathsf{FL}\,(l_{cn}, n, L_1 {+\!\!+} L_2) \\ * \mathsf{D}\,(\#\mathrm{text}_o[\mathrm{s}']_{fs'})_I^{([o],\mathtt{null})} * o \dot{\notin} L_1 * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}, \mathtt{o}{:}o\} \\ * l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}, \mathtt{o}{:}o, \mathtt{i}{:}\,|L_1|\} * \exists l''.\ \mathsf{array}(l'', [o]) \end{array}\right\}$$

// Frame on

$$\left\{\begin{array}{l} \mathsf{vars}(\mathtt{n}{:}n, \mathtt{o}{:}o, \mathtt{r}{:}r)_{Ls} * \mathbf{l} \doteq l'{:}l{:}Ls * \mathsf{Protos} * \mathsf{true} \\ * \bigcup_{(h,\mathbf{h}) \in s} \langle\!\langle h \rangle\!\rangle_{\mathrm{JS}} * \mathsf{H}\,(\mathbf{h}, I) * \mathsf{Crust}\left(\mathbf{x} \mapsto \mathrm{s}_n[\mathbf{y}, \mathbf{z}_1 \otimes \#\mathrm{text}_o[\mathrm{s}']_{fs'} \otimes \mathbf{z}_2]_{fs}^{ts} \bullet \mathbf{h}, I\right) \\ * \exists u, l_a, l_{cn}, L_a, L_1, L_2.\ I(\mathbf{x}) \doteq ([n], u) * fs \doteq \{l_{cn}\} * \mathsf{ENode}\,(n, \mathrm{s}, l_a, l_{cn}, u) \\ * \mathsf{FL}\,(l_{cn}, n, L_1 {+\!\!+} L_2) * \mathsf{TLs}\,(n, ts) * l_a \Mapsto \{\mathrm{s}'' : l \mid (\mathrm{s}'', l) \in L_a\} \\ * \mathsf{D}\,(\mathbf{y})_I^{(L_a,\mathtt{null})} * \mathsf{D}\,(\mathbf{z}_1)_I^{(L_1,n)} * \mathsf{D}\,(\#\mathrm{text}_o[\mathrm{s}']_{fs'})_I^{([o],\mathtt{null})} * \mathsf{D}\,(\mathbf{z}_2)_I^{(L_2,n)} * o \dot{\notin} L_1 \\ * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}, \mathtt{o}{:}o\} \\ * l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}, \mathtt{o}{:}o, \mathtt{i}{:}\,|L_1|\} * \exists l''.\ \mathsf{array}(l'', [o]) \end{array}\right\}$$

// From the definition of $\llbracket . \rrbracket^I$

$$\left\{\begin{array}{l} q' * \mathbf{l} \doteq l'{:}l{:}Ls * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}, \mathtt{o}{:}o\} \\ * l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}, \mathtt{o}{:}o, \mathtt{i}{:}\,|L_1|\} * \exists l''.\ \mathsf{array}(l'', [o]) \end{array}\right\}$$

```
o;
```

$$\left\{\begin{array}{l} q' * \mathbf{l} \doteq l'{:}l{:}Ls * l \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}, \mathtt{o}{:}o\} \\ * l' \Mapsto \{@\mathit{this}{:}n, @\mathit{proto}{:}\mathtt{null}, \mathtt{o}{:}o, \mathtt{i}{:}\,|L_1|\} * \exists l''.\ \mathsf{array}(l'', [o]) * \mathbf{r} \doteq l'.\mathsf{o} \end{array}\right\}$$

$$\left\{ q' * \mathbf{l} \doteq l'{:}l{:}Ls * (l', \mathsf{o}) \mapsto o * \mathbf{r} \doteq l'.\mathsf{o} * \mathsf{true} \right\}$$

C. Auxiliary CoLoSL Lemmata

Lemma 24 ($\models_\dagger$ implies $\vDash_{\text{SL}}$)**.** *For all* $P \in \textsc{Ast}$*,* $\Gamma \in \textsc{LEnv}$*,* $s, g \in \textsc{IState}$ *and* $\mathfrak{I} \in \textsc{AMod}$*:*

$$\Gamma, s \models_{g,\mathfrak{I}} P \implies \Gamma, s \vDash_{\text{SL}} P$$

Proof. We proceed by induction on the structure of assertion P.

Case $P = p$ where $p \in \textsc{LAst}$
Immediate from the definition of $\models_{g,\mathfrak{I}}$ for local assertions.

Case $P = P_1 \vee P_2$ where $P_1, P_2 \in \textsc{Ast}$
Pick arbitrary $\Gamma \in \textsc{LEnv}$, $s, g \in \textsc{IState}$ and $\mathfrak{I} \in \textsc{AMod}$, such that:

$$s, \Gamma \models_{g,\mathfrak{I}} P_1 \vee P_2 \tag{C.1}$$

$$\forall \Gamma, s, g, \mathfrak{I}.\ \Gamma, s \models_{g,\mathfrak{I}} P_1 \implies \Gamma, s \vDash_{\text{SL}} P_1 \tag{I.H1}$$

$$\forall \Gamma, s, g, \mathfrak{I}.\ \Gamma, s \models_{g,\mathfrak{I}} P_2 \implies \Gamma, s \vDash_{\text{SL}} P_2 \tag{I.H2}$$

From (C.1) and the $\models_{g,\mathfrak{I}}$ definition we know $\Gamma, s \models_{g,\mathfrak{I}} P_1$ *or* $\Gamma, s \models_{g,\mathfrak{I}} P_2$. Consequently, from (I.H1) and (I.H2) we have: $\Gamma, s \vDash_{\text{SL}} P_1$ *or* $\Gamma, s \vDash_{\text{SL}} P_2$. Thus, from the definition of $\vDash_{\text{SL}}$ we have $\Gamma, s \vDash_{\text{SL}} P_1 \vee P_2$ as required.

Cases $P{=}\exists x.\, P'$ or $P{=}P_1 \wedge P_2$ or $P{=}P_1 * P_2$ or $P{=}P_1 \circledast P_2$
These cases are analogous to the previous case and are omitted here.

Case $P{=}\boxed{P'}_I$ where $P' \in \textsc{Ast}$ and $I \in \textsc{IAst}$
Pick arbitrary $\Gamma \in \textsc{LEnv}$, $s, g \in \textsc{IState}$ and $\mathfrak{I} \in \textsc{AMod}$ such that $\Gamma, s \models_{g,\mathfrak{I}}$ $\boxed{P'}_I$. From the definition of $\models_{g,\mathfrak{I}}$ we then know $(s, g, \mathfrak{I}) \models \boxed{P'}_I$ and hence, from the definition of $\models$, we have $s \in \textsc{Unit}_{\textsc{Ins}}$. Consequently, from the definition of $\vDash_{\text{SL}}$ we have $\Gamma, s \vDash_{\text{SL}} \boxed{P'}_I$ as required. $\square$

Lemma 25 ($\models$ implies $\vDash_{\text{SL}}$)**.** *For all* $P \in \textsc{Ast}$*,* $\Gamma \in \text{L}\textsc{Env}$*,* $l \in \text{I}\textsc{State}$ *and* $\mathcal{I} \in \text{AM}\textsc{od}$*:*

$$\Gamma, l \models P \implies \Gamma, l \vDash_{\text{SL}} P$$

Proof. We proceed by induction on the structure of assertion P.

Case $P = p$ where $p \in \text{LA}\textsc{st}$
Immediate from the definition of $\models$ for local assertions.

Case $P = P_1 \vee P_2$ where $P_1, P_2 \in \textsc{Ast}$
Pick an arbitrary $\Gamma \in \text{L}\textsc{Env}$ and $l \in \text{I}\textsc{State}$, such that:

$$s, \Gamma \models_{g,\mathcal{I}} P_1 \vee P_2 \tag{C.2}$$

$$\forall \Gamma, l.\ \Gamma, l \models P_1 \implies \Gamma, l \vDash_{\text{SL}} P_1 \tag{I.H1}$$

$$\forall \Gamma, l.\ \Gamma, l \models P_2 \implies \Gamma, l \vDash_{\text{SL}} P_2 \tag{I.H2}$$

From (C.2) and the definition of $\models$ we know $\Gamma, l \models P_1$ *or* $\Gamma, l \models P_2$. Consequently, from (I.H1) and (I.H2) we have: $\Gamma, l \vDash_{\text{SL}} P_1$ *or* $\Gamma, l \vDash_{\text{SL}} P_2$. Thus, from the definition of $\vDash_{\text{SL}}$ we have:

$$\Gamma, l \vDash_{\text{SL}} P_1 \vee P_2$$

as required.

Cases $P{=}\exists x.\ P'$ or $P{=}P_1 \wedge P_2$ or $P{=}P_1 * P_2$ or $P{=}P_1 \circledast P_2$
These cases are analogous to the previous case and are omitted here.

Case $P{=}\boxed{P'}_I$ where $P' \in \textsc{Ast}$ and $I \in \text{IA}\textsc{st}$
Pick an arbitrary $\Gamma \in \text{L}\textsc{Env}$ and $l \in \text{I}\textsc{State}$ such that:

$$\Gamma, l \vDash_{\text{SL}} \boxed{P'}_I \tag{C.3}$$

From (C.3) and the definition of $\models$ for boxed assertions we have $l \in \textsc{Unit}_{\textsc{Ins}}$. Consequently, from the definition of $\vDash_{\text{SL}}$ we have:

$$\Gamma, l \vDash_{\text{SL}} \boxed{P'}_I$$

as required. ☐

Lemma 26 (Compatibility)**.** *Given a partial commutative monoid* ($\mathcal{M}$, $\bullet_{\mathcal{M}}$, $\textsc{Unit}_{\mathcal{M}}$)*, for all* $a, b, c, d \in \mathcal{M}$*:*

$$a \bullet_{\mathcal{M}} b = d \ \land \ c \leq b \implies \exists f \in \mathcal{M}.\ a \bullet_{\mathcal{M}} c = f$$

Proof. Pick arbitrary $a, b, c, d \in \mathcal{M}$ such that:

$$a \bullet_{\mathcal{M}} b = d \tag{C.4}$$

$$c \leq b \tag{C.5}$$

From (C.5), we have:

$$\exists e \in \mathcal{M}.\ c \bullet_{\mathcal{M}} e = b \tag{C.6}$$

and consequently from (C.4) we have:

$$a \bullet_{\mathcal{M}} c \bullet_{\mathcal{M}} e = d \tag{C.7}$$

Let us assume that

$$\neg \exists f \in \mathcal{M}.\ a \bullet_{\mathcal{M}} c = f \tag{C.8}$$

From (C.7) and the associativity of monoids we know that

$$a \bullet_{\mathcal{M}} c \bullet_{\mathcal{M}} e = (a \bullet_{\mathcal{M}} c) \bullet_{\mathcal{M}} e = \text{undefined} \bullet_{\mathcal{M}} e = \text{undefined} \notin \mathcal{M}$$

which contradicts C.7. As such, our assumption in (C.8) is wrong and we have:

$$\exists f \in \mathcal{M}.\ a \bullet_{\mathcal{M}} c = f$$

as required. $\square$

Lemma 27 (Closure monotonicity)**.** *For all* $\mathcal{I}, \mathcal{I}' \in \textsc{AMod}$ *and* $n \in \mathbb{N}^+$*:*

$$\forall s, r \in \textsc{IState}.\ \mathcal{I}{\downarrow_n} \left(s, r, \mathcal{I}'\right) \Rightarrow \mathcal{I}{\downarrow_{(n-1)}} \left(s, r, \mathcal{I}'\right)$$

Proof. Pick arbitrary $\mathcal{I}, \mathcal{I}' \in \textsc{AMod}$ and proceed by natural induction on the number of steps n.

Base case n=1

Pick arbitrary $s, r \in \textsc{IState}$. We are then required to show $\mathcal{I} \downarrow_0 (s, r, \mathcal{I}')$ which trivially follows from the definition of $\downarrow_0$.

Inductive case n=m+1

Pick an arbitrary $s, r \in \textsc{IState}$ such that

$$\mathcal{I}\downarrow_{(m+1)} \left(s, r, \mathcal{I}'\right) \tag{C.9}$$

$$\forall s', r' \in \textsc{IState}.\ \mathcal{I}\downarrow_m \left(s', r', \mathcal{I}'\right) \implies \mathcal{I}\downarrow_{(m-1)} \left(s', r', \mathcal{I}'\right) \tag{I.H}$$

We are then required to show:

$$\forall \kappa.\ \forall a \in \mathcal{I}'(\kappa).\ \mathsf{reflected}(a, s \circ r, \mathcal{I}'(\kappa)) \tag{C.10}$$

$$\begin{array}{l}\forall \kappa.\ \forall a \in \mathcal{I}(\kappa).\ \mathsf{potential}(a, s \circ r) \Rightarrow \\ \quad (\mathsf{reflected}(a, s \circ r, \mathcal{I}'(\kappa)) \vee \neg\mathsf{visible}(a, s)) \wedge \\ \quad \forall (s', r') \in a[s, r].\ \mathcal{I}\downarrow_{(m-1)} (s', r', \mathcal{I}')\end{array} \tag{C.11}$$

RTS. (C.10)

Pick an arbitrary κ and $a \in \mathcal{I}'(\kappa)$. From (C.9) and the definition of $\downarrow$ we then have $\mathsf{reflected}(a, s \circ r, \mathcal{I}(\kappa))$ as required.

RTS. (C.11)

Pick an arbitrary κ and $a \in \mathcal{I}(\kappa)$ such that $\mathsf{potential}(a, s \circ r)$ holds. Then from (C.9) we have:

$$\begin{array}{l}(\mathsf{reflected}(a, s \circ r, \mathcal{I}'(\kappa)) \vee \neg\mathsf{visible}(a, s)) \wedge \\ \forall (s', r') \in a[s, r].\ \mathcal{I}\downarrow_m (s', r', \mathcal{I}')\end{array}$$

and consequently from (I.H)

$$\begin{array}{l}(\mathsf{reflected}(a, s \circ r, \mathcal{I}'(\kappa)) \vee \neg\mathsf{visible}(a, s)) \wedge \\ \forall (s', r') \in a[s, r].\ \mathcal{I}\downarrow_{(m-1)} (s', r', \mathcal{I}')\end{array}$$

as required.

$\square$

Lemma 28 (Forget closure). *For all* $\mathcal{I}, \mathcal{I}' \in \text{AMod}$ *(Def. 94) and* $s_1, s_2, r \in \text{IState}$ *(Def. 91):*

$$\mathcal{I}{\downarrow} \left(s_1 \circ s_2, r, \mathcal{I}'\right) \Rightarrow \mathcal{I}{\downarrow} \left(s_1, s_2 \circ r, \mathcal{I}'\right)$$

Proof. Pick arbitrary $\mathcal{I}, \mathcal{I}' \in \text{AMod}$ and $s_1, s_2, r \in \text{IState}$ such that:

$$\mathcal{I}{\downarrow} \left(s_1 \circ s_2, r, \mathcal{I}'\right) \tag{C.12}$$

From the definition of $\downarrow$ (Def. 107), it then suffices to show

$$\forall n \in \mathbb{N}.\ \mathcal{I}{\downarrow_n} \left(s_1, s_2 \circ r, \mathcal{I}'\right) \tag{C.13}$$

Rather than proving (C.13) directly, we first establish the following:

$$\begin{aligned}&\forall n \in \mathbb{N}.\ \forall s_1, s_2, r \in \text{IState}.\\ &\qquad \mathcal{I}{\downarrow_n} \left(s_1 \circ s_2, r, \mathcal{I}'\right) \Rightarrow \mathcal{I}{\downarrow_n} \left(s_1, s_2 \circ r, \mathcal{I}'\right)\end{aligned} \tag{C.14}$$

We can then despatch (C.13) as follows. For an arbitrary $n \in \mathbb{N}$, from (C.12) and the definition of $\downarrow$ we have $\mathcal{I} \downarrow_n (s_1 \circ s_2, r, \mathcal{I}')$. Consequently from (C.14) we have $\mathcal{I}{\downarrow_n} (s_1, s_2 \circ r, \mathcal{I}')$ as required.

RTS. (C.14)
We proceed by induction on the number of steps n.

Base case n=0
Pick arbitrary $s_1, s_2, r \in \text{IState}$. We are then required to show $\mathcal{I} \downarrow_0 (s_1, s_2 \circ r, \mathcal{I}')$ which follows trivially from the definition of $\downarrow_0$.

Inductive Step n=m+1
Pick arbitrary $s_1, s_2, r \in \text{IState}$ such that:

$$\mathcal{I}{\downarrow_{m+1}} \left(s_1 \circ s_2, r, \mathcal{I}'\right) \tag{C.15}$$

$$\begin{aligned}&\forall s_1, s_2, r \in \text{IState}.\\ &\quad \mathcal{I}{\downarrow_m} \left(s_1 \circ s_2, r, \mathcal{I}'\right) \implies \mathcal{I}{\downarrow_m} \left(s_1, s_2 \circ r, \mathcal{I}'\right)\end{aligned} \tag{I.H.}$$

We are the required to show:

$$\forall \kappa.\ \forall a \in \mathcal{I}'(\kappa).\ \mathsf{reflected}(a, s_1 \circ s_2 \circ r, \mathcal{I}(\kappa)) \tag{C.16}$$

$$\forall \kappa.\ \forall a \in \mathcal{I}(\kappa).\ \mathsf{potential}(a, s_1 \circ s_2 \circ r) \Rightarrow$$

$$(\mathsf{reflected}(a, s_1 \circ s_2 \circ r, \mathcal{I}'(\kappa)) \vee \neg\mathsf{visible}(a, s_1)) \tag{C.17}$$

$$\wedge\, \forall (s', r') \in a[s_1, s_2 \circ r].\ \mathcal{I} \downarrow_{m+1} \left(s', r', \mathcal{I}'\right)) \tag{C.18}$$

RTS. (C.16)
Pick arbitrary κ and $a \in \mathcal{I}'(\kappa)$. From (C.15) and the definition of $\downarrow_{m+1}$ we then have $\mathsf{reflected}(a, s_1 \circ s_2 \circ r, \mathcal{I}(\kappa))$ as required.

RTS. (C.17)
Pick an arbitrary κ and $a \in \mathcal{I}(\kappa)$ such that $\mathsf{potential}(a, s_1 \circ s_2 \circ r)$ holds. Then from (C.15) we have:

$$\mathsf{reflected}(a, s_1 \circ s_2 \circ r, \mathcal{I}'(\kappa)) \vee \neg\mathsf{visible}(a, s_1 \circ s_2)$$

In the case of the first disjunct the desired result holds trivially. In the case of the second disjunct, from the definition of $\mathsf{visible}$ (Def. 98) we have $\neg\mathsf{visible}(a, s_1)$ as required.

RTS. (C.18)
Pick an arbitrary κ and $a = (p, q, c) \in \mathcal{I}(\kappa)$ such that $\mathsf{potential}(a, s_1 \circ s_2 \circ r)$ holds. From (C.15) and the definition of $\downarrow$ we then have:

$$\forall (s', r') \in a[s_1 \circ s_2, r].\ \mathcal{I} \downarrow_{m} \left(s', r', \mathcal{I}'\right) \tag{C.19}$$

Pick an arbitrary (s', r') such that

$$(s', r') \in a[s_1, s_2 \circ r] \tag{C.20}$$

Then from the definition of $a[s_1, s_2 \circ r]$ and by the cross-split property we know there exists $ps_1, ps_2, p_r, s'_1, s'_2, r'' \in \textsc{IState}$ such that :

$$\begin{array}{l} p = ps_1 \circ ps_2 \circ p_r \wedge s_1 = ps_1 \circ s'_1 \wedge s_2 = ps_2 \circ s'_2 \wedge r = p_r \circ r'' \wedge \\ \left(\begin{array}{l} (ps_1 \notin \textsc{Unit}_{\textsc{Ins}} \wedge s' = q \circ s'_1 \wedge r' = s'_2 \circ r'') \\ \vee\, (ps_1 \in \textsc{Unit}_{\textsc{Ins}} \wedge s' = s_1 \wedge r' = q \circ s'_2 \circ r'') \end{array} \right) \end{array} \tag{C.21}$$

and consequently from the definition of $a[s_1 \circ s_2, r]$

$$\begin{array}{l} p = ps_1 \circ ps_2 \circ p_r \wedge s_1 = ps_1 \circ s_1' \wedge s_2 = ps_2 \circ s_2' \wedge r = p_r \circ r'' \wedge \\ \left(\begin{array}{l} (s' = q \circ s_1' \wedge r' = s_2' \circ r'' \wedge (q \circ s_1' \circ s_2', r'') \in a[s_1 \circ s_2, r]) \\ \vee \left(\begin{array}{l} ps_1 \in \textsc{Unit}_{\textsc{Ins}} \wedge s' = s_1 \wedge r' = q \circ s_2' \circ r'' \wedge \\ \left(\begin{array}{l} (ps_2 \in \textsc{Unit}_{\textsc{Ins}} \wedge (s_1' \circ s_2', q \circ r'') \in a[s_1 \circ s_2, r]) \\ \vee (ps_2 \notin \textsc{Unit}_{\textsc{Ins}} \wedge (s_1' \circ q \circ s_2', r'') \in a[s_1 \circ s_2, r]) \end{array} \right) \end{array} \right) \end{array} \right) \end{array}$$

That is,

$$\exists s'', r''.\ (s' \circ s'', r'') \in a[s_1 \circ s_2, r] \wedge r' = s'' \circ r'' \tag{C.22}$$

From (C.19) and (C.22) we then have

$$\exists s'', r''.\ \mathcal{I}{\downarrow_m} \left(s' \circ s'', r'', \mathcal{I}'\right) \wedge r' = s'' \circ r''$$

Finally from (I.H.) we have

$$\exists s'', r''.\ \mathcal{I}{\downarrow_m} \left(s', s'' \circ r'', \mathcal{I}'\right) \wedge r' = s'' \circ r''$$

That is,

$$\mathcal{I}{\downarrow_m} \left(s', r', \mathcal{I}'\right)$$

as required. □

Lemma 29 (Merge closure). *For all* $\mathcal{I}, \mathcal{I}_1, \mathcal{I}_2 \in \text{AMod}$ *and* $s_p, s_c, s_q, r \in$ IState*:*

$$\mathcal{I}{\downarrow} (s_p \circ s_c, s_q \circ r, \mathcal{I}_1) \land \mathcal{I}{\downarrow} (s_q \circ s_c, s_p \circ r, \mathcal{I}_2) \implies \\ \mathcal{I}{\downarrow} (s_p \circ s_c \circ s_q, r, \mathcal{I}_1 \cup \mathcal{I}_2)$$

Proof. Pick arbitrary $\mathcal{I}, \mathcal{I}_1, \mathcal{I}_2 \in \text{AMod}$ and $s_p, s_c, s_q, r \in \text{IState}$ such that

$$\mathcal{I}{\downarrow} (s_p \circ s_c, s_q \circ r, \mathcal{I}_1) \tag{C.23}$$

$$\mathcal{I}{\downarrow} (s_q \circ s_c, s_p \circ r, \mathcal{I}_2) \tag{C.24}$$

From the definition of $\downarrow$, it then suffices to show

$$\forall n \in \mathbb{N}.\ \mathcal{I}{\downarrow_n} (s_p \circ s_c \circ s_q, r, \mathcal{I}_1 \cup \mathcal{I}_2) \tag{C.25}$$

RTS. (C.25)
Rather than proving (C.25) directly, we first establish the following.

$$\begin{aligned} &\forall n \in \mathbb{N}.\ \forall s_p, s_c, s_q, r \in \text{IState}. \\ &\qquad \mathcal{I}{\downarrow_n} (s_p \circ s_c, s_q \circ r, \mathcal{I}_1) \land \mathcal{I}{\downarrow_n} (s_c \circ s_q, s_p \circ r, \mathcal{I}_2) \\ &\qquad\qquad \implies \mathcal{I}{\downarrow_n} (s_p \circ s_c \circ s_q, r, \mathcal{I}_1 \cup \mathcal{I}_2) \end{aligned} \tag{C.26}$$

We then despatch (C.25) as follows. For an arbitrary $n \in \mathbb{N}$, from (C.23), (C.24) and the definition of $\downarrow$ we have $\mathcal{I}{\downarrow_n} (s_p \circ s_c, s_q \circ r, \mathcal{I}_1) \land \mathcal{I}{\downarrow_n}$ $(s_c \circ s_q, s_p \circ r, \mathcal{I}_2)$. From (C.26) we then have $\mathcal{I}{\downarrow_n} (s_p \circ s_c \circ s_q, r, \mathcal{I}_1 \cup \mathcal{I}_2)$ as required.

RTS. (C.26)
We proceed by induction on the number of steps n.

Base case $n = 0$
Pick arbitrary $s_p, s_q, s_c, r \in \text{IState}$. We are then required to show $\mathcal{I}{\downarrow_0}$ $(s_p \circ s_c \circ s_q, r, \mathcal{I}')$ which follows trivially from the definition of $\downarrow_0$.

Inductive Step $n{=}m{+}1$ Pick arbitrary $s_p, s_q, s_c, r \in \text{IState}$ and $n \in \mathbb{N}$, such that

$$\mathcal{I}{\downarrow_{(m+1)}} (s_p \circ s_c, s_q \circ r, \mathcal{I}_1) \tag{C.27}$$

$$\mathcal{I}\downarrow_{(m+1)} (s_q \circ s_c, s_p \circ r, \mathcal{I}_2) \tag{C.28}$$

$$\begin{aligned}&\forall s_p, s_q, s_c, r \in \textsc{IState}.\\ &\quad \mathcal{I}\downarrow_m (s_p \circ s_c, s_q \circ r, \mathcal{I}_1) \;\wedge\; \mathcal{I}\downarrow_m (s_q \circ s_c, s_p \circ r, \mathcal{I}_2)\\ &\qquad\quad \implies \mathcal{I}\downarrow_m (s_p \circ s_c \circ s_q, r, \mathcal{I}_1 \cup \mathcal{I}_2)\end{aligned} \tag{I.H.}$$

We are then required to show:

$$\forall \kappa.\ \forall a \in (\mathcal{I}_1 \cup \mathcal{I}_2)(\kappa).\ \mathsf{reflected}(a, s_p \circ s_c \circ s_q \circ r, \mathcal{I}(\kappa)) \tag{C.29}$$

$$\begin{aligned}&\forall \kappa.\ \forall a \in \mathcal{I}(\kappa).\ \mathsf{potential}(a, s_p \circ s_c \circ s_q \circ r) \Rightarrow\\ &\quad (\mathsf{reflected}(a, s_p \circ s_c \circ s_q \circ r, (\mathcal{I}_1 \cup \mathcal{I}_2)(\kappa)) \vee \neg\mathsf{visible}(a, s_p \circ s_c \circ s_q)\,)\wedge\\ &\quad \forall (s', r') \in a[s_p \circ s_c \circ s_q, r].\ \mathcal{I}\downarrow_m (s', r', \mathcal{I}_1 \cup \mathcal{I}_2))\end{aligned} \tag{C.30}$$

RTS. (C.29)
Pick an arbitrary κ and $a \in (\mathcal{I}_1 \cup \mathcal{I}_2)(\kappa)$. From the definition of $\mathcal{I}_1 \cup \mathcal{I}_2$ we know either $a \in \mathcal{I}_1(\kappa)$ or $a \in \mathcal{I}_2(\kappa)$. In the former case from (C.27) and the definition of $\downarrow_{m+1}$ we have $\mathsf{reflected}(a, s_p \circ s_c \circ s_q \circ r, \mathcal{I}(\kappa))$ as required. Similarly, in the latter case from (C.28) and the definition of $\downarrow_{m+1}$ we have $\mathsf{reflected}(a, s_p \circ s_c \circ s_q \circ r, \mathcal{I}(\kappa))$ as required.

RTS. (C.30)
Pick arbitrary κ and $a = (p, q, c) \in \mathcal{I}(\kappa)$ such that:

$$\mathsf{potential}(a, s_p \circ s_c \circ s_q \circ r) \tag{C.31}$$

From (C.27) and (C.31) we have:

$$\begin{aligned}&(\mathsf{reflected}(a, s_p \circ s_c \circ s_q \circ r, \mathcal{I}_1(\kappa)) \vee \neg\mathsf{visible}(a, s_p \circ s_c))\wedge\\ &\forall (s', r') \in a[s_p \circ s_c, s_q \circ r].\ \mathcal{I}\downarrow_{(n-1)} (s', r', \mathcal{I}_1)\end{aligned}$$

and consequently from the definition of $\mathcal{I}_1 \cup \mathcal{I}_2$ we have:

$$\begin{aligned}&(\mathsf{reflected}(a, s_p \circ s_c \circ s_q \circ r, (\mathcal{I}_1 \cup \mathcal{I}_2)(\kappa)) \vee \neg\mathsf{visible}(a, s_p \circ s_c)) \wedge\\ &\forall (s', r') \in a[s_p \circ s_c, s_q \circ r].\ \mathcal{I}\downarrow_m (s', r', \mathcal{I}_1)\end{aligned} \tag{C.32}$$

Similarly, from (C.28) and (C.31) we have:

$$\begin{aligned}&(\mathsf{reflected}(a, s_p \circ s_c \circ s_q \circ r, (\mathcal{I}_1 \cup \mathcal{I}_2)(\kappa)) \vee \neg\mathsf{visible}(a, s_c \circ s_q)) \wedge \\ &\forall (s', r') \in a[s_c \circ s_q, s_p \circ r].\ \mathcal{I}{\downarrow_m} (s', r', \mathcal{I}_2)\end{aligned} \tag{C.33}$$

From (C.32), (C.33) and the definition of $\mathsf{visible}$ we have:

$$\begin{aligned}&(\mathsf{reflected}(a, s_p \circ s_c \circ s_q \circ r, (\mathcal{I}_1 \cup \mathcal{I}_2)(\kappa)) \vee \neg\mathsf{visible}(a, s_p \circ s_c \circ s_q)) \wedge \\ &\forall (s', r') \in a[s_p \circ s_c, s_q \circ r].\ \mathcal{I}{\downarrow_m} (s', r', \mathcal{I}_1) \wedge \\ &\forall (s', r') \in a[s_c \circ s_q, s_p \circ r].\ \mathcal{I}{\downarrow_m} (s', r', \mathcal{I}_2)\end{aligned} \tag{C.34}$$

Pick arbitrary $s', r' \in \textsc{IState}$ such that

$$(s', r') \in a[s_p \circ s_c \circ s_q, r] \tag{C.35}$$

Then from the definition of $a[s_p \circ s_c \circ s_q]$ and by the cross-split property we know there exist $p_p, p_c, p_q, s'_p, s'_c, s'_q \in \textsc{IState}$ such that :

$$\begin{aligned}&s' = s_p \circ s_c \circ s_q \vee \\ &\begin{pmatrix} (p_p \notin \textsc{Unit}_{\textsc{Ins}} \vee p_c \notin \textsc{Unit}_{\textsc{Ins}} \vee p_q \notin \textsc{Unit}_{\textsc{Ins}}) \\ \wedge\, s' = q \circ s'_p \circ s'_c \circ s'_q \wedge p = p_p \circ p_c \circ p_q \circ p_r \\ \wedge\, s_p = p_p \circ s'_p \wedge s_c = p_c \circ s'_c \wedge s_q = p_q \circ s'_q \wedge r = p_r \circ r' \end{pmatrix}\end{aligned} \tag{C.36}$$

and consequently from the definitions of $a[s_p \circ s_c, s_q \circ r]$ and $a[s_c \circ s_q, s_p \circ r]$ we have:

$$\begin{aligned}&\begin{pmatrix} s' = s_p \circ s_c \circ s_q \wedge (s_p \circ s_c, s_q \circ r') \in a[s_p \circ s_c, s_q \circ r] \\ \wedge\, (s_c \circ s_q, s_p \circ r') \in a[s_c \circ s_q, s_p \circ r] \end{pmatrix} \\ &\vee \begin{pmatrix} s' = q \circ s'_p \circ s'_c \circ s'_q \wedge \\ \begin{pmatrix} \begin{pmatrix} (p_c \notin \textsc{Unit}_{\textsc{Ins}} \vee (p_c \in \textsc{Unit}_{\textsc{Ins}} \wedge p_p, p_q \notin \textsc{Unit}_{\textsc{Ins}})) \wedge \\ (q \circ s'_p \circ s'_c, s'_q \circ r') \in a[s_p \circ s_c, s_q \circ r] \wedge \\ (q \circ s'_c \circ s'_q, s'_p \circ r') \in a[s_c \circ s_q, s_p \circ r] \end{pmatrix} \\ \vee \begin{pmatrix} p_p \notin \textsc{Unit}_{\textsc{Ins}} \wedge p_c, p_q \in \textsc{Unit}_{\textsc{Ins}} \wedge \\ (q \circ s'_p \circ s'_c, s'_q \circ r') \in a[s_p \circ s_c, s_q \circ r] \wedge \\ (s'_c \circ s'_q, q \circ s'_p \circ r') \in a[s_c \circ s_q, s_p \circ r] \end{pmatrix} \\ \vee \begin{pmatrix} p_q \notin \textsc{Unit}_{\textsc{Ins}} \wedge p_c, p_p \in \textsc{Unit}_{\textsc{Ins}} \wedge \\ (s'_p \circ s'_c, q \circ s'_q \circ r') \in a[s_p \circ s_c, s_q \circ r] \wedge \\ (q \circ s'_c \circ s'_q, s'_p \circ r') \in a[s_c \circ s_q, s_p \circ r] \end{pmatrix} \end{pmatrix} \end{pmatrix}\end{aligned} \tag{C.37}$$

From (29), (C.27), (C.28) and (C.31) we have:

$$
\begin{array}{l}
\left(\begin{array}{l}
s' = s_p \circ s_c \circ s_q \land \\
\mathcal{I}{\downarrow_m} \left(s_p \circ s_c, s_q \circ r', \mathcal{I}_1\right) \land \mathcal{I}{\downarrow_m} \left(s_c \circ s_q, s_p \circ r', \mathcal{I}_2\right)
\end{array}\right) \\
\lor \left(\begin{array}{l}
s' = q \circ s'_p \circ s'_c \circ s'_q \land \\
\left(\begin{array}{l}
\left(\begin{array}{l}
(p_c \notin \textsc{Unit}_{\textsc{Ins}} \lor (p_c \in \textsc{Unit}_{\textsc{Ins}} \land p_p, p_q \notin \textsc{Unit}_{\textsc{Ins}})) \land \\
\mathcal{I}{\downarrow_m} \left(q \circ s'_p \circ s'_c, s'_q \circ r', \mathcal{I}_1\right) \land \\
\mathcal{I}{\downarrow_m} \left(q \circ s'_c \circ s'_q, s'_p \circ r', \mathcal{I}_2\right)
\end{array}\right) \\
\lor \left(\begin{array}{l}
p_p \notin \textsc{Unit}_{\textsc{Ins}} \land p_c, p_q \in \textsc{Unit}_{\textsc{Ins}} \land \\
\mathcal{I}{\downarrow_m} \left(q \circ s'_p \circ s'_c, s'_q \circ r', \mathcal{I}_1\right) \land \\
\mathcal{I}{\downarrow_m} \left(s'_c \circ s'_q, q \circ s'_p \circ r', \mathcal{I}_2\right)
\end{array}\right) \\
\lor \left(\begin{array}{l}
p_q \notin \textsc{Unit}_{\textsc{Ins}} \land p_c, p_p \in \textsc{Unit}_{\textsc{Ins}} \land \\
\mathcal{I}{\downarrow_m} \left(s'_p \circ s'_c, q \circ s'_q \circ r', \mathcal{I}_1\right) \land \\
\mathcal{I}{\downarrow_m} \left(q \circ s'_c \circ s'_q, s'_p \circ r', \mathcal{I}_2\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}
\tag{C.38}
$$

and thus from (C.38) and (I.H.)

$$
\mathcal{I}{\downarrow_m} \left(s', r', \mathcal{I}_1 \cup \mathcal{I}_2\right) \tag{C.39}
$$

Finally, from (C.34), (C.35) and (C.39) we have:

$$
\begin{array}{l}
(\mathsf{reflected}(a, s_p \circ s_c \circ s_q \circ r, (\mathcal{I}_1 \cup \mathcal{I}_2)(\kappa)) \lor \neg\mathsf{visible}(a, s_p \circ s_c \circ s_q)) \land \\
\forall (s', r') \in a[s_p \circ s_c \circ s_q, r].\ \mathcal{I}{\downarrow_m} (s', r', \mathcal{I}_1 \cup \mathcal{I}_2)
\end{array}
$$

as required.

□

Lemma 30 (Shift auxiliary). *For all $\mathcal{I}_1, \mathcal{I}_2 \in$ AMod, $s, s', r, r' \in$ IState and $a \in rng(\mathcal{I}_1)$:*

$$\mathcal{I}_1 \sqsubseteq^{\{s\}} \mathcal{I}_2 \wedge \big((s', r') \in a(s) \vee (s', r') \in a[s, r]\big) \implies \exists r'' \leq r'.\, \mathcal{I}_1 \sqsubseteq^{\{s' \circ r''\}} \mathcal{I}_2$$

Proof. Pick arbitrary $\mathcal{I}_1, \mathcal{I}_2 \in$ AMod, $s, s', r, r' \in$ IState and $a \in rng(\mathcal{I}_1)$ such that:

$$\mathcal{I}_1 \sqsubseteq^{\{s\}} \mathcal{I}_2 \tag{C.40}$$

There are two cases to consider:
Case 1. $(s', r') \in a(s)$
From the definition of $\sqsubseteq^{\{s\}}$ and (C.40) we know there exists a fence $\mathcal{F}$ such that:

$$s \in \mathcal{F} \tag{C.41}$$

$$\mathcal{F} \rhd \mathcal{I}_1 \tag{C.42}$$

$$\begin{aligned} &\forall l \in \mathcal{F}.\, \forall \kappa.\, \forall a \in \mathcal{I}_2(\kappa).\, \mathsf{reflected}(a, l, \mathcal{I}_1(\kappa)) \wedge \\ &\forall a \in \mathcal{I}_1(\kappa).\, a(l) \neq \emptyset \wedge \mathsf{visible}(a, l) \Rightarrow \mathsf{reflected}(a, l, \mathcal{I}_2(\kappa)) \end{aligned} \tag{C.43}$$

From the definition of $\rhd$, (C.41)-(C.42) and the assumption of case 1. we know there exists $r'' \leq r'$ such that :

$$s' \circ r'' \in \mathcal{F} \tag{C.44}$$

Finally by definition of $\sqsubseteq^{\{s' \circ r''\}}$ and (C.42)-(C.44) we have

$$\mathcal{I}_1 \sqsubseteq^{\{s' \circ r''\}} \mathcal{I}_2$$

as required.

Case 2. $(s', r') \in a[s, r]$
From the definitions of $a[s, r]$ and $a(s)$ and the assumption of the case we know $(s', r') \in a(s)$. The required result then follows from case 1. $\square$

Lemma 31 (Shift closure). *For all* $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I} \in \text{AMod}$ *and* $s, r \in \text{IState}$*,*

$$\mathcal{I}{\downarrow} (s, r, \mathcal{I}_1) \land \mathcal{I}_1 \sqsubseteq^{\{s\}} \mathcal{I}_2 \implies \mathcal{I}{\downarrow} (s, r, \mathcal{I}_2)$$

Proof. Pick arbitrary $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I} \in \text{AMod}$ and $s, r \in \text{IState}$ such that

$$\mathcal{I}{\downarrow} (s, r, \mathcal{I}_1) \tag{C.45}$$

$$\mathcal{I}_1 \sqsubseteq^{\{s\}} \mathcal{I}_2 \tag{C.46}$$

From the definition of $\downarrow$, we are then required to show:

$$\forall n \in \mathbb{N}.\ \mathcal{I}{\downarrow_n} (s, r, \mathcal{I}_2) \tag{C.47}$$

Rather than proving (C.47) directly, we first establish the following.

$$\begin{aligned} &\forall n \in \mathbb{N}.\ \forall s, r, r' \in \text{IState}. \\ &\quad \mathcal{I}{\downarrow_n} (s, r, \mathcal{I}_1) \land r' \leq r \land \mathcal{I}_1 \sqsubseteq^{\{s \circ r'\}} \mathcal{I}_2 \implies \mathcal{I}{\downarrow_n} (s, r, \mathcal{I}_2) \end{aligned} \tag{C.48}$$

We can then despatch (C.47) as follows. For an arbitrary $n \in \mathbb{N}$, from (C.45) and the definition of $\downarrow$ we have $\mathcal{I}{\downarrow_n} (s, r, \mathcal{I}_1)$. Let $r' \in \text{Unit}_{\text{Ins}}$ denote a unit element. As such we have $r' \leq r$ and $s \circ r' {=} s$. Consequently from (C.46) and (C.48) we have $\mathcal{I}{\downarrow_n} (s, r, \mathcal{I}_2)$ as required.

RTS. (C.48)
We proceed by induction on the number of steps n.

Base case $n = 0$
Pick arbitrary $s, r \in \text{IState}$. We are then required to show $\mathcal{I}{\downarrow_0} (s, r, \mathcal{I}_2)$ which follows trivially from the definition of $\downarrow_0$.

Inductive Case $n{=}m{+}1$
Pick arbitrary $s, r, r_0 \in \text{IState}$ such that:

$$\mathcal{I}{\downarrow_{m+1}} (s, r, \mathcal{I}_1) \tag{C.49}$$

$$r_0 \leq r \tag{C.50}$$

$$\mathcal{I}_1 \sqsubseteq^{\{s \circ r_0\}} \mathcal{I}_2 \tag{C.51}$$

$\forall s, r, r_0 \in \text{IState}.$

$$\mathcal{I}{\downarrow_m}\,(s, r, \mathcal{I}_1) \wedge r_0 \leq r \wedge \mathcal{I}_1 \sqsubseteq^{\{s \circ r_0\}} \mathcal{I}_2 \implies \mathcal{I}{\downarrow_m}\,(s, r, \mathcal{I}_2) \tag{I.H}$$

We are then required to show:

$$\forall \kappa.\, \forall a \in \mathcal{I}_2(\kappa).\, \mathsf{reflected}(a, s \circ r, \mathcal{I}(\kappa))) \tag{C.52}$$

$$\begin{aligned} &\forall \kappa.\, \forall a \in \mathcal{I}(\kappa).\, \mathsf{potential}(a, s \circ r) \Rightarrow \\ &\quad (\mathsf{reflected}(a, s \circ r, \mathcal{I}_2(\kappa)) \vee \neg\mathsf{visible}(a, s)\,)\wedge \\ &\quad \forall (s', r') \in a[s, r].\, \mathcal{I}{\downarrow_m}\,(s', r', \mathcal{I}_2) \end{aligned} \tag{C.53}$$

RTS. (C.52)
Pick arbitrary κ and $a = (p, q, c) \in \mathcal{I}_2(\kappa)$ and $l \in \text{IState}$ such that :

$$p \circ c \leq s \circ r \circ l \tag{C.54}$$

From (C.50), (C.51) and the definition of reflected we then know there exist a'', c'' such that:

$$a'' = (p, q, c'') \in \mathcal{I}_1(\kappa) \wedge p \circ c'' \leq s \circ r \circ l$$

Consequently from (C.49) and the definition of reflected we know there exists a', c' such that:

$$a' = (p, q, c') \in \mathcal{I}(\kappa) \wedge p \circ c' \leq s \circ r \circ l \tag{C.55}$$

Thus from (C.54),(C.55) and the definition of reflected we have:

$$\mathsf{reflected}(a, s \circ r, \mathcal{I}(\kappa))$$

as required.

RTS. (C.53)
Pick arbitrary κ and $a = (p, q, c) \in \mathcal{I}(\kappa)$ such that:

$$\mathsf{potential}(a, s \circ r) \tag{C.56}$$

From (C.49) and (C.56) we have:

$$\begin{aligned}&(\mathsf{reflected}(a, s \circ r, \mathcal{I}_1(\kappa)) \lor \neg\mathsf{visible}(a, s)\ )\land\\ &\forall(s', r') \in a[s, r].\ \mathcal{I}{\downarrow_m} (s', r', \mathcal{I}_1)\end{aligned} \tag{C.57}$$

Pick an arbitrary (s', r') such that

$$(s', r') \in a[s, r] \tag{C.58}$$

Then from (C.57) we have:

$$\mathcal{I}{\downarrow_m} \left(s', r', \mathcal{I}_1\right) \tag{C.59}$$

On the other hand, from (C.58) and Lemma 30 we know there exists $r_1 \leq r'$ such that:

$$\mathcal{I}_1 \sqsubseteq^{\{s' \circ r_1\}} \mathcal{I}_2 \tag{C.60}$$

Consequently, from (C.59), (C.60) and (I.H) we have:

$$\mathcal{I}{\downarrow_m} \left(s', r', \mathcal{I}_2\right)$$

and thus from (C.58) we have

$$\forall(s', r') \in a[s, r].\ \mathcal{I}{\downarrow_m} \left(s', r', \mathcal{I}_2\right) \tag{C.61}$$

Since either $\mathsf{visible}(a, s)$ or $\neg\mathsf{visible}(a, s)$, there are two cases to consider:

Case 1. $\neg\mathsf{visible}(a, s)$
From the assumption of the case and (C.61) we then have:

$$\neg\mathsf{visible}(a, s) \land \forall(s', r') \in a[s, r].\ \mathcal{I}{\downarrow_m} \left(s', r', \mathcal{I}_2\right)$$

as required.

Case 2. $\mathsf{visible}(a, s)$
From (C.57) and the assumption of the case we have:

$$\mathsf{reflected}(a, s \circ r, \mathcal{I}_1(\kappa)) \tag{C.62}$$

Pick an arbitrary $l \in \text{IState}$ such that:

$$p \circ c \leq s \circ r \circ l \tag{C.63}$$

Then from (C.62) and the definition of $\mathsf{reflected}$ we know there exist a', c' such that:

$$a' = (p, q, c') \land a' \in \mathcal{I}_1(\kappa) \land p \circ c' \leq s \circ r \circ l \tag{C.64}$$

From (C.56) and by definition of $\mathsf{potential}$ we know $a[s \circ r]$ is non-empty. From (C.64), and the definition of $a'[s \circ r]$ we know that $a'[s \circ r]$ is also non-empty. Consequently, from the definition of $a'(s)$, we know $a'(s)$ is also non-empty.
On the other hand, from the definition of $\mathsf{visible}$, (C.64) and the assumption of the case we have:

$$\mathsf{visible}(a', s) \tag{C.65}$$

Thus from (C.50), (C.51), (C.64), (C.65) and from the definition of $\sqsubseteq^{\{s \circ r_0\}}$ we know there exist a'', c'' such that:

$$a'' = (p, q, c'') \land a'' \in \mathcal{I}_2(\kappa) \land p \circ c'' \leq s \circ r \circ l \tag{C.66}$$

Finally, from (C.63), (C.66) and by definition of $\mathsf{reflected}$ we have:

$$\mathsf{reflected}(a, s \circ r, \mathcal{I}_2(\kappa)) \tag{C.67}$$

From (C.61) and (C.67) we have

$$\mathsf{reflected}(a, s \circ r, \mathcal{I}_2(\kappa)) \land \forall (s', r') \in a[s, r].\ \mathcal{I}{\downarrow_m} \left(s', r', \mathcal{I}_2\right)$$

as required.

$\square$

Lemma 32 (Confinement monotonocity)**.** *For all* $g, g' \in \text{IS}\textsc{tate}$ *and* $\mathcal{I}, \mathcal{I}' \in \text{AM}\textsc{od}$*,*

$$g \sharp g' \land g \copyright \mathcal{I} \land g' \copyright \mathcal{I}' \implies g \circ g' \copyright \mathcal{I} \cup \mathcal{I}'$$

Proof. Pick arbitrary $g, g' \in \text{IS}\textsc{tate}$ and $\mathcal{I}, \mathcal{I}' \in \text{AM}\textsc{od}$ such that:

$$g \sharp g' \land g \copyright \mathcal{I} \land g' \copyright \mathcal{I}' \tag{C.68}$$

From the definition of $\copyright$ and (C.68) we know there exist $\mathcal{F}, \mathcal{F}'$ such that

$$g \in \mathcal{F} \land \mathcal{F} \blacktriangleright \mathcal{I} \tag{C.69}$$

$$g' \in \mathcal{F} \land \mathcal{F}' \blacktriangleright \mathcal{I}' \tag{C.70}$$

Let

$$\mathcal{F}'' \triangleq \left\{ l \circ l' \,\middle|\, l \in \mathcal{F} \land l' \in \mathcal{F}' \land l \sharp l' \right\} \tag{C.71}$$

From the definition of $\mathcal{F}''$ and since $g \sharp g'$ (C.68) we know:

$$g \circ g' \in \mathcal{F}'' \tag{C.72}$$

Pick arbitrary $l'' \in \mathcal{F}''$ and action $a{=}(p, q, c) \in rng[(\mathcal{I} \cup \mathcal{I}')_{\text{A}}]$ From the definition of $\mathcal{I} \cup \mathcal{I}'$ we know that either $a \in rng(\mathcal{I})$ or $a \in rng(\mathcal{I}')$. Without loss of generality, let us assume that $a \in rng(\mathcal{I})$. From the definition of $\mathcal{F}''$ we know there exists l, l' such that:

$$l''{=}l \circ l' \land l \in \mathcal{F} \land l' \in \mathcal{F}' \tag{C.73}$$

On the other hand from (C.69), and the definition of $\copyright$ we know:

$$\forall r.\ l \sharp r \land \mathsf{agree}(p \circ c, l) \Rightarrow p \leq l \land p \perp r \tag{C.74}$$

Pick arbitrary r such that $l'' \sharp r$. Assume that $\mathsf{agree}(p \circ c, l'')$. From the definition of agree and l'' we know that $\mathsf{agree}(p \circ c, l)$ and $l \sharp r$. As such, from (C.74) above we have $p \leq l \land p \perp r$, and consequently from the definition of l'', $p \leq l'' \land p \perp r$. That is, we have:

$$l'' \copyright a \tag{C.75}$$

Pick an arbitrary $r' \in a[l'']$. From the definition of $a[l'']$ we know that there exists s such that:

$$\mathsf{agree}(p \circ c, l'') \land p \circ s{=}l'' \land q \sharp s \land r'{=}q \circ s \tag{C.76}$$

From the definition of agree and the definition of l'' in (C.73) we have $\mathsf{agree}(p \circ c, l)$ and thus from the definition of l'' and (C.74) we have:

$$p \leq l \land p \perp l' \tag{C.77}$$

On the other hand, from the cross-split property and (C.76) we know there exist $p_l, p_{l'}, s_l, s_{l'}$ such that $p{=}p_l \circ p_{l'}$, $s{=}s_l \circ s_{l'}$, $l{=}p_l \circ s_l$ and $l'{=}p_{l'} \circ s_{l'}$. Since $p \perp l'$ (C.77) we then know that $p_{l'} \in \textsc{Unit}_{\textsc{Ins}}$, and thus $p{=}p_l$, $l{=}p{\circ}s_l$ and $l'{=}s_{l'}$. As such from (C.76) we know:

$$r'{=}q \circ s_l \circ l' \tag{C.78}$$

On the other hand, from (C.76), the definitions of l'' and agree and since $s{=}s_l \circ s_{l'}$ we know $\mathsf{agree}(p \circ c, l) \land q \sharp s_l$. Consequently, from the definition of $a[l]$ and since $l{=}p \circ s_l$ we know

$$q \circ s_l \in a[l] \tag{C.79}$$

From (C.69), (C.73), (C.79) and the definition of $\blacktriangleright$ we have:

$$q \circ s_l \in \mathcal{F} \tag{C.80}$$

From (C.74), (C.80), the definition of $a[l'']$ (C.76) and the definition of $\mathcal{F}''$ we have:

$$a[l''] \in \mathcal{F}'' \tag{C.81}$$

From (C.75), (C.81) and the definition of $\blacktriangleright$ we have $\mathcal{F}'' \blacktriangleright \mathcal{I} \cup \mathcal{I}'$. Thus from (C.75) and above we have:

$$g \circ g' \text{ ⓒ } \mathcal{I} \cup \mathcal{I}'$$

as required. $\square$

Lemma 33 (Extend closure)**.** *For all* $\mathcal{I}, \mathcal{I}_e \in$ AMod *and for all* $g, s_e \in$ IState*,*

$$g \circledcirc \mathcal{I} \wedge s_e \circledcirc \mathcal{I}_e \implies \mathcal{I} \cup \mathcal{I}_e {\downarrow} (s_e, g, \mathcal{I}_e)$$

Proof. Pick arbitrary $\mathcal{I}, \mathcal{I}_e \in$ AMod and $g, s_e \in$ IState such that

$$g \circledcirc \mathcal{I} \wedge s_e \circledcirc \mathcal{I}_e \tag{C.82}$$

From the definition of $\circledcirc$ we then know there exist $\mathcal{F}$ and $\mathcal{F}_e$ such that

$$g \in \mathcal{F} \wedge s_e \in \mathcal{F}_e \tag{C.83}$$

$$\mathcal{F} \blacktriangleright \mathcal{I} \wedge \mathcal{F}_e \blacktriangleright \mathcal{I}_e \tag{C.84}$$

From the definition of $\downarrow$, it then suffices to show

$$\forall n \in \mathbb{N}.\, \mathcal{I} \cup \mathcal{I}_e {\downarrow_n} (s_e, g, \mathcal{I}_e) \tag{C.85}$$

Rather than proving (C.85) directly, we first establish the following:

$$\begin{aligned} &\forall n \in \mathbb{N}.\, \forall g, s_e \in \text{IState}. \\ &\quad g \in \mathcal{F} \wedge s_e \in \mathcal{F}_e \implies \mathcal{I} \cup \mathcal{I}_e {\downarrow_n} (s_e, g, \mathcal{I}_e) \end{aligned} \tag{C.86}$$

We can then despatch (C.85) as follows. For an arbitrary $n \in \mathbb{N}$, from (C.83) and (C.86) we have $\mathcal{I} \cup \mathcal{I}_e {\downarrow_n} (s_e, g, \mathcal{I}_e)$ as required.

RTS. (C.86)
We proceed by induction on the number of steps n.

Base case $n = 0$
Pick arbitrary $g, s_e \in$ IState. We are then required to show $\mathcal{I} \cup \mathcal{I}_e \downarrow_0 (s_e, g, \mathcal{I}_e)$ which follows trivially from the definition of $\downarrow_0$.

Inductive case $n{=}m{+}1$
Pick arbitrary $n \in \mathbb{N}$ and $g, s_e \in$ IState such that:

$$g \in \mathcal{F} \tag{C.87}$$

$$s_e \in \mathcal{F}_e \tag{C.88}$$

$$\forall g', s'_e.\ g' \in \mathcal{F} \wedge s'_e \in \mathcal{F}_e \implies \mathcal{I} \cup \mathcal{I}_e {\downarrow_m} \left(s'_e, g', \mathcal{I}_e\right) \tag{I.H}$$

We are then required to show:

$$\forall \kappa.\ \forall a \in \mathcal{I}_e(\kappa).\ \mathsf{reflected}(a, s_e \circ g, (\mathcal{I} \cup \mathcal{I}_e)(\kappa)) \tag{C.89}$$

$$\begin{aligned}&\forall \kappa.\ \forall a \in (\mathcal{I} \cup \mathcal{I}_e)(\kappa).\ \mathsf{potential}(a, s_e \circ g) \Rightarrow \\ &\quad (\mathsf{reflected}(a, s_e \circ g, \mathcal{I}_e(\kappa)) \vee \neg\mathsf{visible}(a, s_e)\ ) \wedge \\ &\quad \forall (s', r') \in a[s_e, g].\ \mathcal{I} \cup \mathcal{I}_e {\downarrow_m} (s', r', \mathcal{I}_e)\end{aligned} \tag{C.90}$$

RTS. C.89

Pick arbitrary κ and $a \in \mathcal{I}_e(\kappa)$. From the definitions of $\mathcal{I} \cup \mathcal{I}_e$ and $\mathsf{reflected}$ we then trivially have $\mathsf{reflected}(a, s_e \circ g, (\mathcal{I} \cup \mathcal{I}_e)(\kappa))$ as required.

RTS. C.90

Pick an arbitrary κ, $a = (p, q, c) \in (\mathcal{I} \cup \mathcal{I}_e)\,(\kappa)$ and (s', r') such that:

$$\mathsf{potential}(a, s_e \circ g) \tag{C.91}$$

$$(s', r') \in a[s_e, g] \tag{C.92}$$

There are two cases to consider.

Case 1. $a \in \mathcal{I}(\kappa)$

From the assumption of the case, (C.87), (C.84), the definition of $\blacktriangleright$ and since $s_e \circ g$ is defined we have:

$$p \perp s_e \tag{C.93}$$

From (C.91) and the definition of $\mathsf{potential}$ we have $\exists l_1, l_2.\ p \circ c \circ l_1 = s_e \circ g \circ l_2$ and $\exists l.\ p \circ l = s_e \circ g \wedge q \sharp l$. Consequently, we have $\exists l_1, l_2.\ p \circ c \circ l_1 = g \circ l_2$ and $\exists l.\ p \circ l = s_e \circ g \wedge q \sharp l$.

On the other hand from the cross-split property we know there exist p_g, p_e, l_g and l_e such that $p = p_g \circ p_e$, $l = l_g \circ l_e$, $g = p_g \circ l_g$ and $s_e = p_e \circ l_e$. Since $p \perp s_e$ (C.93), we know $p_e \in \textsc{Unit}_{\textsc{Ins}}$ and thus $p = p_g$ and $g = p \circ l_g$. Lastly, since $q \sharp l$, from the definition of l we know that $q \sharp l_g$, and thus we have $\exists l_g.\ p \circ l_g = g \wedge q \sharp l_g$. That is,

$$\mathsf{potential}(a, g) \tag{C.94}$$

From (C.93) and the definition of visible we have:

$$\neg\mathsf{visible}(a, s_e) \tag{C.95}$$

On the other hand, from (C.92), (C.93) and the definitions of $a[s_e, g]$, $a[g]$ and $\perp$, we know:

$$s' = s_e \tag{C.96}$$

$$r \in a[g] \tag{C.97}$$

Consequently, from (C.84), (C.87), (C.94), (C.97) and the definition of $\blacktriangleright$ we have:

$$r' \in \mathcal{F} \tag{C.98}$$

Finally, from (C.88), (C.96), (C.98), (I.H) we have:

$$\mathcal{I} \cup \mathcal{I}_e {\downarrow_m} \left(s', r', \mathcal{I}_e\right) \tag{C.99}$$

and consequently from (C.95) and (C.92), (C.99) we have

$$\neg\mathsf{visible}(a, s_e) \land \forall (s', r') \in a[s_e, g].\, \mathcal{I} \cup \mathcal{I}_e {\downarrow_m} \left(s', r', \mathcal{I}_e\right)$$

as required.

Case 2. $a \in \mathcal{I}_e(\kappa)$
From the assumption of the case and the definition of reflected we trivially have:

$$\mathsf{reflected}(a, s_e \circ g, \mathcal{I}_e(\kappa)) \tag{C.100}$$

From the assumption of the case, (C.88), (C.84) and the definition of $\blacktriangleright$ we have:

$$p \perp g \tag{C.101}$$

From (C.91) and the definition of potential we have $\exists l_1, l_2.\, p{\circ}c{\circ}l_1{=}s_e{\circ}g{\circ}l_2$ and $\exists l.\, p \circ l{=}s_e \circ g \land q \sharp l$. Consequently, we have $\exists l_1, l_2.\, p \circ c \circ l_1 = s_e \circ l_2$

and $\exists l.\ p \circ l{=}s_e \circ g \wedge q \sharp l$.
On the other hand from the cross split property we know there exist p_g, p_e, l_g and l_e such that $p{=}p_g \circ p_e$, $l{=}l_g \circ l_e$, $g{=}p_g \circ l_g$ and $s_e{=}p_e \circ l_e$. Since $p \perp g$ (C.93), we know $p_g \in \textsc{Unit}_{\textsc{Ins}}$ and thus $p{=}p_e$ and $s_e{=}p \circ l_e$. Lastly, since $q \sharp l$, from the definition of l we know that $q \sharp l_e$, and thus we have $\exists l_e.\ p \circ l_e{=}s_e \wedge q \sharp l_e$. That is,

$$s_e{=}p \circ l_e \wedge \mathsf{potential}(a, s_e) \tag{C.102}$$

On the other hand, from (C.92), (C.101) and the definitions of $a[s_e, g]$ and $\perp$, we have:

$$r' = g \tag{C.103}$$

$$s' \in a[s_e] \tag{C.104}$$

Consequently, from (C.84), (C.88), (C.102), (C.104) and the definition of $\blacktriangleright$ we have:

$$s' \in \mathcal{F}_e \tag{C.105}$$

Finally, from (C.87), (C.103), (C.105) and (I.H) we have:

$$\mathcal{I} \cup \mathcal{I}_e {\downarrow_m} \left(s', r', \mathcal{I}_e\right) \tag{C.106}$$

Thus from (C.92) and (C.100), (C.106) we have:

$$\mathsf{reflected}(a, s_e \circ g, \mathcal{I}_e(\kappa)) \wedge \forall (s', r') \in a[s_e, g].\ \mathcal{I} \cup \mathcal{I}_e {\downarrow_m} \left(s', r', \mathcal{I}_e\right)$$

as required.

$\square$

Lemma 34 (Extend closure (continued))**.** *For all* $\mathcal{I}_0, \mathcal{I}, \mathcal{I}_e \in \text{AMod}$ *and* $s, r, s_e \in \text{IState}$*:*

$$\begin{aligned} & s \circ r \circledcirc \mathcal{I} \land s_e \circledcirc \mathcal{I}_e \land \mathcal{I}\!\downarrow (s, r, \mathcal{I}_0) \\ & \quad \Rightarrow \mathcal{I} \cup \mathcal{I}_e\!\downarrow (s, r \circ s_e, \mathcal{I}_0) \end{aligned}$$

Proof. Pick arbitrary $\mathcal{I}_0, \mathcal{I}, \mathcal{I}_e \in \text{AMod}$ and $s, r, s_e \in \text{IState}$ such that

$$s \circ r \circledcirc \mathcal{I} \land s_e \circledcirc \mathcal{I}_e \tag{C.107}$$

$$\mathcal{I}\!\downarrow (s, r, \mathcal{I}_0) \tag{C.108}$$

From (C.107) and the definition of $\circledcirc$ we know there exist $\mathcal{F}$ and $\mathcal{F}_e$ such that

$$s \circ r \in \mathcal{F} \land s_e \in \mathcal{F}_e \tag{C.109}$$

$$\mathcal{F} \blacktriangleright \mathcal{I} \land \mathcal{F}_e \blacktriangleright \mathcal{I}e \tag{C.110}$$

From the definition of $\downarrow$, it then suffices to show

$$\forall n \in \mathbb{N}.\ \mathcal{I} \cup \mathcal{I}_e\!\downarrow_n (s, r \circ s_e, \mathcal{I}_0) \tag{C.111}$$

RTS. (C.111)
Rather than proving (C.111) directly, we first establish the following.

$$\begin{aligned} & \forall n \in \mathbb{N}.\ \forall s, r, s_e \in \text{IState}. \\ & \quad s \circ r \in \mathcal{F} \land s_e \in \mathcal{F}_e \land \mathcal{I}\!\downarrow_n (s, r, \mathcal{I}_0) \Rightarrow \mathcal{I} \cup \mathcal{I}_e\!\downarrow_n (s, r \circ s_e, \mathcal{I}_0) \end{aligned} \tag{C.112}$$

We can then despatch (C.111)as follows. For an arbitrary $n \in \mathbb{N}$, from (C.108) and the definition of $\downarrow$ we have $\mathcal{I}\!\downarrow_n (s, r, \mathcal{I}_0)$. Consequently from (C.109) and (C.112) we have $\mathcal{I} \cup \mathcal{I}_e\!\downarrow_n (s, r \circ s_e, \mathcal{I}_0)$ as required.

RTS. (C.112)
We proceed by induction on the number of steps n.

Base case $n = 0$
Pick arbitrary $s, r, s_e \in \text{IState}$. We are then required to show $\mathcal{I} \cup \mathcal{I}_e\!\downarrow_0$ $(s, r \circ s_e, \mathcal{I}_0)$ which follows trivially from the definition of $\downarrow_0$.

Inductive case $n{=}m{+}1$

Pick arbitrary $n \in \mathbb{N}$ and $s, r, s_e \in \textsc{IState}$ such that

$$s \circ r \in \mathcal{F} \tag{C.113}$$

$$s_e \in \mathcal{F}_e \tag{C.114}$$

$$\mathcal{I}{\downarrow_{(m+1)}} (s, r, \mathcal{I}_0) \tag{C.115}$$

$$\begin{aligned}&\forall s'', r'', s_e''.\ s'' \circ r'' \in \mathcal{F} \land s_e'' \in \mathcal{F}_e \land \mathcal{I}{\downarrow_m} \left(s'', r'', \mathcal{I}_0\right) \\ &\qquad \Rightarrow \mathcal{I} \cup \mathcal{I}_e{\downarrow_m} \left(s'', r'' \circ s_e'', \mathcal{I}_0\right)\end{aligned} \tag{I.H}$$

We are then required to show:

$$\forall \kappa.\ \forall a \in \mathcal{I}_0(\kappa).\ \mathsf{reflected}(a, s \circ r \circ s_e, (\mathcal{I} \cup \mathcal{I}_e)(\kappa)) \tag{C.116}$$

$$\begin{aligned}&\forall \kappa.\ \forall a \in (\mathcal{I} \cup \mathcal{I}_e)(\kappa).\ \mathsf{potential}(a, s \circ r \circ s_e) \Rightarrow \\ &\quad (\mathsf{reflected}(a, s \circ r \circ s_e, \mathcal{I}_0(\kappa)) \lor \neg\mathsf{visible}(a, s)\ )\land \\ &\quad \forall (s', r') \in a[s, r \circ s_e].\ \mathcal{I} \cup \mathcal{I}_e{\downarrow_m} (s', r', \mathcal{I}_0)\end{aligned} \tag{C.117}$$

RTS. (C.116)

Pick arbitrary κ and $a = (p, q, c) \in \mathcal{I}_0(\kappa)$. From (C.115) and the definition of $\downarrow$ we then have $\mathsf{reflected}(a, s \circ r, \mathcal{I}(\kappa))$. Consequently, from the definition of $\mathsf{reflected}$ we trivially have:

$$\mathsf{reflected}(a, s \circ r, (\mathcal{I} \cup \mathcal{I}_e)(\kappa)) \tag{C.118}$$

Pick an arbitrary $l \in \textsc{IState}$ such that

$$p \circ c \leq s \circ r \circ s_e \circ l \tag{C.119}$$

From (C.118) and the definition of $\mathsf{reflected}$ we then know there exist a', c' such that:

$$a' = (p, q, c') \in (\mathcal{I} \cup \mathcal{I}_e)(\kappa) \land p \circ c' \leq s \circ r \circ s_e \circ l \tag{C.120}$$

Finally, from (C.119), (C.120) and the definition of $\mathsf{reflected}$ we have

$$\mathsf{reflected}(a, s \circ r \circ s_e, (\mathcal{I} \cup \mathcal{I}_e)(\kappa))$$

as required.

RTS. (C.117)
Pick arbitrary κ, $a = (p, q, c) \in (\mathcal{I} \cup \mathcal{I}_e)(\kappa)$ and (s', r') such that:

$$\mathsf{potential}(a, s \circ r \circ s_e) \tag{C.121}$$
$$(s', r') \in a[s, r \circ s_e] \tag{C.122}$$

Since either $a \in \mathcal{I}(\kappa)$ or $a \in \mathcal{I}_e(\kappa)$, there are two cases to consider:

Case 1. $a \in \mathcal{I}(\kappa)$
From the assumption of the case, (C.113), (C.110) and the definition of ► we have:

$$p \perp s_e \tag{C.123}$$

From (C.121) and the definition of $\mathsf{potential}$ we have $\exists l_1, l_2.\ p{\circ}c{\circ}l_1{=}s{\circ}r{\circ}s_e{\circ} l_2$ and $\exists l.\ p{\circ}l{=}s{\circ}r{\circ}s_e{\wedge}q \sharp l$. Consequently, we have $\exists l_1, l_2.\ p{\circ}c{\circ}l_1 = s{\circ}r{\circ}l_2$ and $\exists l.\ p \circ l{=}s \circ r \circ s_e \wedge q \sharp l$.
On the other hand from the cross split property we know there exist p_{sr}, p_e, l_g and l_e such that $p{=}p_{sr} \circ p_e$, $l{=}l_{sr} \circ l_e$, $(s \circ r){=}p_{sr} \circ l_{sr}$ and $s_e{=}p_e \circ l_e$. Since $p \perp s_e$ (C.123), we know $p_e \in \textsc{Unit}_{\textsc{Ins}}$ and thus $p{=}p_{sr}$ and $s \circ r{=}p \circ l_{sr}$. Lastly, since $q \sharp l$, from the definition of l we know that $q \sharp l_{sr}$, and thus we have $\exists l_{sr}.\ p \circ l_{sr}{=}s \circ r \wedge q \sharp l_{sr}$. That is,

$$\mathsf{potential}(a, s \circ r) \tag{C.124}$$

On the other hand, from (C.122), (C.123) and the definitions of $a[s, r \circ s_e]$ and $\perp$, we know there exists r'':

$$r' = r'' \circ s_e \tag{C.125}$$
$$(s', r'') \in a[s, r] \tag{C.126}$$

From (C.126) and the definitions of $a[s, r]$ and $a[s \circ r]$, we know $s' \circ r'' \in a[s \circ r]$. Consequently, from (C.110), (C.113), (C.124), and the definition of ► we have:

$$s' \circ r'' \in \mathcal{F} \tag{C.127}$$

On the other hand, from (C.115), (C.124) and (C.126) we have:

$$\begin{aligned}
&(\mathsf{reflected}(a, s \circ r, \mathcal{I}_0(\kappa)) \lor \neg\mathsf{visible}(a, s)) \land && \text{(C.128)}\\
&\mathcal{I} {\downarrow_m} \left(s', r'', \mathcal{I}_0\right) && \text{(C.129)}
\end{aligned}$$

From (C.114), (C.127), (C.129) and (I.H) we have:

$$\mathcal{I} \cup \mathcal{I}_e {\downarrow_m} \left(s', r'' \circ s_e, \mathcal{I}_0\right)$$

and thus from (C.125)

$$\mathcal{I} \cup \mathcal{I}_e {\downarrow_m} \left(s', r', \mathcal{I}_0\right) \tag{C.130}$$

Consequently, from (C.122), (C.130) we have:

$$\forall (s', r') \in a[s, r \circ s_e].\ \mathcal{I} \cup \mathcal{I}_e {\downarrow_m} \left(s', r', \mathcal{I}_0\right) \tag{C.131}$$

From (C.128) there are two cases to consider:

Case 1.1. $\neg\mathsf{visible}(a, s)$
From (C.131) and the assumption of the case we have:

$$\begin{aligned}
&\neg\mathsf{visible}(a, s) \land\\
&\forall (s', r') \in a[s, r \circ s_e].\ \mathcal{I} \cup \mathcal{I}_e {\downarrow_m} \left(s', r', \mathcal{I}_0\right)
\end{aligned}$$

as required.

Case 1.2. $\mathsf{reflected}(a, s \circ r, \mathcal{I}_0(\kappa))$
Pick an arbitrary $l \in \textsc{IState}$ such that

$$p \circ c \leq s \circ r \circ s_e \circ l \tag{C.132}$$

From the assumption of the case and the definition of reflected we then know there exist a', c' such that:

$$a' = (p, q, c') \wedge a' \in \mathcal{I}_0(\kappa) \wedge p \circ c' \leq s \circ r \circ s_e \circ l \tag{C.133}$$

From (C.132), (C.133) and the definition of $\mathsf{reflected}$ we have:

$$\mathsf{reflected}(a, s \circ r \circ s_e, \mathcal{I}_0(\kappa)) \tag{C.134}$$

Thus from (C.131) and (C.134) we have:

$$\begin{array}{l} \mathsf{reflected}(a, g \circ s_e, \mathcal{I}_0(\kappa)) \wedge \\ \forall (s', r') \in a[s, r \circ s_e].\ \mathcal{I} \cup \mathcal{I}_e \downarrow_m \left(s', r', \mathcal{I}_0\right) \end{array}$$

as required.

Case 2. $a \in \mathcal{I}_e(\kappa)$
From the assumption of the case, (C.114), (C.110) and the definition of $\blacktriangleright$ we have:

$$p \perp s \circ r \tag{C.135}$$

From (C.121) and the definition of $\mathsf{potential}$ we have $\exists l_1, l_2.\ p{\circ}c{\circ}l_1{=}s{\circ}r{\circ}s_e{\circ} l_2$ and $\exists l.\ p{\circ}l{=}s{\circ}r{\circ}s_e{\wedge}q \sharp l$. Consequently, we have $\exists l_1, l_2.\ p{\circ}c{\circ}l_1 = s{\circ}r{\circ}l_2$ and $\exists l.\ p \circ l{=}s \circ r \circ s_e \wedge q \sharp l$.
On the other hand from the cross split property we know there exist p_{sr}, p_e, l_g and l_e such that $p{=}p_{sr} \circ p_e$, $l{=}l_{sr} \circ l_e$, $(s \circ r){=}p_{sr} \circ l_{sr}$ and $s_e{=}p_e \circ l_e$. Since $p \perp s_e$ (C.135), we know $p_{sr} \in \textsc{Unit}_{\textsc{Ins}}$ and thus $p{=}p_e$ and $s_e{=}p \circ l_e$. Lastly, since $q \sharp l$, from the definition of l we know that $q \sharp l_e$, and thus we have $\exists l_e.\ p \circ l_e{=}s_e \wedge q \sharp l_e$. That is,

$$\mathsf{potential}(a, s_e) \tag{C.136}$$

From (C.135) and the definition of $\mathsf{visible}$ we have:

$$\neg\mathsf{visible}(a, s) \tag{C.137}$$

From (C.135) and the definitions of $a[s, r \circ s_e]$ and $\perp$, we know there exists

s'_e such that:

$$s' = s \wedge r' = r \circ s'_e \tag{C.138}$$

$$s'_e \in a[s_e] \tag{C.139}$$

Consequently, from (C.110), (C.114), (C.136), (C.139) and the definition of $\blacktriangleright$ we have:

$$s'_e \in \mathcal{F}_e \tag{C.140}$$

From (C.115), (C.138) and Lemma 27 below we have:

$$\mathcal{I}{\downarrow_m} \left(s', r, \mathcal{I}_0\right) \tag{C.141}$$

From (C.113), (C.140), (C.141), (I.H) we have:

$$\mathcal{I} \cup \mathcal{I}_e{\downarrow_m} \left(s', r \circ s'_e, \mathcal{I}_0\right)$$

and thus from (C.138)

$$\mathcal{I} \cup \mathcal{I}_e{\downarrow_{(n-1)}} \left(s', r', \mathcal{I}_0\right) \tag{C.142}$$

Finally, from (C.122), (C.137) and (C.142) we have:

$$\neg\mathsf{visible}(a, s) \wedge \forall (s', r') \in a[s, r \circ s_e].\ \mathcal{I} \cup \mathcal{I}_e{\downarrow_m} \left(s', r', \mathcal{I}_0\right)$$

as required.

$\square$

Lemma 35 ($\models_\dagger$ monotonicity)**.** *for all* $P \in \textsc{Ast}$, $\Gamma \in \textsc{LEnv}$, $s, g, g' \in \textsc{IState}$ *and* $\mathcal{I}, \mathcal{I}' \in \textsc{AMod}$*:*

$$\Gamma, s \models_{g,\mathcal{I}} P \land g' \copyright \mathcal{I}' \implies \Gamma, s \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P$$

Proof. By induction on the structure of P.

Case $P = p$ **where** $p \in \textsc{LAst}$
Immediate from the semantics of local assertions.

Case $P = P_1 \lor P_2$
Pick an arbitrary $\Gamma \in \textsc{LEnv}$, $s, g, g' \in \textsc{IState}$ and $\mathcal{I}, \mathcal{I}' \in \textsc{AMod}$ such that

$$\Gamma, s \models_{g,\mathcal{I}} P_1 \lor P_2 \tag{C.143}$$

$$g' \copyright \mathcal{I}' \tag{C.144}$$

$$\begin{aligned}&\forall s, g, g' \in \textsc{IState}.\ \forall \mathcal{I}', \mathcal{I} \in \textsc{AMod}.\\ &\quad \Gamma, s \models_{g,\mathcal{I}} P_1 \land g' \copyright \mathcal{I}' \implies \Gamma, s \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P_1\end{aligned} \tag{I.H1}$$

$$\begin{aligned}&\forall s, g, g' \in \textsc{IState}.\ \forall \mathcal{I}', \mathcal{I} \in \textsc{AMod}.\\ &\quad \Gamma, s \models_{g,\mathcal{I}} P_2 \land g' \copyright \mathcal{I}' \implies \Gamma, s \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P_2\end{aligned} \tag{I.H2}$$

From (C.143) and the definition of $\models_{g,\mathcal{I}}$ we know $\Gamma, s \models_{g,\mathcal{I}} P_1$ or $\Gamma, s \models_{g,\mathcal{I}} P_2$; consequently, from (C.144), (I.H1) and (I.H2) we have: $\Gamma, s \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P_1$ or $\Gamma, s \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P_2$. Thus, from the definition of $\models_{g \circ g', \mathcal{I} \cup \mathcal{I}'}$ we have:

$$\Gamma, s \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P_1 \lor P_2$$

as required.

Cases $P{=}\exists x.\ P'$ **or** $P{=}P_1 \land P_2$ **or** $P{=}P_1 * P_2$ **or** $P{=}P_1 \uplus P_2$
These cases are analogous to the previous case and are omitted here.

Case $P \triangleq \boxed{P'}_I$
Pick an arbitrary $\Gamma \in \textsc{LEnv}$, $s, g \in \textsc{IState}$ and $\mathcal{I}, \mathcal{I}' \in \textsc{AMod}$ such that

$$\Gamma, s \models_{g,\mathcal{I}} \boxed{P'}_I \tag{C.145}$$

$$g' \copyright \mathcal{I}' \tag{C.146}$$

$$\begin{aligned}&\forall s, g, g' \in \textsc{IState}.\ \forall \mathcal{I}', \mathcal{I} \in \textsc{AMod}.\\ &\quad \Gamma, s \models_{g,\mathcal{I}} P' \wedge g' \circledcirc \mathcal{I}' \implies \Gamma, s \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P' \end{aligned} \tag{I.H}$$

From (C.145) and the definition of $\models_{g,\mathcal{I}}$ we have:

$$s \in \textsc{Unit}_{\textsc{Ins}} \wedge \exists s', r'.\ g = s' \circ r' \wedge \Gamma, s' \models_{g,\mathcal{I}} P' \wedge \mathcal{I}{\downarrow} \left(s', r', \langle\!| I |\!\rangle_\Gamma\right)$$

Thus from (C.146) and (I.H) we have:

$$\begin{aligned}&s \in \textsc{Unit}_{\textsc{Ins}} \wedge \exists s', r'.\ g = s' \circ r' \wedge \Gamma, s' \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P' \wedge\\ &\mathcal{I}{\downarrow} \left(s', r', \langle\!| I |\!\rangle_\Gamma\right)\end{aligned}$$

Consequently since we have $g \circledcirc \mathcal{I}$ (from the well-formedness of worlds and that $(l, g, \mathcal{I}) \in \textsc{World}$), from (C.146), the definition of $\blacktriangleright$ and Lemma 34 we have:

$$\begin{aligned}&s \in \textsc{Unit}_{\textsc{Ins}} \wedge \exists s', r'.\ g = s' \circ r' \wedge \Gamma, s' \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P' \wedge\\ &\mathcal{I} \cup \mathcal{I}'{\downarrow} \left(s', r' \circ g', \langle\!| I |\!\rangle_\Gamma\right)\end{aligned}$$

After rewriting we have:

$$\begin{aligned}&s \in \textsc{Unit}_{\textsc{Ins}} \wedge \exists s', r'.\ g \circ g' = s' \circ r' \wedge s', \Gamma \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P' \wedge\\ &\mathcal{I} \cup \mathcal{I}'{\downarrow} \left(s', r', \langle\!| I |\!\rangle_\Gamma\right)\end{aligned}$$

That is,

$$\Gamma, s \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} \boxed{P'}_I$$

as required. □

Lemma 36 (Confinement (auxiliary))**.** *for all* $P \in \textsc{Ast}$*,* $\Gamma \in \textsc{LEnv}$*,* $s, g \in \textsc{IState}$ *and* $\mathcal{I}, \mathcal{I}' \in \textsc{AMod}$*:*

$$\Gamma, (s, g, \mathcal{I}) \models P \wedge s \copyright \mathcal{I}' \implies \Gamma, s \models_{g \circ s, \mathcal{I} \cup \mathcal{I}'} P$$

Proof. By induction on the structure of P.

Case $P = p$ **where** $p \in \textsc{LAst}$
Immediate from the semantics of local assertions.

Case $P = P_1 \vee P_2$
Pick an arbitrary $\Gamma \in \textsc{LEnv}$, $s, g \in \textsc{IState}$ and $\mathcal{I}, \mathcal{I}' \in \textsc{AMod}$ such that

$$\Gamma, (s, g, \mathcal{I}) \models P_1 \vee P_2 \tag{C.147}$$

$$g' \copyright \mathcal{I}' \tag{C.148}$$

$$\begin{aligned} &\forall s, g \in \textsc{IState}.\ \forall \mathcal{I}', \mathcal{I} \in \textsc{AMod}. \\ &\quad \Gamma, (s, g, \mathcal{I}) \models P_1 \wedge g' \copyright \mathcal{I}' \implies \Gamma, s \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P_1 \end{aligned} \tag{I.H1}$$

$$\begin{aligned} &\forall s, g \in \textsc{IState}.\ \forall \mathcal{I}', \mathcal{I} \in \textsc{AMod}. \\ &\quad \Gamma, (s, g, \mathcal{I}) \models P_2 \wedge g' \copyright \mathcal{I}' \implies \Gamma, s \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P_2 \end{aligned} \tag{I.H2}$$

From (C.147) and the definition of $\models$ we know $\Gamma, (s, g, \mathcal{I}) \models P_1$ or $\Gamma, (s, g, \mathcal{I}) \models P_2$; consequently, from (C.148), (I.H1) and (I.H2) we have: $\Gamma, s \models_{g \circ s, \mathcal{I} \cup \mathcal{I}'} P_1$ or $\Gamma, s \models_{g \circ s, \mathcal{I} \cup \mathcal{I}'} P_2$. Thus, from the definition of $\models_{g \circ s, \mathcal{I} \cup \mathcal{I}'}$ we have:

$$\Gamma, s \models_{g \circ s, \mathcal{I} \cup \mathcal{I}'} P_1 \vee P_2$$

as required.

Cases $P {=} \exists x.\ P'$ **or** $P {=} P_1 \wedge P_2$ **or** $P {=} P_1 * P_2$ **or** $P {=} P_1 \mathbin{\cup\!\!\!\!\ast} P_2$
These cases are analogous to the previous case and are omitted here.

Case $P \triangleq \boxed{P'}_I$
Pick an arbitrary $\Gamma \in \textsc{LEnv}$, $s, g \in \textsc{IState}$ and $\mathcal{I}, \mathcal{I}' \in \textsc{AMod}$ such that

$$\Gamma, (s, g, \mathcal{I}) \models \boxed{P'}_I \tag{C.149}$$

$$s \circledcirc \mathcal{I}' \tag{C.150}$$

$$\forall s, g \in \text{IState}.\ \forall \mathcal{I}', \mathcal{I} \in \text{AMod}.$$
$$\Gamma, (s, g, \mathcal{I}) \models P' \land s \circledcirc \mathcal{I}' \implies \Gamma, s \models_{g \circ s, \mathcal{I} \cup \mathcal{I}'} P' \tag{I.H}$$

From (C.149) and the definition of $\models$ we have:

$$s \in \text{Unit}_{\text{Ins}} \land \exists s', r'.\ g = s' \circ r' \land \Gamma, s' \models_{g, \mathcal{I}} P' \land \mathcal{I}\!\downarrow \left(s', r', (\langle\!| I |\!\rangle_\Gamma, -)\right)$$

From (C.150) and Lemma 24 we have:

$$s \in \text{Unit}_{\text{Ins}} \land \exists s', r'.\ g = s' \circ r' \land \Gamma, s' \models_{g \circ s, \mathcal{I} \cup \mathcal{I}'} P' \land$$
$$\mathcal{I}\!\downarrow \left(s', r', (\langle\!| I |\!\rangle_\Gamma, -)\right)$$

From the well-formedness of worlds and (C.149) we know $g \circledcirc \mathcal{I}$. As such, from (C.150), the definition of $\blacktriangleright$ and Lemma 34 we have:

$$s \in \text{Unit}_{\text{Ins}} \land \exists s', r'.\ g = s' \circ r' \land \Gamma, s' \models_{g \circ s, \mathcal{I} \cup \mathcal{I}'} P' \land$$
$$\mathcal{I} \cup \mathcal{I}'\!\downarrow \left(s', r' \circ s, (\langle\!| I |\!\rangle_\Gamma, -)\right)$$

After rewriting we have:

$$s \in \text{Unit}_{\text{Ins}} \land \exists s', r'.\ g \circ s = s' \circ r' \land s', \Gamma \models_{g \circ s, \mathcal{I} \cup \mathcal{I}'} P' \land$$
$$\mathcal{I} \cup \mathcal{I}'\!\downarrow \left(s', r', \langle\!| I |\!\rangle_\Gamma\right)$$

That is,

$$\Gamma, s \models_{g \circ s, \mathcal{I} \cup \mathcal{I}'} \boxed{P'}_I$$

as required. □

Lemma 37 (Extend stability)**.** *for all $P \in$ Ast, $\Gamma \in$ LEnv, $w, w' \in$ World,*

$$\Gamma, w \models P \wedge (w, w') \in R^e \implies \Gamma, w' \models P$$

Proof. We proceed by induction on the structure of assertion P.

Case $P = p$ where $p \in$ LAst
Immediate from the semantics of local assertions.

Case $P = P_1 \vee P_2$
Pick arbitrary $\Gamma \in$ LEnv, $l, g, g' \in$ IState and $\mathcal{I}, \mathcal{I}' \in$ AMod such that:

$$(l, g, \mathcal{I}), \Gamma \models P_1 \vee P_2 \tag{C.151}$$

$$g' \copyright \mathcal{I}' \tag{C.152}$$

$$\begin{aligned}&\forall l, g, g' \in \text{IState}.\ \forall \mathcal{I}', \mathcal{I} \in \text{AMod}.\\ &\quad \Gamma, (l, g, \mathcal{I}) \models P_1 \wedge g' \copyright \mathcal{I}' \implies \Gamma, (l, g \circ g', \mathcal{I} \cup \mathcal{I}') \models P_1\end{aligned} \tag{I.H1}$$

$$\begin{aligned}&\forall l, g, g' \in \text{IState}.\ \forall \mathcal{I}', \mathcal{I} \in \text{AMod}.\\ &\quad \Gamma, (l, g, \mathcal{I}) \models P_2 \wedge g' \copyright \mathcal{I}' \implies \Gamma, (l, g \circ g', \mathcal{I} \cup \mathcal{I}') \models P_2\end{aligned} \tag{I.H2}$$

From (C.151) and the definition of $\models$ we know $\Gamma, (l, g, \mathcal{I}) \models P_1$ or $(l, g, \mathcal{I}), \Gamma \models P_2$; consequently, from (C.152), (I.H1) and (I.H2) we have: $\Gamma, (l, g \circ g', \mathcal{I} \cup \mathcal{I}') \models P_1$ or $\Gamma, (l, g \circ g', \mathcal{I} \cup \mathcal{I}') \models P_2$. Thus, from the definition of $\models$ we have $\Gamma, (l, g \circ g', \mathcal{I} \cup \mathcal{I}') \models P_1 \vee P_2$ as required.

Cases $P = \exists x.\ P'$ or $P = P_1 \wedge P_2$ or $P = P_1 * P_2$ or $P = P_1 \uplus P_2$
These cases are analogous to the previous case and are omitted here.

Case $P \triangleq \boxed{P'}_I$
Pick arbitrary $\Gamma \in$ LEnv, $s, g \in$ IState and $\mathcal{I}, \mathcal{I}' \in$ AMod such that

$$(l, g, \mathcal{I}), \Gamma \models \boxed{P'}_I \tag{C.153}$$

$$g' \copyright \mathcal{I}' \tag{C.154}$$

$$\begin{aligned}&\forall l, g, g' \in \text{IState}.\ \forall \mathcal{I}', \mathcal{I} \in \text{AMod}.\\ &\quad \Gamma, (l, g, \mathcal{I}) \models P' \wedge g' \copyright g \implies (l, g \circ g', \mathcal{I} \cup \mathcal{I}'), \Gamma \models P'\end{aligned} \tag{I.H}$$

From (C.153) and the definition of $\models$ we have:

$$l \in \textsc{Unit}_{\textsc{Ins}} \land \exists s', r'.\ g = s' \circ r' \land \Gamma, s' \models_{g,\mathcal{I}} P' \land \mathcal{I}{\downarrow} \left(s', r', \llparenthesis I \rrparenthesis_{\Gamma}\right)$$

Thus from (C.154) and Lemma 35 we have

$$l \in \textsc{Unit}_{\textsc{Ins}} \land \exists s', r'.\ g = s' \circ r' \circ g' \land \Gamma, s' \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P' \land \mathcal{I}{\downarrow} \left(s', r', \llparenthesis I \rrparenthesis_{\Gamma}\right)$$

Consequently since we have $g \circledcirc \mathcal{I}$ (from the well-formedness of $(l, g, \mathcal{I}) \in \textsc{World}$), from (C.154), the $\blacktriangleright$ definition and Lemma 34 we have:

$$\begin{aligned}&l \in \textsc{Unit}_{\textsc{Ins}} \land \exists s', r'.\ g = s' \circ r' \land s', \Gamma \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P' \land \\ &\mathcal{I} \cup \mathcal{I}'{\downarrow} \left(s', r' \circ g', \llparenthesis I \rrparenthesis_{\Gamma}\right)\end{aligned}$$

After rewriting we have:

$$\begin{aligned}&l \in \textsc{Unit}_{\textsc{Ins}} \land \exists s', r'.\ g \circ g' = s' \circ r' \land s', \Gamma \models_{g \circ g', \mathcal{I} \cup \mathcal{I}'} P' \land \\ &\mathcal{I} \cup \mathcal{I}'{\downarrow} \left(s', r', \llparenthesis I \rrparenthesis_{\Gamma}\right)\end{aligned}$$

That is,

$$(l, g \circ g', \mathcal{I} \cup \mathcal{I}'), \Gamma \models \boxed{P'}_{I}$$

as required. $\square$

Lemma 38 (Stability). *Given the update rely R^u (Def. 112), for all $P \in \textsc{Ast}$ (Def. 103), if P is stable with respect to the actions in R^u (Def. 112), then it is stable:*

$$\mathsf{stable}\,(P, R^u) \Rightarrow \mathsf{stable}\,(P)$$

Proof. Let $S \in \mathcal{P}\,(\textsc{World} \times \textsc{World})$ denote a binary relation on worlds defined as follows.

$$S \triangleq \bigcup_{i \in \mathbb{N}} S^i \quad \text{where} \quad \begin{aligned} S^0 &\triangleq R^{\mathsf{e}} \cup R^{\mathsf{u}} \cup \{(w, w) \mid w \in \textsc{World}\} \\ S^{n+1} &\triangleq \left\{(w, w') \mid (w, w'') \in S^0 \land (w'', w) \in S^n\right\}\end{aligned}$$

From the definition of the rely relation on worlds ($\mathcal{R}$) we then have $\mathcal{R} = S$.

It thus suffices to show that for all $n \in \mathbb{N}$ and $P \in \textsc{Ast}$,

$$\mathsf{stable}\,(P, R^{\mathsf{u}}) \Rightarrow \mathsf{stable}\,(P, S^{n})$$

We proceed by induction on n.

Base case $n = 0$
Pick arbitrary $\Gamma \in \textsc{LEnv}$, $w, w' \in \textsc{World}$ and $P \in \textsc{Ast}$ such that:

$$(w, w') \in S^{0} \wedge \Gamma, w \models P \tag{C.155}$$

$$\mathsf{stable}\,(P, R^{\mathsf{u}}) \tag{C.156}$$

We are then required to show:

$$\Gamma, w' \models P$$

From (C.155) and the definition of S^0 there are three cases to consider:

1. If $(w, w') \in \{(w, w) \mid w \in \textsc{World}\}$, then $w' = w$ and from (C.155) we trivially have $\Gamma, w' \models P$.
2. If $(w, w') \in R^{\mathsf{e}}$ then from Lemma 37 and (C.155) we have $\Gamma, w' \models P$.
3. If $(w, w') \in R^{\mathsf{u}}$ then from (C.156) we have $\Gamma, w' \models P$.

Inductive case $n{=}m{+}1$
Pick arbitrary $w, w' \in \textsc{World}$, $\Gamma \in \textsc{LEnv}$ and $P \in \textsc{Ast}$ such that

$$(w, w') \in S^{m+1} \wedge \Gamma, w \models P \tag{C.157}$$

$$\mathsf{stable}\,(P, R^{\mathsf{u}}) \tag{C.158}$$

$$\mathsf{stable}\,(P, R^{\mathsf{u}}) \Rightarrow \mathsf{stable}\,(P, S^{m}) \tag{I.H.}$$

From (C.157) and S^{m+1} we know there exists $w'' \in \textsc{World}$ such that

$$(w, w'') \in S^{0} \wedge (w'', w') \in S^{m} \tag{C.159}$$

From (C.158), (C.159) and the base case we know $\Gamma, w'' \models P$. Consequently, from (C.158), (C.159) and (I.H.) we have $\Gamma, w' \models P$ as required. □

Lemma 39 (Weakening (full proof))**.** *For all $P \in \textsc{Ast}$ (Def. 103):*

$$P \vdash \rfloor P$$

Proof. By induction on the structure of assertions.

Case $P = p$ where $p \in \textsc{LAst}$
Follows immediately from the definition of $\rfloor$ for local assertions.

Case $P = \exists \textsc{x}.\ Q$
Pick arbitrary $\Gamma \in \textsc{LEnv}$ and $w \in \textsc{World}$ such that $\Gamma, w \models \exists \textsc{x}.\ Q$. From the definition of $\models$ we then know there exists v such that $[\Gamma \mid \textsc{x}{:}v], w \models Q$. From the inductive hypothesis we have $[\Gamma \mid \textsc{x}{:}v], w \models \rfloor Q$ and consequently from the definition of $\models$ we have $\Gamma, w \models \rfloor(\exists \textsc{x}.\ Q)$ as required.

Case $P = Q \vee R$
Pick arbitrary $\Gamma \in \textsc{LEnv}$ and $w \in \textsc{World}$ such that $\Gamma, w \models Q \vee R$. From the definition of $\models$ we then know that $\Gamma, w \models Q$ or $\Gamma, w \models R$. From the inductive hypothesis we have $\Gamma, w \models \rfloor Q$ or $\Gamma, w \models \rfloor R$. From the definition of $\models$ we then have $\Gamma, w \models \rfloor Q \vee \rfloor R$. Consequently from the definition of $\rfloor$ we have $\Gamma, w \models \rfloor(Q \vee R)$ as required.

Case $P = Q * R$ or $P = Q \uplus R$
These cases are analogous to the $Q \vee R$ case and are omitted here.

Case $P = \boxed{Q}_I$
Pick arbitrary $\Gamma \in \textsc{LEnv}$ and $w{=}(l, g, \mathfrak{I}) \in \textsc{World}$ such that $\Gamma, w \models \boxed{Q}_I$. From the definition of $\models$ we know that $l \in \textsc{Unit}_{\textsc{Ins}}$. From the definition of $\models$ we then have $\Gamma, w \models \mathsf{emp}$. Consequently from the definition of $\rfloor$ we have $\Gamma, w \models \rfloor \boxed{Q}_I$ as required.

□

Lemma 40 (Boxed assertions)**.** *For all* $\Gamma \in \textsc{LEnv}$*,* $(l, g, \mathfrak{I}) \in \textsc{World}$ *(Def. 102),* $s, g \in \textsc{IState}$ *(Def. 91),* $\mathfrak{I} \in \textsc{AMod}$ *(Def. 94) and* $P^{\mathsf{b}} \in \textsc{BAst}$ *(Def. 118):*

$$\begin{aligned}
\Gamma, (l, g, \mathfrak{I}) \models P^{\mathsf{b}} &\implies l \in \textsc{Unit}_{\textsc{Ins}} \wedge \Gamma, l \models_{g,\mathfrak{I}} P^{\mathsf{b}} && \text{(C.160)}\\
&\quad\; \wedge \forall l' \in \textsc{Unit}_{\textsc{Ins}}.\ \Gamma, (l', g, \mathfrak{I}) \models P^{\mathsf{b}}\\
\Gamma, s \models_{g,\mathfrak{I}} P^{\mathsf{b}} &\implies s \in \textsc{Unit}_{\textsc{Ins}} \wedge \Gamma, (s, g, \mathfrak{I}) \models P^{\mathsf{b}} && \text{(C.161)}\\
&\quad\; \wedge \forall s' \in \textsc{Unit}_{\textsc{Ins}}.\ \Gamma, s' \models_{g,\mathfrak{I}} P^{\mathsf{b}}
\end{aligned}$$

where $\models$ *and* $\models_{g,\mathfrak{I}}$ *denote the satisfiability relations in Def. 104.*

Proof (C.160). Pick an arbitrary $\Gamma \in \textsc{LEnv}$, and $(l, g, \mathfrak{I}) \in \textsc{World}$ such that $\Gamma, (l, g, \mathfrak{I}) \models P^{\mathsf{b}}$. We then proceed by induction on the structure of P^{b}.

Case $P^{\mathsf{b}} = \mathsf{emp}$
Follows immediately from the definitions of $\models$ and $\models_{g,\mathfrak{I}}$ for emp.

Case $P^{\mathsf{b}} = \boxed{Q}_I$
Follows immediately from the definition of $\models$ for $\boxed{Q}_I$.

Case $P^{\mathsf{b}} = P_1^{\mathsf{b}} * P_2^{\mathsf{b}}$
From the definition of $\models$ and composition on worlds we know that there exist $l_1, l_2 \in \textsc{IState}$ such that $l{=}l_1 \circ l_2$; $\Gamma, (l_1, g, \mathfrak{I}) \models P_1^{\mathsf{b}}$ and $\Gamma, (l_2, g, \mathfrak{I}) \models P_2^{\mathsf{b}}$. From the inductive hypotheses we then have:

$$\begin{aligned}
&l_1 \in \textsc{Unit}_{\textsc{Ins}} \wedge \Gamma, l_1 \models_{g,\mathfrak{I}} P_1^{\mathsf{b}} \wedge \forall l_1' \in \textsc{Unit}_{\textsc{Ins}}.\ \Gamma, (l_1', g, \mathfrak{I}) \models P_1^{\mathsf{b}}\\
&l_2 \in \textsc{Unit}_{\textsc{Ins}} \wedge \Gamma, l_2 \models_{g,\mathfrak{I}} P_2^{\mathsf{b}} \wedge \forall l_2' \in \textsc{Unit}_{\textsc{Ins}}.\ \Gamma, (l_2', g, \mathfrak{I}) \models P_2^{\mathsf{b}}
\end{aligned}$$

From the definition of l and $\circ$ we have $l \in \textsc{Unit}_{\textsc{Ins}}$ as required. Similarly from the definitions of l and $\models_{g,\mathfrak{I}}$ we have $\Gamma, l \models_{g,\mathfrak{I}} P_1^{\mathsf{b}} * P_2^{\mathsf{b}}$. For the third conjunct, pick an arbitrary $l' \in \textsc{Unit}_{\textsc{Ins}}$. From the definition of unit set we know there exists a unit element $l_u' \in \textsc{Unit}_{\textsc{Ins}}$ such that $l' \circ l_u'{=}l'$. Since both l' and l_u' are in $\textsc{Unit}_{\textsc{Ins}}$ from above we have:

$$(l', g, \mathfrak{I}) \models P_1^{\mathsf{b}} \wedge (l_u', g, \mathfrak{I}) \models P_2^{\mathsf{b}}$$

Consequently from the definition of $\models$ we have $(l' \circ l_u', g, \mathfrak{I}) \models P_1^{\mathsf{b}} * P_2^{\mathsf{b}}$.

Lastly, since $l' \circ l'_u{=}l'$, we have $(l', g, \mathfrak{I}) \models P^{\mathsf{b}}_1 * P^{\mathsf{b}}_2$ as required. $\square$

Proof (C.161). Pick an arbitrary $\Gamma \in \textsc{LEnv}$, $s, g \in \textsc{IState}$ and $\mathfrak{I} \in \textsc{AMod}$ such that $\Gamma, s \models_{g,\mathfrak{I}} P^{\mathsf{b}}$. We first establish the second conjunct. The first and third conjuncts then follow from the second conjunct and the previous part (C.160). For the second conjunct we proceed by induction on the structure of P^{b}.

Case $P^{\mathsf{b}} = \mathsf{emp}$
Follows immediately from the definition of $\models_{g,\mathfrak{I}}$ for emp.

Case $P^{\mathsf{b}} = \boxed{Q}_I$
From immediately from the definition of $\models_{g,\mathfrak{I}}$ for $\boxed{Q}_I$.

Case $P^{\mathsf{b}} = P^{\mathsf{b}}_1 * P^{\mathsf{b}}_2$
From the definition of $\models_{g,\mathfrak{I}}$ and composition on worlds we know that there exist $s_1, s_2 \in \textsc{IState}$ such that $s{=}s_1 \circ s_2$, $\Gamma, s_1 \models_{g,\mathfrak{I}} P^{\mathsf{b}}_1$ and $\Gamma, s_2 \models_{g,\mathfrak{I}} P^{\mathsf{b}}_2$. From the inductive hypotheses we then have $\Gamma, (s_1, g, \mathfrak{I}) \models P^{\mathsf{b}}_1$ and $\Gamma, (s_2, g, \mathfrak{I}) \models P^{\mathsf{b}}_2$. Consequently, from the definitions of s, and $\models$ we have $\Gamma, (s, g, \mathfrak{I}) \models P^{\mathsf{b}}_1 * P^{\mathsf{b}}_2$ as required. $\square$

Lemma 41 (Boxed assertions (continued))**.** *For all* $\Gamma \in \textsc{LEnv}$*,* $(l, g, \mathfrak{I}) \in \textsc{World}$ *(Def. 102) and* $P^{\mathsf{b}}, Q^{\mathsf{b}} \in \textsc{BAst}$ *(Def. 118):*

$$\Gamma, (l, g, \mathfrak{I}) \models P^{\mathsf{b}} * Q^{\mathsf{b}} \iff \Gamma, (l, g, \mathfrak{I}) \models P^{\mathsf{b}} \circledast Q^{\mathsf{b}}$$

where $\models$ *denotes the satisfiability relation in Def. 104.*

Proof.

The $\Rightarrow$ direction

$$\begin{aligned}
&\Gamma, (l, g, \mathfrak{I}) \models P^{\mathsf{b}} * Q^{\mathsf{b}} \\
\overset{\models \text{Def.}}{\Longrightarrow}\ &\exists l_1, l_2.\ \Gamma, (l_1, g, \mathfrak{I}) \models P^{\mathsf{b}} \land \Gamma, (l_2, g, \mathfrak{I}) \models Q^{\mathsf{b}} \land l{=}l_1 \circ l_2 \\
\overset{\text{Lemma40}}{\Longrightarrow}\ &\exists l_1, l_2.\ \Gamma, (l_1, g, \mathfrak{I}) \models P^{\mathsf{b}} \land \Gamma, (l_2, g, \mathfrak{I}) \models Q^{\mathsf{b}} \land l{=}l_1 \circ l_2 \\
&\land l_1, l_2, l \in \textsc{Unit}_{\textsc{Ins}} \\
\overset{\text{Lemma40}}{\Longrightarrow}\ &\Gamma, (l, g, \mathfrak{I}) \models P^{\mathsf{b}} \land \Gamma, (l, g, \mathfrak{I}) \models Q^{\mathsf{b}} \\
\overset{\models \text{Def.}}{\Longrightarrow}\ &\Gamma, (l, g, \mathfrak{I}) \models P^{\mathsf{b}} \circledast Q^{\mathsf{b}}
\end{aligned}$$

as required.

The $\Leftarrow$ direction

$$\begin{aligned}
&\Gamma, (l, g, \mathfrak{I}) \models P^{\mathsf{b}} \circledast Q^{\mathsf{b}} \\
\overset{\models \text{Def.}}{\Longrightarrow}\ &\exists l_1, l_2, l_3.\ \Gamma, (l_1 \circ l_2, g, \mathfrak{I}) \models P^{\mathsf{b}} \land \Gamma, (l_2 \circ l_3, g, \mathfrak{I}) \models Q^{\mathsf{b}} \\
&\land l{=}l_1 \circ l_2 \circ l_3 \\
\overset{\text{Lemma40}}{\Longrightarrow}\ &\exists l_1, l_2, l_3.\ \Gamma, (l_1 \circ l_2, g, \mathfrak{I}) \models P^{\mathsf{b}} \land \Gamma, (l_2 \circ l_3, g, \mathfrak{I}) \models Q^{\mathsf{b}} \\
&\land l{=}l_1 \circ l_2 \circ l_3 \land l_1 \circ l_2, l_2 \circ l_3 \in \textsc{Unit}_{\textsc{Ins}} \\
\overset{\textsc{Unit}_{\textsc{Ins}}\text{Def.}}{\Longrightarrow}\ &\exists l_1, l_2, l_3.\ \Gamma, (l_1 \circ l_2, g, \mathfrak{I}) \models P^{\mathsf{b}} \land \Gamma, (l_2 \circ l_3, g, \mathfrak{I}) \models Q^{\mathsf{b}} \\
&\land l{=}l_1 \circ l_2 \circ l_3 \land l_1 \circ l_2, l_2 \circ l_3, l_1, l_2, l_3 \in \textsc{Unit}_{\textsc{Ins}} \\
\overset{\text{Lemma40}}{\Longrightarrow}\ &\exists l_1, l_2, l_3.\ \Gamma, (l_1 \circ l_2, g, \mathfrak{I}) \models P^{\mathsf{b}} \land \Gamma, (l_3, g, \mathfrak{I}) \models Q^{\mathsf{b}} \\
&\land l{=}l_1 \circ l_2 \circ l_3 \\
\overset{\models \text{Def.}}{\Longrightarrow}\ &\Gamma, (l, g, \mathfrak{I}) \models P^{\mathsf{b}} * Q^{\mathsf{b}}
\end{aligned}$$

□

Lemma 42 (Flattening normalisation (auxiliary))**.** *For all* $\Gamma \in$ LEnv*,* $l, g \in$ IState *(Def. 91),* $\mathfrak{I} \in$ AMod *(Def. 94) and quantifier-free assertions* $P \in$ Ast *(Def. 103):*

$$\Gamma, l \models_{g,\mathfrak{I}} P \Leftrightarrow \Gamma, l \models_{g,\mathfrak{I}} \mathsf{nf}\,(P)$$

where $\models_{g,\mathfrak{I}}$ *denotes the satisfiability relations in Def. 104.*

Proof. Pick arbitrary $l, g \in$ IState, $\mathfrak{I} \in$ AMod and quantifier-free assertion $P \in$ Ast. We proceed by induction over the structure of P.

Case $P = p$ where $p \in$ LAst
Follows immediately from the $\mathsf{nf}\,(.)$ definition for local assertions ($\mathsf{nf}\,(p) = p$).

Case $P = Q \vee R$

$$\begin{aligned}
\Gamma, l \models_{g,\mathfrak{I}} \models P &\overset{\models \text{Def.}}{\Longleftrightarrow} \Gamma, l \models_{g,\mathfrak{I}} Q \;\;\vee\;\; \Gamma, l \models_{g,\mathfrak{I}} R \\
&\overset{\text{I.H.}}{\Longleftrightarrow} \Gamma, l \models_{g,\mathfrak{I}} \mathsf{nf}\,(Q) \;\;\vee\;\; \Gamma, l \models_{g,\mathfrak{I}} \mathsf{nf}\,(R) \\
&\overset{\models \text{Def.}}{\Longleftrightarrow} \Gamma, l \models_{g,\mathfrak{I}} \mathsf{nf}\,(Q) \vee \mathsf{nf}\,(R) \\
&\overset{\mathsf{nf}(.)\text{ Def.}}{\Longleftrightarrow} \Gamma, l \models_{g,\mathfrak{I}} \mathsf{nf}\,(Q \vee R) \iff \Gamma, l \models_{g,\mathfrak{I}} \mathsf{nf}\,(P)
\end{aligned}$$

Case $P = Q * R$
Let $\mathsf{nf}\,(Q) = \bigvee_{j \in J} \left(q_j * Q^{\mathsf{b}}_j\right)$ and $\mathsf{nf}\,(R) = \bigvee_{k \in K} \left(r_k * R^{\mathsf{b}}_k\right)$. We then have:

$$\begin{aligned}
\Gamma, l \models_{g,\mathfrak{I}} P &\overset{\models \text{Def.}}{\Longleftrightarrow} \exists l_1, l_2.\, l{=}l_1 \circ l_2 \wedge \Gamma, l_1 \models_{g,\mathfrak{I}} Q \;\wedge\; \Gamma, l_2 \models_{g,\mathfrak{I}} R \\
&\overset{\text{I.H.}}{\Longleftrightarrow} \exists l_1, l_2.\, l{=}l_1 \circ l_2 \wedge \Gamma, l_1 \models_{g,\mathfrak{I}} \mathsf{nf}\,(Q) \wedge \Gamma, l_2 \models_{g,\mathfrak{I}} \mathsf{nf}\,(R) \\
&\Longleftrightarrow \exists l_1, l_2.\, l{=}l_1 \circ l_2 \\
&\qquad \wedge \Gamma, l_1 \models_{g,\mathfrak{I}} \bigvee_{j \in J} (q_j * Q^{\mathsf{b}}_j) \wedge \Gamma, l_2 \models_{g,\mathfrak{I}} \bigvee_{k \in K} (r_k * R^{\mathsf{b}}_k) \\
&\overset{\models \text{Def.}}{\Longleftrightarrow} \exists j \in J.\, \exists k \in K.\, \exists l_1, l_2.\, l{=}l_1 \circ l_2 \\
&\qquad \wedge \Gamma, l_1 \models_{g,\mathfrak{I}} q_j * Q^{\mathsf{b}}_j \wedge \Gamma, l_2 \models_{g,\mathfrak{I}} r_k * R^{\mathsf{b}}_k \\
&\overset{\models \text{Def.}}{\Longleftrightarrow} \exists j \in J.\, \exists k \in K.\, \exists l_1, l_2, l_3, l_4.\, l{=}l_1 \circ l_2 \circ l_3 \circ l_4 \\
&\qquad \wedge \Gamma, l_1 \models_{g,\mathfrak{I}} q_j \wedge \Gamma, l_3 \models_{g,\mathfrak{I}} Q^{\mathsf{b}}_j \\
&\qquad \wedge\; \Gamma, l_2 \models_{g,\mathfrak{I}} r_k \wedge \Gamma, l_4 \models_{g,\mathfrak{I}} R^{\mathsf{b}}_k
\end{aligned}$$

$$\begin{aligned}
&\overset{\models\text{Def.,Lem.40}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_0, l_4, l_5.\ l{=}l_0 \circ l_3 \circ l_4 \\
&\qquad \wedge\, l_3, l_4 \in \textsc{Unit}_{\textsc{Ins}} \wedge \Gamma, l_0 \models_{g,\mathfrak{I}} q_j * r_k \\
&\qquad \wedge\ \Gamma, l_3 \models_{g,\mathfrak{I}} Q_j^{\mathsf{b}} \wedge \Gamma, l_4 \models_{g,\mathfrak{I}} R_k^{\mathsf{b}} \\
&\overset{\text{Lemma40}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_0, l_4, l_5.\ l{=}l_0 \circ l_3 \circ l_4 \wedge \Gamma, l_0 \models_{g,\mathfrak{I}} q_j * r_k \\
&\qquad \wedge\ \Gamma, l_3 \models_{g,\mathfrak{I}} Q_j^{\mathsf{b}} \wedge \Gamma, l_4 \models_{g,\mathfrak{I}} R_k^{\mathsf{b}} \\
&\overset{\models\text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \Gamma, l \models_{g,\mathfrak{I}} q_j * r_k * Q_j^{\mathsf{b}} * R_k^{\mathsf{b}} \\
&\overset{\models\text{Def.}}{\iff} \Gamma, l \models_{g,\mathfrak{I}} \bigvee_{j\in J,\, k\in K} \left(q_j * r_k * Q_j^{\mathsf{b}} * R_k^{\mathsf{b}}\right) \\
&\overset{\mathsf{nf}(.)\text{ Def.}}{\iff} \Gamma, l \models_{g,\mathfrak{I}} \mathsf{nf}\,(Q * R) \\
&\iff \Gamma, l \models_{g,\mathfrak{I}} \mathsf{nf}\,(P)
\end{aligned}$$

Case $P = Q \uplus R$

Let $\mathsf{nf}\,(Q) = \bigvee_{j\in J} \left(q_j * Q_j^{\mathsf{b}}\right)$ and $\mathsf{nf}\,(R) = \bigvee_{k\in K} \left(r_k * R_k^{\mathsf{b}}\right)$. We then have:

$$\begin{aligned}
\Gamma, (l, g, \mathfrak{I}) \models P &\overset{\models\text{Def.}}{\iff} \exists l_1, l_2, l_3.\ l{=}l_1 \circ l_2 \circ l_3 \\
&\qquad \wedge \Gamma, l_1 \circ l_2 \models_{g,\mathfrak{I}} Q \ \wedge\ \Gamma, l_2 \circ l_3 \models_{g,\mathfrak{I}} R \\
&\overset{\text{I.H.}}{\iff} \exists l_1, l_2, l_3.\ l{=}l_1 \circ l_2 \circ l_3 \\
&\qquad \wedge \Gamma, l_1 \circ l_2 \models_{g,\mathfrak{I}} \mathsf{nf}\,(Q) \wedge \Gamma, l_2 \circ l_3 \models_{g,\mathfrak{I}} \mathsf{nf}\,(R) \\
&\iff \exists l_1, l_2, l_3.\ l{=}l_1 \circ l_2 \circ l_3 \\
&\qquad \wedge \Gamma, l_1 \circ l_2 \models_{g,\mathfrak{I}} \bigvee_{j\in J} (q_j * Q_j^{\mathsf{b}}) \wedge \Gamma, l_2 \circ l_3 \models_{g,\mathfrak{I}} \bigvee_{k\in K} (r_k * R_k^{\mathsf{b}}) \\
&\overset{\models\text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_1, l_2, l_3.\ l{=}l_1 \circ l_2 \circ l_3 \\
&\qquad \wedge \Gamma, l_1 \circ l_2 \models_{g,\mathfrak{I}} q_j * Q_j^{\mathsf{b}} \wedge \Gamma, l_2 \circ l_3 \models_{g,\mathfrak{I}} r_k * R_k^{\mathsf{b}} \\
&\overset{\models\text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_1, l_2, l_3, l_4, l_5, l_6, l_7.\ l{=}l_1 \circ l_2 \circ l_3 \\
&\qquad \wedge l_1 \circ l_2{=}l_4 \circ l_5 \wedge l_2 \circ l_3{=}l_6 \circ l_7 \\
&\qquad \wedge \Gamma, l_4 \models_{g,\mathfrak{I}} q_j \wedge \Gamma, l_5 \models_{g,\mathfrak{I}} Q_j^{\mathsf{b}} \\
&\qquad \wedge\ \Gamma, l_6 \models_{g,\mathfrak{I}} r_k \wedge \Gamma, l_7 \models_{g,\mathfrak{I}} R_k^{\mathsf{b}}
\end{aligned}$$

$$\overset{\text{Lemma40}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_1, l_2, l_3, l_4, l_5.\ l{=}l_1 \circ l_2 \circ l_3 \circ l_5 \circ l_7$$
$$\land\, l_1 \circ l_2{=}l_1 \circ l_2 \circ l_5 \land l_2 \circ l_3{=}l_2 \circ l_3 \circ l_7 \land l_5, l_7 \in \textsc{Unit}_{\textsc{Ins}}$$
$$\land\, \Gamma, l_1 \circ l_2 \models_{g,\mathcal{I}} q_j \land \Gamma, l_5 \models_{g,\mathcal{I}} Q^{\mathsf{b}}_j$$
$$\land\ \Gamma, l_2 \circ l_3 \models_{g,\mathcal{I}} r_k \land \Gamma, l_7 \models_{g,\mathcal{I}} R^{\mathsf{b}}_k$$
$$\overset{\models\text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_0, l_5, l_7.\ l{=}l_0 \circ l_5 \circ l_7$$
$$\land\, l_5, l_7 \in \textsc{Unit}_{\textsc{Ins}} \land \Gamma, l_0 \models_{g,\mathcal{I}} q_j \uplus r_k$$
$$\land\ \Gamma, l_5 \models_{g,\mathcal{I}} Q^{\mathsf{b}}_j \land \Gamma, l_7 \models_{g,\mathcal{I}} R^{\mathsf{b}}_k$$
$$\overset{\text{Lemma40}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_0, l_5, l_7.\ l{=}l_0 \circ l_5 \circ l_7 \land \Gamma, l_0 \models_{g,\mathcal{I}} q_j \uplus r_k$$
$$\land\ \Gamma, l_5 \models_{g,\mathcal{I}} Q^{\mathsf{b}}_j \land \Gamma, l_7 \models_{g,\mathcal{I}} R^{\mathsf{b}}_k$$
$$\overset{\models\text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \Gamma, l \models_{g,\mathcal{I}} (q_j \uplus r_k) * Q^{\mathsf{b}}_j * R^{\mathsf{b}}_k$$
$$\overset{\models\text{Def.}}{\iff} \Gamma, l \models_{g,\mathcal{I}} \bigvee_{j \in J,\, k \in K} \left((q_j \uplus r_k) * Q^{\mathsf{b}}_j * R^{\mathsf{b}}_k\right)$$
$$\overset{\mathsf{nf}(.)\text{ Def.}}{\iff} \Gamma, l \models_{g,\mathcal{I}} \mathsf{nf}\,(Q \uplus R) \iff \Gamma, l \models_{g,\mathcal{I}} \mathsf{nf}\,(P)$$

Case $P = \boxed{Q}_I$
Let $\mathsf{nf}\,(Q) = \bigvee_{j \in J} \left(q_j * Q^{\mathsf{b}}_j\right)$. We then proceed as follows:

$$\Gamma, l \models_{g,\mathcal{I}} P$$
$$\overset{\models\text{Def.}}{\iff} l \in \textsc{Unit}_{\textsc{Ins}} \land \exists \mathcal{I}'.\ \mathcal{I}'_{\text{A}}{=}\, \langle\!|I|\!\rangle_\Gamma \land \exists s, r.\ g{=}s \circ r$$
$$\land\, \Gamma, s \models_{g,\mathcal{I}} Q \land \mathcal{I}{\downarrow}\left(s, r, \mathcal{I}'\right)$$
$$\overset{\textsc{Unit}_{\textsc{Ins}}}{\iff} \exists l_u, s_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \land \exists s, r.\ s{=}s \circ s_u$$
$$\land\, l \in \textsc{Unit}_{\textsc{Ins}} \land \exists \mathcal{I}'.\ \mathcal{I}'_{\text{A}}{=}\, \langle\!|I|\!\rangle_\Gamma \land g{=}s \circ r$$
$$\land\, \Gamma, s \models_{g,\mathcal{I}} Q \land \mathcal{I}{\downarrow}\left(s, r, \mathcal{I}'\right)$$
$$\overset{\text{I.H.}}{\iff} \exists l_u, s_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \land \exists s, r.\ s{=}s \circ s_u$$
$$\land\, l \in \textsc{Unit}_{\textsc{Ins}} \land \exists \mathcal{I}'.\ \mathcal{I}'_{\text{A}}{=}\, \langle\!|I|\!\rangle_\Gamma \land g{=}s \circ r$$
$$\land\, \Gamma, s \models_{g,\mathcal{I}} \mathsf{nf}\,(Q) \land \mathcal{I}{\downarrow}\left(s, r, \mathcal{I}'\right)$$
$$\overset{\mathsf{nf}(.)\text{ Def.}}{\iff} \exists l_u, s_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \land \exists s, r.\ s{=}s \circ s_u$$
$$\land\, l \in \textsc{Unit}_{\textsc{Ins}} \land \exists \mathcal{I}'.\ \mathcal{I}'_{\text{A}}{=}\, \langle\!|I|\!\rangle_\Gamma \land g{=}s \circ r$$
$$\land\, \Gamma, s \models_{g,\mathcal{I}} \bigvee_{j \in J} \left(q_j * Q^{\mathsf{b}}_j\right) \land \mathcal{I}{\downarrow}\left(s, r, \mathcal{I}'\right)$$

$$\begin{aligned}
\overset{\models\text{Def.}}{\iff}\ & \exists j \in J.\ \exists l_u, s_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \land \exists s, r.\ s{=}s \circ s_u \\
& \land l \in \textsc{Unit}_{\textsc{Ins}} \land \exists \mathcal{J}'.\ \mathcal{J}'_{\mathrm{A}}{=} \llparenthesis I \rrparenthesis_{\Gamma} \land g{=}s \circ r \\
& \land \Gamma, s \models_{g,\mathcal{J}} q_j * Q_j^{\mathsf{b}} \land \mathcal{J}\!\downarrow \left(s, r, \mathcal{J}'\right) \\
\overset{\models\text{Def.}}{\iff}\ & \exists j \in J.\ \exists l_u, s_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \land \exists s, r.\ s{=}s \circ s_u \\
& \land l \in \textsc{Unit}_{\textsc{Ins}} \land \exists \mathcal{J}'.\ \mathcal{J}'_{\mathrm{A}}{=} \llparenthesis I \rrparenthesis_{\Gamma} \land g{=}s \circ r \land \exists s_1, s_2.\ s{=}s_1 \circ s_2 \\
& \land \Gamma, s_1 \models_{g,\mathcal{J}} q_j \land \mathcal{J}\!\downarrow \left(s, r, \mathcal{J}'\right) \land \Gamma, s_2 \models_{g,\mathcal{J}} Q_j^{\mathsf{b}} \\
\overset{\text{Lemma40}}{\iff}\ & \exists j \in J.\ \exists l_u, s_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \land \exists s, r.\ s{=}s \circ s_u \\
& \land l \in \textsc{Unit}_{\textsc{Ins}} \land \exists \mathcal{J}'.\ \mathcal{J}'_{\mathrm{A}}{=} \llparenthesis I \rrparenthesis_{\Gamma} \land g{=}s \circ r \land \exists s_1, s_2.\ s{=}s_1 \circ s_2 \\
& \land s_2 \in \textsc{Unit}_{\textsc{Ins}} \land \Gamma, s_1 \models_{g,\mathcal{J}} q_j \land \mathcal{J}\!\downarrow \left(s, r, \mathcal{J}'\right) \land \Gamma, s_2 \models_{g,\mathcal{J}} Q_j^{\mathsf{b}} \\
\overset{\text{Lemma40}}{\iff}\ & \exists j \in J.\ \exists l_u, s_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \land \exists s, r.\ s{=}s \circ s_u \\
& \land l \in \textsc{Unit}_{\textsc{Ins}} \land \exists \mathcal{J}'.\ \mathcal{J}'_{\mathrm{A}}{=} \llparenthesis I \rrparenthesis_{\Gamma} \land g{=}s \circ r \\
& \land \Gamma, s \models_{g,\mathcal{J}} q_j \land \mathcal{J}\!\downarrow \left(s, r, \mathcal{J}'\right) \land \Gamma, s_u \models_{g,\mathcal{J}} Q_j^{\mathsf{b}} \\
\overset{\text{Lemma40}}{\iff}\ & \exists j \in J.\ \exists l_u, s_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \land \exists s, r.\ s{=}s \circ s_u \\
& \land l \in \textsc{Unit}_{\textsc{Ins}} \land \exists \mathcal{J}'.\ \mathcal{J}'_{\mathrm{A}}{=} \llparenthesis I \rrparenthesis_{\Gamma} \land g{=}s \circ r \\
& \land \Gamma, s \models_{g,\mathcal{J}} q_j \land \mathcal{J}\!\downarrow \left(s, r, \mathcal{J}'\right) \land \Gamma, l_u \models_{g,\mathcal{J}} Q_j^{\mathsf{b}} \\
\overset{\textsc{Unit}_{\textsc{Ins}}}{\iff}\ & \exists j \in J.\ \exists l_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \\
& \land l \in \textsc{Unit}_{\textsc{Ins}} \land \exists \mathcal{J}'.\ \mathcal{J}'_{\mathrm{A}}{=} \llparenthesis I \rrparenthesis_{\Gamma} \land \exists s, r.\ g{=}s \circ r \\
& \land \Gamma, s \models_{g,\mathcal{J}} q_j \land \mathcal{J}\!\downarrow \left(s, r, \mathcal{J}'\right) \land \Gamma, l_u \models_{g,\mathcal{J}} Q_j^{\mathsf{b}} \\
\overset{\models\text{Def.}}{\iff}\ & \exists j \in J.\ \exists l_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \\
& \land \Gamma, (l, g, \mathcal{J}) \models \boxed{q_j}_{I} \ \land\ \Gamma, l_u \models_{g,\mathcal{J}} Q_j^{\mathsf{b}} \\
\overset{\models\text{Def.}}{\iff}\ & \exists j \in J.\ \exists l_u \in \textsc{Unit}_{\textsc{Ins}}.\ l \circ l_u{=}l \\
& \land \Gamma, l \models_{g,\mathcal{J}} \boxed{q_j}_{I} \ \land\ \Gamma, l_u \models_{g,\mathcal{J}} Q_j^{\mathsf{b}} \\
\overset{\text{Lemma40}}{\iff}\ & \exists j \in J.\ \exists l_1, l_2.\ l{=}l_1 \circ l_2 \land \Gamma, l_1 \models_{g,\mathcal{J}} \boxed{q_j}_{I} \ \land\ \Gamma, l_2 \models_{g,\mathcal{J}} Q_j^{\mathsf{b}} \\
\overset{\models\text{Def.}}{\iff}\ & \exists j \in J.\ \Gamma, l \models_{g,\mathcal{J}} \boxed{q_j}_{I} * Q_j^{\mathsf{b}} \\
\overset{\models\text{Def.}}{\iff}\ & \Gamma, l \models_{g,\mathcal{J}} \bigvee_{j \in J} \left(\boxed{q_j}_{I} * Q_j^{\mathsf{b}}\right) \\
\overset{\mathsf{nf}(.)\ \text{Def.}}{\iff}\ & \Gamma, l \models_{g,\mathcal{J}} \mathsf{nf}\left(\boxed{Q}_{I}\right) \iff \Gamma, l \models_{g,\mathcal{J}} \mathsf{nf}\left(P\right)
\end{aligned}$$

$\square$

Lemma 43 (Flattening normalisation (full proof))**.** *For all assertions $P \in$* Ast *(Def. 103) in the prenex normal form:*

$$\frac{}{\vdash P \Leftrightarrow \mathsf{nf}\left(P\right)}$$

where $\mathsf{nf}\left(.\right)$ *denotes the flattening normalisation function in Def. 118.*

Proof. Pick arbitrary $\Gamma \in$ LEnv, $(l, g, \mathfrak{I}) \in$ World (Def. 102) and assertion $\text{Ⓠ}P \in$ Ast (Def. 103) in the prenex normal form with prefix Ⓠ and matrix P. We are then required to show:

$$\Gamma, (l, g, \mathfrak{I}) \models P \Leftrightarrow \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\left(P\right)$$

where P denotes a quantifier-free assertion and $\models$ denotes the satisfiability relations in Def. 104. We proceed by structural induction on P.

Case $P = p$ where $p \in$ LAst
Follows immediately from the $\mathsf{nf}\left(.\right)$ definition for local assertions ($\mathsf{nf}\left(p\right) = p$).

Case $P = Q \vee R$

$$\begin{aligned}
\Gamma, (l, g, \mathfrak{I}) \models P &\overset{\models \text{Def.}}{\iff} \Gamma, (l, g, \mathfrak{I}) \models Q \;\;\vee\;\; \Gamma, (l, g, \mathfrak{I}) \models R \\
&\overset{\text{I.H.}}{\iff} \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\left(Q\right) \;\;\vee\;\; \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\left(R\right) \\
&\overset{\models \text{Def.}}{\iff} \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\left(Q\right) \vee \mathsf{nf}\left(R\right) \\
&\overset{\mathsf{nf}(.)\,\text{Def.}}{\iff} \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\left(Q \vee R\right) \iff \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\left(P\right)
\end{aligned}$$

Case $P = Q * R$
Let $\mathsf{nf}\left(Q\right) = \bigvee_{j \in J} \left(q_j * Q^{\mathsf{b}}_j\right)$ and $\mathsf{nf}\left(R\right) = \bigvee_{k \in K} \left(r_k * R^{\mathsf{b}}_k\right)$. We then have:

$$\begin{aligned}
&\Gamma, (l, g, \mathfrak{I}) \models P \\
\overset{\models \text{Def.}}{\iff}\;& \exists l_1, l_2.\, l{=}l_1 \circ l_2 \wedge \Gamma, (l_1, g, \mathfrak{I}) \models Q \;\wedge\; \Gamma, (l_2, g, \mathfrak{I}) \models R \\
\overset{\text{I.H.}}{\iff}\;& \exists l_1, l_2.\, l{=}l_1 \circ l_2 \wedge \Gamma, (l_1, g, \mathfrak{I}) \models \mathsf{nf}\left(Q\right) \wedge \Gamma, (l_2, g, \mathfrak{I}) \models \mathsf{nf}\left(R\right) \\
\iff\;& \exists l_1,l_2.\, l{=}l_1 \circ l_2 \wedge \Gamma,(l_1,g,\mathfrak{I}) {\models} \bigvee_{j \in J} (q_j {*} Q^{\mathsf{b}}_j) \wedge \Gamma,(l_2,g,\mathfrak{I}) {\models} \bigvee_{k \in K} (r_k {*} R^{\mathsf{b}}_k) \\
\overset{\models \text{Def.}}{\iff}\;& \exists j \in J.\, \exists k \in K.\, \exists l_1, l_2.\, l{=}l_1 \circ l_2 \\
&\quad \wedge \Gamma, (l_1, g, \mathfrak{I}) \models q_j * Q^{\mathsf{b}}_j \wedge \Gamma, (l_2, g, \mathfrak{I}) \models r_k * R^{\mathsf{b}}_k
\end{aligned}$$

$$\overset{\models \text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_1, l_2, l_3, l_4.\ l{=}l_1 \circ l_2 \circ l_3 \circ l_4$$
$$\wedge\, \Gamma, (l_1, g, \mathfrak{I}) \models q_j \wedge \Gamma, (l_3, g, \mathfrak{I}) \models Q^{\mathsf{b}}_j$$
$$\wedge\ \Gamma, (l_2, g, \mathfrak{I}) \models r_k \wedge \Gamma, (l_4, g, \mathfrak{I}) \models R^{\mathsf{b}}_k$$
$$\overset{\models \text{Def.,Lem.40}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_0, l_4, l_5.\ l{=}l_0 \circ l_3 \circ l_4$$
$$\wedge\, l_3, l_4 \in \textsc{Unit}_{\textsc{Ins}} \wedge \Gamma, (l_0, g, \mathfrak{I}) \models q_j * r_k$$
$$\wedge\ \Gamma, (l_3, g, \mathfrak{I}) \models Q^{\mathsf{b}}_j \wedge \Gamma, (l_4, g, \mathfrak{I}) \models R^{\mathsf{b}}_k$$
$$\overset{\text{Lemma40}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_0, l_4, l_5.\ l{=}l_0 \circ l_3 \circ l_4 \wedge \Gamma, (l_0, g, \mathfrak{I}) \models q_j * r_k$$
$$\wedge\ \Gamma, (l_3, g, \mathfrak{I}) \models Q^{\mathsf{b}}_j \wedge \Gamma, (l_4, g, \mathfrak{I}) \models R^{\mathsf{b}}_k$$
$$\overset{\models \text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \Gamma, (l, g, \mathfrak{I}) \models q_j * r_k * Q^{\mathsf{b}}_j * R^{\mathsf{b}}_k$$
$$\overset{\models \text{Def.}}{\iff} \Gamma, (l, g, \mathfrak{I}) \models \bigvee_{j \in J,\, k \in K} \left(q_j * r_k * Q^{\mathsf{b}}_j * R^{\mathsf{b}}_k\right)$$
$$\overset{\mathsf{nf}(.)\ \text{Def.}}{\iff} \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\,(Q * R) \iff \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\,(P)$$

Case $P = Q \uplus R$

Let $\mathsf{nf}\,(Q) = \bigvee_{j \in J} \left(q_j * Q^{\mathsf{b}}_j\right)$ and $\mathsf{nf}\,(R) = \bigvee_{k \in K} \left(r_k * R^{\mathsf{b}}_k\right)$. We then have:

$$\Gamma, (l, g, \mathfrak{I}) \models P$$
$$\overset{\models \text{Def.}}{\iff} \exists l_1, l_2, l_3.\ l{=}l_1 \circ l_2 \circ l_3$$
$$\wedge\, \Gamma, (l_1 \circ l_2, g, \mathfrak{I}) \models Q\ \wedge\ \Gamma, (l_2 \circ l_3, g, \mathfrak{I}) \models R$$
$$\overset{\text{I.H.}}{\iff} \exists l_1, l_2, l_3.\ l{=}l_1 \circ l_2 \circ l_3$$
$$\wedge\, \Gamma, (l_1 \circ l_2, g, \mathfrak{I}) \models \mathsf{nf}\,(Q) \wedge \Gamma, (l_2 \circ l_3, g, \mathfrak{I}) \models \mathsf{nf}\,(R)$$
$$\iff \exists l_1, l_2, l_3.\ l{=}l_1 \circ l_2 \circ l_3$$
$$\wedge\, \Gamma, (l_1 \circ l_2, g, \mathfrak{I}) \models \bigvee_{j \in J} (q_j * Q^{\mathsf{b}}_j) \wedge \Gamma, (l_2 \circ l_3, g, \mathfrak{I}) \models \bigvee_{k \in K} (r_k * R^{\mathsf{b}}_k)$$
$$\overset{\models \text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_1, l_2, l_3.\ l{=}l_1 \circ l_2 \circ l_3$$
$$\wedge\, \Gamma, (l_1 \circ l_2, g, \mathfrak{I}) \models q_j * Q^{\mathsf{b}}_j \wedge \Gamma, (l_2 \circ l_3, g, \mathfrak{I}) \models r_k * R^{\mathsf{b}}_k$$
$$\overset{\models \text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_1, l_2, l_3, l_4, l_5, l_6, l_7.\ l{=}l_1 \circ l_2 \circ l_3$$
$$\wedge\, l_1 \circ l_2{=}l_4 \circ l_5 \wedge l_2 \circ l_3{=}l_6 \circ l_7$$
$$\wedge\, \Gamma, (l_4, g, \mathfrak{I}) \models q_j \wedge \Gamma, (l_5, g, \mathfrak{I}) \models Q^{\mathsf{b}}_j$$
$$\wedge\ \Gamma, (l_6, g, \mathfrak{I}) \models r_k \wedge \Gamma, (l_7, g, \mathfrak{I}) \models R^{\mathsf{b}}_k$$

$$\overset{\text{Lemma40}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_1, l_2, l_3, l_4, l_5.\ l{=}l_1 \circ l_2 \circ l_3 \circ l_5 \circ l_7$$
$$\wedge\, l_1 \circ l_2{=}l_1 \circ l_2 \circ l_5 \wedge l_2 \circ l_3{=}l_2 \circ l_3 \circ l_7 \wedge l_5, l_7 \in \textsc{Unit}_{\textsc{Ins}}$$
$$\wedge\, \Gamma, (l_1 \circ l_2, g, \mathfrak{I}) \models q_j \wedge \Gamma, (l_5, g, \mathfrak{I}) \models Q^{\mathsf{b}}_j$$
$$\wedge\ \Gamma, (l_2 \circ l_3, g, \mathfrak{I}) \models r_k \wedge \Gamma, (l_7, g, \mathfrak{I}) \models R^{\mathsf{b}}_k$$
$$\overset{\models\text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_0, l_5, l_7.\ l{=}l_0 \circ l_5 \circ l_7$$
$$\wedge\, l_5, l_7 \in \textsc{Unit}_{\textsc{Ins}} \wedge \Gamma, (l_0, g, \mathfrak{I}) \models q_j \uplus r_k$$
$$\wedge\ \Gamma, (l_5, g, \mathfrak{I}) \models Q^{\mathsf{b}}_j \wedge \Gamma, (l_7, g, \mathfrak{I}) \models R^{\mathsf{b}}_k$$
$$\overset{\text{Lemma40}}{\iff} \exists j \in J.\ \exists k \in K.\ \exists l_0, l_5, l_7.\ l{=}l_0 \circ l_5 \circ l_7 \wedge \Gamma, (l_0, g, \mathfrak{I}) \models q_j \uplus r_k$$
$$\wedge\ \Gamma, (l_5, g, \mathfrak{I}) \models Q^{\mathsf{b}}_j \wedge \Gamma, (l_7, g, \mathfrak{I}) \models R^{\mathsf{b}}_k$$
$$\overset{\models\text{Def.}}{\iff} \exists j \in J.\ \exists k \in K.\ \Gamma, (l, g, \mathfrak{I}) \models (q_j \uplus r_k) * Q^{\mathsf{b}}_j * R^{\mathsf{b}}_k$$
$$\overset{\models\text{Def.}}{\iff} \Gamma, (l, g, \mathfrak{I}) \models \bigvee_{j \in J,\, k \in K} \left((q_j \uplus r_k) * Q^{\mathsf{b}}_j * R^{\mathsf{b}}_k\right)$$
$$\overset{\mathsf{nf}(.)\,\text{Def.}}{\iff} \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\,(Q \uplus R) \iff \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\,(P)$$

Case $P = \boxed{Q}_I$
Let $\mathsf{nf}\,(Q) = \bigvee_{j \in J} \left(q_j * Q^{\mathsf{b}}_j\right)$. We then proceed as follows:

$$\Gamma, (l, g, \mathfrak{I}) \models P \overset{\models\text{Def.}}{\iff} \Gamma, l \models_{g,\mathfrak{I}} \boxed{Q}_I$$
$$\overset{\text{Lemma42}}{\iff} \Gamma, l \models_{g,\mathfrak{I}} \mathsf{nf}\left(\boxed{Q}_I\right)$$
$$\overset{\mathsf{nf}(.)\,\text{Def.}}{\iff} \Gamma, l \models_{g,\mathfrak{I}} \bigvee_{j \in J} \left(\boxed{q_j}_I * Q^{\mathsf{b}}_j\right)$$
$$\overset{\models\text{Def.}}{\iff} \bigvee_{j \in J} \left(l \models_{g,\mathfrak{I}} \boxed{q_j}_I * Q^{\mathsf{b}}_j\right)$$
$$\overset{\text{Lemma40}}{\iff} \bigvee_{j \in J} \left(\Gamma, (l, g, \mathfrak{I}) \models \boxed{q_j}_I * Q^{\mathsf{b}}_j\right)$$
$$\overset{\models\text{Def.}}{\iff} \Gamma, (l, g, \mathfrak{I}) \models \bigvee_{j \in J} \left(\boxed{q_j}_I * Q^{\mathsf{b}}_j\right) \overset{\mathsf{nf}(.)\,\text{Def.}}{\iff} \Gamma, (l, g, \mathfrak{I}) \models \mathsf{nf}\,(P)$$

□

Lemma 44 (Sequential command soundness)**.** *For all* $\mathtt{C}^{\mathrm{s}} \in \textsc{Seq}$*,* $(L_1, \mathtt{C}^{\mathrm{s}}, L_2) \in \textsc{Ax}_{\mathrm{S}}$ *and* $h \in \mathrm{L}\textsc{State}$*:*

$$\llbracket \mathtt{C}^{\mathrm{s}} \rrbracket_{\mathrm{S}} \left(\lfloor L_1 \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \right) \subseteq \lfloor L_2 \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}}$$

Proof. Pick an arbitrary $h \in \mathrm{L}\textsc{State}$. We proceed by induction over the structure of $\mathtt{C}^{\mathrm{s}}$.

Case $\mathtt{C}^{\mathrm{P}}$
Follows immediately from Par. 30.

Case `skip`

$$\begin{aligned} \llbracket \mathtt{C}^{\mathrm{s}} \rrbracket_{\mathrm{S}} \left(\lfloor L \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \right) = & \; \lfloor L \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \\ \subseteq & \; \lfloor L \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \end{aligned}$$

as required.

Case $\mathtt{C}^{\mathrm{s}}_1 ; \mathtt{C}^{\mathrm{s}}_2$
RTS.

$$\llbracket \mathtt{C}^{\mathrm{s}}_1 ; \mathtt{C}^{\mathrm{s}}_2 \rrbracket_{\mathrm{S}} \left(\lfloor L \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \right) \subseteq \lfloor L' \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}}$$

where $(L, \mathtt{C}^{\mathrm{s}}_1, L''), (L'', \mathtt{C}^{\mathrm{s}}_2, L') \in \textsc{Ax}_{\mathrm{S}}$
Proof.

$$\begin{aligned} \llbracket \mathtt{C}^{\mathrm{s}}_1 ; \mathtt{C}^{\mathrm{s}}_2 \rrbracket_{\mathrm{S}} \left(\lfloor L \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \right) = & \; \llbracket \mathtt{C}^{\mathrm{s}}_2 \rrbracket_{\mathrm{S}} \left(\llbracket \mathtt{C}^{\mathrm{s}}_1 \rrbracket_{\mathrm{S}} \left(\lfloor L \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \right) \right) \\ \text{(I.H.)} \quad \subseteq & \; \llbracket \mathtt{C}^{\mathrm{s}}_2 \rrbracket_{\mathrm{S}} \left(\lfloor L'' \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \right) \\ \text{(I.H.)} \quad \subseteq & \; \lfloor L' \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \end{aligned}$$

as required.

Case $\mathtt{C}^{\mathrm{s}}_1 + \mathtt{C}^{\mathrm{s}}_2$
RTS.

$$\llbracket \mathtt{C}^{\mathrm{s}}_1 + \mathtt{C}^{\mathrm{s}}_2 \rrbracket_{\mathrm{S}} \left(\lfloor L \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \right) \subseteq \lfloor L' \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}}$$

where $(L, \mathtt{C}^{\mathrm{s}}_1, L'), (L, \mathtt{C}^{\mathrm{s}}_2, L') \in \textsc{Ax}_{\mathrm{S}}$.

Proof.

$$\begin{aligned} \llbracket \mathtt{C}^{\mathrm{s}}_1 + \mathtt{C}^{\mathrm{s}}_2 \rrbracket_{\mathrm{S}} \left(\lfloor L \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \right) = & \; \llbracket \mathtt{C}^{\mathrm{s}}_1 \rrbracket_{\mathrm{S}} \left(\lfloor L \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \right) \cup \llbracket \mathtt{C}^{\mathrm{s}}_2 \rrbracket_{\mathrm{S}} \left(\lfloor L \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \right) \\ \text{(I.H.)} \quad \subseteq & \; \lfloor L' \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \cup \lfloor L' \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \\ \subseteq & \; \lfloor L' \bullet_{\mathrm{L}} \{h\} \rfloor_{\mathrm{L}} \end{aligned}$$

as required.

Case $(\mathtt{C}^{\mathrm{s}})^*$

RTS.
$$[\![(\mathtt{C}^{\mathrm{s}})^*]\!]_{\mathrm{S}}\left(\lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\right) \subseteq \lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}$$

where $(L, \mathtt{C}^{\mathrm{s}}, L) \in \mathrm{Ax}_{\mathrm{S}}$.

Proof. Let $(\mathtt{C}^{\mathrm{s}})^0 = \mathtt{skip}$ and $(\mathtt{C}^{\mathrm{s}})^{n+1} = \mathtt{C}^{\mathrm{s}}\mathtt{;}(\mathtt{C}^{\mathrm{s}})^n$. From the definition of $[\![.]\!]_{\mathrm{S}}(.)$ we then have:
$$[\![(\mathtt{C}^{\mathrm{s}})^*]\!]_{\mathrm{S}}(\sigma) = \bigcup_{n\in\mathbb{N}} [\![(\mathtt{C}^{\mathrm{s}})^n]\!]_{\mathrm{S}}(\sigma)$$

It then suffices to show that for all $n \in \mathbb{N}$:
$$[\![(\mathtt{C}^{\mathrm{s}})^n]\!]_{\mathrm{S}}\left(\lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\right) \subseteq \lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}$$

where
$$[\![\mathtt{C}^{\mathrm{s}}]\!]_{\mathrm{S}}\left(\lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\right) \subseteq \lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}} \tag{I.H.}$$

We proceed by induction on n.

Case $n{=}0$
$$\begin{aligned}[\![(\mathtt{C}^{\mathrm{s}})^0]\!]_{\mathrm{S}}\left(\lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\right) &= [\![\mathtt{skip}]\!]_{\mathrm{S}}\left(\lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\right)\\ &= \lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}} \subseteq \lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\end{aligned}$$

as required.

Case $n{=}m{+}1$

From the inductive hypothesis we have:
$$[\![(\mathtt{C}^{\mathrm{s}})^m]\!]_{\mathrm{S}}\left(\lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\right) \subseteq \lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}} \tag{I.H.2}$$

We then proceed as follows:
$$\begin{aligned}[\![(\mathtt{C}^{\mathrm{s}})^{m+1}]\!]_{\mathrm{S}}\left(\lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\right) &= [\![\mathtt{C}^{\mathrm{s}}\mathtt{;}(\mathtt{C}^{\mathrm{s}})^m]\!]_{\mathrm{S}}\left(\lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\right)\\ &= [\![(\mathtt{C}^{\mathrm{s}})^m]\!]_{\mathrm{S}}\left([\![\mathtt{C}^{\mathrm{s}}]\!]_{\mathrm{S}}\left(\lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\right)\right)\\ \text{(I.H.)} \qquad &\subseteq [\![(\mathtt{C}^{\mathrm{s}})^m]\!]_{\mathrm{S}}\left(\lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\right)\\ \text{(I.H.2)} \qquad &\subseteq \lfloor L \bullet_{\mathrm{L}} \{h\}\rfloor_{\mathrm{L}}\end{aligned}$$

as required.

□

Lemma 45 (Update guarantee containment). *For all* $w_1, w_2{=}(l_2, g_2, \mathcal{I}_2), w, w' = (l', g', \mathcal{I}') \in \text{World}$,

$$w_1 \bullet w_2 = w \wedge (l', g', \mathcal{I}') \in G^{\mathsf{u}}(w_1) \implies (l_2, g', \mathcal{I}') \in R^{\mathsf{u}}(w_2)$$

where $R^{\mathsf{u}}(w) \triangleq \{w' \mid (w, w') \in R^{\mathsf{u}}(w)\}$.

Proof. Pick arbitrary $w, w_1{=}(l_1, g_1, \mathcal{I}_1)$, $w_2{=}(l_2, g_2, \mathcal{I}_2)$ and $(l', g', \mathcal{I}')$ such that:

$$w_1 \bullet w_2 = w \tag{C.162}$$

$$(l', g', \mathcal{I}') \in G^{\mathsf{u}}(w_1) \tag{C.163}$$

From (C.162) we know:

$$g_1 = g_2 \tag{C.164}$$

$$\mathcal{I}_1 = \mathcal{I}_2 \tag{C.165}$$

By definition of $G^{\mathsf{u}}$and from (C.163) and (C.165) we know:

$$\mathcal{I}' = \mathcal{I}_1 = \mathcal{I}_2 \tag{C.166}$$

$$(l_1 \circ g_1)_{\text{K}})^{\sharp} = \left((l' \circ g')_{\text{L}}\right)^{\sharp} \tag{C.167}$$

$$g' = g_1 \vee \begin{pmatrix} \exists \kappa \leq (l_1)_{\text{K}}.\ (g_1, g') \in \lceil \mathcal{I}_1 \rceil (\kappa) \wedge \\ ((l_1 \circ g_1)_{\text{L}})^{\sharp} = ((l' \circ g')_{\text{L}})^{\sharp} \end{pmatrix}$$

There are two cases to consider:

Case 1. $g_1 = g'$
From (C.164) and the assumption of the case we know $g' = g_2$. Consequently, from (C.166) we have:

$$((w_2)_{\mathsf{L}}, g', \mathcal{I}') = (l_2, g_2, \mathcal{I}_2) \tag{C.168}$$

By definition of R^{u} and from (C.168) we can conclude:

$$((w_2)_{\mathsf{L}}, g', \mathcal{I}') \in R^{\mathsf{u}}(l_2, g_2, \mathcal{I}_2) \tag{C.169}$$

as required.

Case 2.

$$\exists \kappa \leq (l_1)_{\text{K}}.\ (g_1, g') \in \lceil \mathcal{I}_1 \rceil (\kappa) \tag{C.170}$$

$$((l_1 \circ g_1)_{\text{L}})^{\sharp} = \left((l' \circ g')_{\text{L}}\right)^{\sharp} \tag{C.171}$$

From (C.162), (C.164) and (C.165) we know that

$$w = (l_1 \circ l_2, g_2, \mathcal{I}_2) \tag{C.172}$$

Since $\mathsf{wf}\,(w)$ (by definition of World) and from (C.164) we know:

$$(l_1 \circ l_2 \circ g_2)_{\text{K}} = (l_1)_{\text{K}} \bullet_{\text{K}} (l_2)_{\text{K}} \bullet_{\text{K}} (g_2) = (l_1 \circ g_1)_{\text{K}} \bullet_{\text{K}} (l_2)_{\text{K}} \text{ is defined} \tag{C.173}$$

$$(l_1 \circ l_2 \circ g_1)_{\text{L}} = (l_1 \bullet_{\text{L}} g_1)_{\text{L}} \bullet_{\text{L}} (l_2)_{\text{L}} \text{ is defined} \tag{C.174}$$

Since $\kappa_1 \leq (l_1)_{\text{K}}$ (C.170), from (C.173) and Lemma 26, we know:

$$\kappa \sharp \ (l_2)_{\text{K}} \bullet_{\text{K}} (g_2)_{\text{K}} \tag{C.175}$$

From (C.164), (C.171) and (C.174) we know

$$(l' \circ g')_{\text{L}} \bullet_{\text{L}} (l_2)_{\text{L}} = (l' \circ l_2 \circ g')_{\text{L}} \quad \text{is defined} \tag{C.176}$$

From (C.167) and (C.173) we know

$$(l' \circ g')_{\text{K}} \bullet_{\text{K}} (l_2)_{\text{K}} = (l' \circ l_2 \circ g')_{\text{K}} \quad \text{is defined} \tag{C.177}$$

From (C.176) and (C.177) we know $l'_1 \circ l_2 \circ g'$ is defined and consequently:

$$l_2 \circ g' \quad \text{is defined} \tag{C.178}$$

From (C.165), (C.170), (C.175), (C.178) and by definition of R^{u}, we have:

$$((w_2)_{\mathsf{L}}, g', \mathcal{I}') = (l_2, g_2, \mathcal{I}_2) \in R^{\mathsf{u}}(l_2, s_2, \mathcal{I}_2)$$

as required. □

Lemma 46 (Extension guarantee containment)**.** *For all* $w_1, w_2, w, w' = (l', g', \mathcal{I}') \in \textsc{World}$,

$$w_1 \bullet w_2 = w \land w' \in G^{\mathsf{e}}(w_1) \implies ((w_2)_{\mathsf{L}}, g', \mathcal{I}') \in R^{\mathsf{e}}(w_2)$$

Proof. Pick an arbitrary $w_1 = (l_1, g_1, \mathcal{I}_1)$, $w_2 = (l_2, g_2, \mathcal{I}_2), w$ and $w' = (l', g', \mathcal{I}')$ such that:

$$w_1 \bullet w_2 = w \tag{C.179}$$

$$w' \in G^{\mathsf{e}}(w_1) \tag{C.180}$$

RTS.

$$((w_2)_{\mathsf{L}}, g', \mathcal{I}') \in R^{\mathsf{e}}(w_2)$$

From (C.179) we know:

$$g_1 = g_2 \land \mathcal{I}_1 = \mathcal{I}_2 \tag{C.181}$$

By definition of G^{e} and from (C.180) and (C.181) we know there exists $l_3, l_4, g'' \in \text{I}\textsc{State}$, $\kappa_1, \kappa_2 \in \textsc{Kap}$, $t \in \mathbb{N}$ and $\mathcal{I}_1, \mathcal{I}_2 \in \text{A}\textsc{Mod}$ such that

$$\begin{aligned} & l_1 = l_3 \circ l_4 \land l_1' = l_3 \circ (\mathbf{0}, \kappa_1) \\ & \land\ g'' = l_4 \circ (\mathbf{0}, \kappa_2) \land g' = g_2 \circ g'' \\ & \land\ dom(\kappa_1) \cup dom(\kappa_2) \subseteq \{t\} \\ & \land\ \mathcal{I}_1 {=} (\mathcal{I}_{\text{A}}, \mathcal{I}_{\text{T}} \uplus [t \mapsto 1]) \land \mathcal{I}' = \mathcal{I}_1 \cup \mathcal{I}_2 \end{aligned} \tag{C.182}$$

From (C.182) and the definition of R^{e} we have:

$$((w_2)_{\mathsf{L}}, g', \mathcal{I}') \in R^{\mathsf{e}}(w_2)$$

as required. □

Lemma 47 (Guarantee containment)**.** *For all* $w_1, w_2{=}(l_2, s_2, \mathcal{I}_2), w, w' = (l', g', \mathcal{I}') \in \textsc{World}$,

$$w_1 \bullet w_2 = w \wedge (l', g', \mathcal{I}') \in \mathcal{G}(w_1) \implies (l_2, g', \mathcal{I}') \in \mathcal{R}(w_2)$$

Proof. Pick arbitrary $w_1, w_2{=}(l_2, s_2, \mathcal{I}_2), w, w'{=}(l', g', \mathcal{I}')$ such that:

$$w_1 \bullet w_2 = w \tag{C.183}$$

$$(l_2, g', \mathcal{I}') \in \mathcal{G}(w_1) \tag{C.184}$$

We are then required to show:

$$(l_2, g', \mathcal{I}') \in \mathcal{R}(w_2)$$

From (C.184) and by definition of $\mathcal{G}$ we know:

$$(l', g', \mathcal{I}') \in \left(G^{\mathsf{u}} \cup G^{\mathsf{e}}\right)^* (w_2) \tag{C.185}$$

From (C.183), (C.185) and by Lemmata 45 and 46 we have:

$$(l_2, g', \mathcal{I}') \in \left(R^{\mathsf{u}} \cup R^{\mathsf{e}}\right)^* (w_2)$$

and consequently

$$(l_2, g', \mathcal{I}') \in \mathcal{R}(w_2)$$

as required. $\square$