

**CAPER**  
Automatic Verification with Concurrent Abstract Predicates  
Technical Appendix: Program Logic

Thomas Dinsdale-Young  
Aarhus University, Denmark  
tyoung@cs.au.dk

Pedro da Rocha Pinto  
Imperial College London, UK  
pmd09@doc.ic.ac.uk

Kristoffer Just Andersen  
Aarhus University, Denmark  
kja@cs.au.dk

Lars Birkedal  
Aarhus University, Denmark  
birkedal@cs.au.dk

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Syntax Summary</b>	<b>2</b>
2.1	Object Language . . . . .	2
2.2	Specification Language . . . . .	3
2.3	Spin Lock Example . . . . .	3
<b>3</b>	<b>Syntax of Object Language</b>	<b>4</b>
<b>4</b>	<b>Operational Semantics</b>	<b>5</b>
<b>5</b>	<b>Assertion Logic</b>	<b>10</b>
5.1	Entailment Logic . . . . .	13
<b>6</b>	<b>Specification Logic</b>	<b>13</b>
6.1	Specification Syntax . . . . .	13
6.2	Specification Rules . . . . .	16
6.2.1	Program Specification . . . . .	16
6.2.2	Function Specification . . . . .	16
6.2.3	Statement Specification . . . . .	17
6.2.4	Atomic Statement Specifications . . . . .	18
<b>7</b>	<b>Model</b>	<b>18</b>
<b>8</b>	<b>Interpretation</b>	<b>20</b>
8.1	Interpretation of Specifications . . . . .	23
<b>9</b>	<b>Soundness</b>	<b>28</b>
9.1	Soundness of Specification Logic . . . . .	28
9.1.1	Soundness of Statement Specifications . . . . .	28
9.1.2	Soundness of Atomic Statement Specifications . . . . .	41
9.1.3	Soundness of Program Specifications . . . . .	44

# 1 Introduction

This document contains a formal development of the program logic on which the CAPER tool for automatic verification of fine-grained concurrent programs is built.

The document is structured as follows: Sections 2 and 2.3 gives an overview of how the concrete syntax used in CAPER source files translate to the abstract syntax of the program logic used in the formal development. The abstract syntax and the operational semantics are detailed in Sections 3 and 4. The assertion logic to 6. The semantics of the program logic are developed in 3 stages. Section 7 details the semantic model that the interpretation of Section 8 translates the abstract syntax into. This translation is finally proven sound in Section 9.

Here could go some of the points why this document is long and complicated.

The object language features a few things that are non-standard in the sense of a core, imperative calculus for developing program logics. Really, only the heap commands and basic control flow constructs are important.

But, as a tool for automating verification of programs, it is reasonable to push the object language closer to “the real thing”: modern imperative programming languages and hence the algorithms designed for them make use of local variables and local returns to structure and organize code.

The way that ownership of capabilities is modeled is different from the literature. E.g. Iris is essentially a framework parameterized by choice of a partial commutative monoid that, for a given program or problem can be instantiated “large enough” to model the relationships between capability resources that is desired.

The model of Caper makes a different choice based on two ideas: by not having a parameterized framework of a model, but instead a “large enough” model once and for all, we achieve true modularity of verification. Morally, if two modules are proven correct using Iris, but on two different sets of assumptions, they are not directly composable. This is avoided in Caper.

In addition we have the issue of automation and implementability (and two a lesser extent usability). Products, sums, parameters and permissions give an expressive and convenient vocabulary for expressing many protocols on shared state.

Immediately, the combinators provided do not allow for higher order monoid expressions like Iris.

## 2 Syntax Summary

In this section we give a brief overview of how the concrete syntax of CAPER corresponds to the abstract syntax of the program logic.

### 2.1 Object Language

The object language is a core imperative language with first-order dynamically allocated heap storage in addition to mutable local variables and multiple return points. The latter two features sets CAPER apart from standard core imperative languages used in the literature, but is more representative of modern imperative languages.

$s \in \text{Stmt} ::=$	$s_1; s_2$	sequencing
	$\text{if}(e) \text{ then } \{s_1\} \text{ else } \{s_2\}$	conditional branch
	$\text{while}(e)\{s\}$	zero-or-more iteration
	$x := e$	local assignment
	$x := \text{alloc}(e)$	allocation
	$x := \text{CAS}(e_1, e_2, e_3)$	compare-and-swap
	$[e_1] := e_2$	heap write
	$x := [e]$	heap read
	$x := f(\vec{e})$	function call
	$\text{return } e$	value return
	$\text{fork } f(\vec{e})$	spawn thread
	$\text{skip}$	no-op

FunDef ::= function  $f(\vec{x})\{s\}$

Program ::= FunDef\*;function main(){s}

## 2.2 Specification Language

The assertion language is a standard, first-order intuitionistic separation logic extended with resources representing knowledge and ownership of shared state and permissions to update that state. Shared state is modeled by “regions”, and permissions to manipulate the state is expressed through “guards”. A region declaration consists of a guard algebra declaration (GAD), a region interpretation declaration (ID) and an action declaration (AD). The guard algebra describes which permissions can be owned for a particular region, and how those permissions may be shared, split and combined. The region interpretation describes by way of a “state variable” the configuration of the heap owned by the region. The action interpretation describes the protocol governing the shared state by assigning legal updates to the state variable to guards.

The  $P$  in the grammar ranges over assertions in first-order separation logic extended with guard and region assertions as detailed later in this appendix.

$GAD\ g ::=$	$G$ $\%G$ $\#G$ $g * g$ $g + g$ $0$	Indivisible Guard Divisible Guard Parameterized Guard Product Construction Sum Construction Nil Guard	$ID\ i ::=$	$\cdot$ $i, (\Delta). \Pi \mid e : P$	$AD\ a ::=$	$\cdot$ $a, (\Delta). \Pi \mid G : e_1 \rightsquigarrow e_2$
--------------	--	--	-------------	--	-------------	---

Well-specified programs and individual functions are specified with respect to a region declaration context:

$$RegionContext\ R ::= \bullet \mid R, \mathbb{T}(r, \vec{x})(g, i, a)$$

The variable  $r$  binds the name of the region itself (think this from mainstream OO programming languages) and the variables  $\vec{x}$  bind the names of the resources held by the region. These names are bound in the region interpretation and action declaration.

Well-specified programs are also specified with respect to a function specification context, assigning pre- and postconditions to function names, expressed as assertions ranging over the parameters of the function and its return value.

$$SpecCtxt\ \Phi ::= \bullet \mid \Phi, f : (\Gamma, \vec{x})\{P\}\{r.Q\}$$

Individual functions and fragments of code are verified “Hoare style”, demonstrating that programs run from states satisfying a precondition  $P$  either run to completion, reaching a state satisfying  $Q$ , or return locally a value satisfying  $U$ :

$$R; \Phi; \Gamma \vdash \{P\} s \{Q \mid r.U\}$$

There is no surface syntax for this judgment as constructing them is precisely the function of CAPER.

## 2.3 Spin Lock Example

In order to demonstrate the specification syntax of CAPER we here describe a fully verified program, relating it to the surface syntax in the process. The example involves a simple spin-lock, for simplicity, but it will exercise most all the features of the program logic.

First, the surface syntax of the region declarations for the spin-lock:

```
region SLock(r,x) {
  guards %LOCK * UNLOCK;
  interpretation {
    0 : x |-> 0 &&& r@UNLOCK;
    1 : x |-> 1;
```

```

}
actions {
  LOCK[_] : 0 ~> 1;
  UNLOCK : 1 ~> 0;
}
}

```

The set of primitive guards is  $\{\text{LOCK}, \text{UNLOCK}\}$ , and the guard algebra declaration, call it  $G_{SL}$ ,  $\% \text{LOCK} * \text{UNLOCK}$ , describing that the LOCK resource is divisible, while the UNLOCK resource is not.

The region interpretation,  $I_{SL}$ , is described by two clauses, in abstract syntax written as

$$\begin{aligned}
(\cdot). \top \mid 0 : x \mapsto 0 * r @ \text{UNLOCK}, \\
(\cdot). \top \mid 1 : x \mapsto 1
\end{aligned}$$

The  $(\cdot)$  is an empty context of local variables, as only  $x$  and  $r$  occur in the clauses, and  $\top$  is the trivial condition on the clause. Hence, the only requirement of applicability for the clauses is that the state of the region is equal to 0 or 1, modeling locked or unlocked, respectively.

The action declarations describes a simple state transition system, where wielding the LOCK resource (with any degree of permission) allows a state transition from 0 to 1; vice versa for UNLOCK. The abstract syntax is as follows:

$$\begin{aligned}
(\cdot). \top \mid \text{LOCK}[_] : 0 \rightsquigarrow 1 \\
(\cdot). \top \mid \text{UNLOCK} : 1 \rightsquigarrow 0
\end{aligned}$$

Hence, the region declaration context for this example consists of the single SLock declaration:

$$R := \text{SLock}(r, x)(G_{SL}, I_{SL}, A_{SL})$$

The concrete syntax for the unlock implementation is as follows:

```

function unlock(x)
  requires SLock(r, x, 1) &&& r@UNLOCK;
  ensures SLock(r, x, _); {
    [x] := 0;
  }
}

```

This represents the function specification written as follows:

$$\text{unlock} : (r : \text{Region}, x : \text{Val}) \{ \text{SLock}(r, x, 1) * r @ \text{UNLOCK} \} \{ \text{ret}. \exists s. \text{SLock}(r, x, s) \}$$

### 3 Syntax of Object Language

**Convention 1** (Sets). Sets are typeset with capitalized italicized font when there is no canonical name, e.g. *Type*, *Heap* but  $\mathbb{N}$  and  $\mathbb{Z}$ .

**Convention 2** (Sets of Syntax). We typeset sets of syntax with capitalized teletype, e.g. **Fun**.

**Definition 3** (Var, Variable Names). We suppose a set **Var** ranged over by  $x, y$  etc.

**Definition 4** (Fun, Function Names). We suppose a set **Fun**, ranged over by  $f, g$  etc.

**Definition 5** (Expr, Syntactic Expressions). We define **Expr** as arithmetic and boolean expressions over integer literals and variables drawn from **Var**. We let  $e$  range over syntactic expressions.

$e \in \text{Expr}$	$::=$	$x$	variables
		$n$	constants
		$e + e$	arithmetic operations
		$e - e$	
		$e * e$	
		$e / e$	integer comparisons
		$e = e$	
		$e \neq e$	
		$e < e$	
		$e \leq e$	
		$e \geq e$	
		$e > e$	

**Convention 6** (Finite Sequences). We use the notation  $Foo^*$  as the set of finite sequences with elements drawn from  $Foo$ . Individual sequences are denoted  $\vec{e} \in Foo^*$  where each  $e_i \in Foo$  is an elements of the indicated type. We use  $|\vec{e}|$  as notation for the length of  $\vec{e}$ .

We implicitly lift operations on the underlying set to sequences on that set pointwise.

**Definition 7** (Stmt, Syntactic Statements). The set of statements of the object language is generated by the following BNF:

$s \in \text{Stmt}$	$::=$	$s_1; s_2$	sequencing
		$\text{if}(e) \text{ then } \{s_1\} \text{ else } \{s_2\}$	conditional branch
		$\text{while}(e)\{s\}$	zero-or-more iteration
		$x := e$	local assignment
		$x := \text{alloc}(e)$	allocation
		$x := \text{CAS}(e_1, e_2, e_3)$	compare-and-swap
		$[e_1] := e_2$	heap write
		$x := [e]$	heap read
		$x := f(\vec{e})$	function call
		$\text{return } e$	value return
		$\text{fork } f(\vec{e})$	spawn thread
		$\text{skip}$	no-op

**Definition 8** (FunDef, Syntactic Function Definitions). We define FunDef by the following grammar:

$$\text{FunDef} ::= \text{function } f(\vec{x})\{s\}$$

**Definition 9** (Program, Caper Programs). The set of syntactic Caper Programs is defined as the set of sequences of function definitions with a designated main function as the entry point, i.e.

$$\text{Program} := \text{FunDef}^*; \text{function main}()\{s\}$$

## 4 Operational Semantics

**Definition 10** (Addr, Machine Addresses).

$$n \in \text{Addr} := \mathbb{N}$$

**Definition 11** (Val, Program Values).

$$n \in \text{Val} := \mathbb{Z}$$

Note that  $\text{Addr} \subseteq \text{Val}$ . We can distinguish valid addresses in our meta-mathematics simply by considering  $n \in \text{Addr}$  versus  $n \notin \text{Addr}$ .

**Definition 12** (Heap, Global Machine Memory).

$$h \in \text{Heap} := \text{Addr} \stackrel{\text{fin}}{\rightarrow} \text{Val}$$

Heaps form a partial commutative monoid with the empty heap, written  $\emptyset$  as unit and disjoint union as composition, defined as follows:

$$h_1 \uplus h_2 = \begin{cases} \perp & \text{dom}(h_1) \cap \text{dom}(h_2) \neq \emptyset \\ \left( \lambda a. \begin{cases} h_1(a) & a \in \text{dom}(h_1) \\ h_2(a) & a \in \text{dom}(h_2) \\ \perp & \text{otherwise} \end{cases} \right) & \text{otherwise} \end{cases}$$

As always, a partial commutative monoid is partially ordered by the canonical extension order:

$$h_1 \sqsubseteq h_2 \stackrel{\text{def}}{\iff} \exists h_3. h_1 \uplus h_3 = h_2$$

Observe that  $\sqsubseteq$  is both

1. *reflexive*, as  $\emptyset \uplus h = h \uplus \emptyset = h$  for all  $h$ ;
2. *transitive*, as if we have heaps to extend  $h_1$  to  $h_2$  and  $h_2$  to  $h_3$ , the composition of the two pieces will extend  $h_1$  to  $h_3$ , since  $\uplus$  is associative;
3. *anti-symmetric*, as having heaps to extend  $h_1$  to  $h_2$  and vice versa implies that the pieces are empty, hence  $h_1 = h_2$ . That is, if  $h_1 \cdot h'_1 = h_2$  and  $h_2 \cdot h'_2 = h_1$  it means that  $(h_2 \cdot h'_2) \cdot h'_1 = h_2$ . If  $h_2 \cdot (h'_2 \cdot h'_1) = h_2$  then it must be that  $h'_2 \cdot h'_1 = \emptyset$ , meaning so must both of  $h'_2$  and  $h'_1$ .

**Definition 13** (*Stack, Thread Local Stack*).

$$\sigma \in \text{Stack} := \text{Var} \rightarrow \text{Val}$$

We silently lift the action of stacks on variables to sequences of variables when the intention is clear, e.g.  $\sigma(\vec{x}) = \sigma(x_1), \sigma(x_2), \dots, \sigma(x_n)$  for a sequence  $\vec{x}$  of length  $n$ .

**Definition 14** (*Cont, Continuations*). We define the set of continuations by the following grammar, where  $\sigma \in \text{Stack}$ ,  $s \in \text{Stmt}$  and  $\mathbf{x} \in \text{Var}$ :

$$\kappa \in \text{Cont} ::= s \mid \mathbf{x} := (\sigma, \kappa)$$

We use the second construction to represent the frames of the call stack.

**Definition 15** (*ThreadId, Thread Identifiers*). We suppose a set *ThreadId* of thread identifiers, ranged over by *id*.

**Definition 16** (*Thread, Threads*).

$$t \in \text{Thread} := \text{Stack} \times \text{Cont}$$

**Definition 17** (*ThreadPool, Thread pools*).

$$T \in \text{ThreadPool} := \text{ThreadId} \overset{\text{fin}}{\times} \text{Thread}$$

**Definition 18** (*Program, Machine Configurations*). We enrich machine configurations with a distinguished faulting state denoted  $\downarrow$ :

$$\text{Program} := (\text{Heap} \uplus \{\downarrow\}) \times \text{ThreadPool}$$

**Definition 19** (*Env, Function Environments*). Function environments are partial mappings from function names to parameters and bodies. The set of function environments are generated by the following grammar, where  $\mathbf{f} \in \text{Fun}$ ,  $\vec{x} \in \text{Var}^*$  and  $\mathbf{s} \in \text{Stmt}$ :

$$E \in \text{Env} ::= \cdot \mid E, \mathbf{f} : (\vec{x}, \mathbf{s})$$

where  $\mathbf{f}$  does not appear in the  $E$  it extends. An environment  $E$  induces a partial map

$$\text{Fun} \overset{\text{fin}}{\times} (\text{Var}^*, \text{Stmt})$$

and we denote a well-defined lookup as  $E(\mathbf{f}) = (\vec{x}, \mathbf{s})$ .

**Definition 20** ( $\llbracket - \rrbracket_{\sigma}$ , Expression Evaluation). We define a *total* interpretation of expressions with regards to a stack as follows:

$$\llbracket - \rrbracket_{\sigma} : \text{Expr} \times \text{Stack} \rightarrow \text{Val}$$

$$\begin{aligned} \llbracket \mathbf{x} \rrbracket_{\sigma} &= \sigma(\mathbf{x}) \\ \llbracket \mathbf{e}_1 + \mathbf{e}_2 \rrbracket_{\sigma} &= \llbracket \mathbf{e}_1 \rrbracket_{\sigma} + \llbracket \mathbf{e}_2 \rrbracket_{\sigma} \\ \llbracket \mathbf{e}_1 / \mathbf{e}_2 \rrbracket_{\sigma} &= \begin{cases} \llbracket \mathbf{e}_1 \rrbracket_{\sigma} / \llbracket \mathbf{e}_2 \rrbracket_{\sigma} & \text{if } \llbracket \mathbf{e}_2 \rrbracket_{\sigma} \neq 0 \\ 0 & \text{otherwise} \end{cases} \\ \llbracket \mathbf{e}_1 \leq \mathbf{e}_2 \rrbracket_{\sigma} &= \begin{cases} 1 & \text{if } \llbracket \mathbf{e}_1 \rrbracket_{\sigma} \leq \llbracket \mathbf{e}_2 \rrbracket_{\sigma} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

We omit the remaining, straightforward cases.

Notice that uninitialized variables are not handled exceptionally as the stack is a *total* function. Intuitively, referencing uninitialized local variables returns an arbitrary but consistent value - multiple accesses of the same variables returns the same value.

We present the operational semantics as a Views-style labelled transition system, with a label interpretation providing the semantics of atomic actions: heap operations and faulting.

**Definition 21** (*Action*, Atomic Action Labels). The set of atomic action labels is described by the following grammar, where  $n$  ranges over *Addr* and  $v$  ranges over *Val*.

$$\begin{aligned} \alpha \in \text{Action} ::= & \text{id} \\ & | \downarrow \\ & | \text{alloc}(n, v) \\ & | \text{read}(n, v) \\ & | \text{write}(n, v) \\ & | \text{CAS}(n, v, v, v) \end{aligned}$$

**Convention 22** ( $- \mapsto -$ , Map Extension). We denote the extension of maps by the usual  $m[k \mapsto v]$  notation, and lift it to sequences of matching lengths, using  $m[\vec{k} \mapsto \vec{v}]$  to stand for  $m[k_1 \mapsto v_1] \dots [k_n \mapsto v_n]$ , with circumstances ensuring matching lengths.

**Definition 23** ( $\llbracket - \rrbracket$ , Atomic Action Interpretation). We interpret atomic action labels as functions from atomic action labels to heaps or a designated faulting element  $\downarrow$ .

$$\llbracket - \rrbracket : \text{Action} \rightarrow \text{Heap} \rightarrow \mathcal{P}(\text{Heap} \uplus \{\downarrow\})$$

$$\begin{aligned}
\llbracket \text{id} \rrbracket (h) &= \{h\} \\
\llbracket \downarrow \rrbracket (h) &= \{\downarrow\} \\
\llbracket \text{read}(x, v) \rrbracket (h) &= \begin{cases} \{h\} & \text{if } x \in \text{dom}(h) \text{ and } h(x) = v \\ \emptyset & \text{if } x \in \text{dom}(h) \text{ and } h(x) \neq v \\ \{\downarrow\} & \text{if } x \notin \text{dom}(h) \end{cases} \\
\llbracket \text{write}(x, v) \rrbracket (h) &= \begin{cases} \{h[x \mapsto v]\} & \text{if } x \in \text{dom}(h) \\ \{\downarrow\} & \text{otherwise} \end{cases} \\
\llbracket \text{CAS}(x, v, v', b) \rrbracket (h) &= \begin{cases} \{h[x \mapsto v']\} & \text{if } x \in \text{dom}(h), h(x) = v \text{ and } b \neq 0 \\ \{h\} & \text{if } x \in \text{dom}(h), h(x) \neq v \text{ and } b = 0 \\ \emptyset & \text{if } x \in \text{dom}(h) \text{ and } h(x) = v \text{ but } b = 0, \text{ or} \\ & h(x) \neq v \text{ and } b \neq 0 \\ \{\downarrow\} & \text{if } x \notin \text{dom}(h) \end{cases} \\
\llbracket \text{alloc}(v, n) \rrbracket (h) &= \begin{cases} \{h[n \mapsto \_]\dots[n + (v - 1) \mapsto \_]\} & \text{if } n, \dots, n + (v - 1) \notin \text{dom}(h) \\ & \text{and } v > 0 \\ \emptyset & \text{if } n \text{ or } n + 1 \text{ or } \dots \text{ or } n + (v - 1) \\ & \in \text{dom}(h) \text{ and } v > 0 \\ \{\downarrow\} & \text{if } v < 1 \end{cases}
\end{aligned}$$

We lift a function

$$\llbracket \alpha \rrbracket : \text{Heap} \rightarrow \mathcal{P}(\text{Heap} \uplus \{\downarrow\})$$

to

$$(\text{Heap} \uplus \{\downarrow\}) \rightarrow \mathcal{P}(\text{Heap} \uplus \{\downarrow\})$$

by letting  $\llbracket \alpha \rrbracket (\downarrow) = \{\downarrow\}$ .

**Definition 24** ( $- \vdash - \xrightarrow{-}$ , Thread Semantics). We define the semantics of individual threads as a labelled transition relation, with the set of labels consisting of atomic action labels extended by a designated  $\text{fork}(f, \vec{v})$  label; letting

$$\text{Label} := \text{Action} + \{\text{fork}(f, \vec{v}) \mid f \in \text{Fun}, \vec{v} \in \text{Val}^*\}$$

we define the relation

$$- \vdash - \xrightarrow{-} - \subset \text{Env} \times \text{Thread} \times \text{Label} \times \text{Thread}$$

$$\begin{array}{c}
\text{SEQ} \\
\frac{E \vdash (\sigma, \mathbf{s}_1) \xrightarrow{\alpha} (\sigma', \mathbf{s}'_1)}{E \vdash (\sigma, \mathbf{s}_1; \mathbf{s}_2) \xrightarrow{\alpha} (\sigma', \mathbf{s}'_1; \mathbf{s}_2)} \\
\\
\text{IFTRUE} \\
\frac{\llbracket \mathbf{e} \rrbracket_{\sigma} \neq 0}{E \vdash (\sigma, \text{if}(\mathbf{e}) \text{ then } \{\mathbf{s}_1\} \text{ else } \{\mathbf{s}_2\}) \xrightarrow{\text{id}} (\sigma, \mathbf{s}_1)} \\
\\
\text{WHILETRUE} \\
\frac{\llbracket \mathbf{e} \rrbracket_{\sigma} \neq 0}{E \vdash (\sigma, \text{while}(\mathbf{e})\{\mathbf{s}\}) \xrightarrow{\text{id}} (\sigma, \mathbf{s}; \text{while}(\mathbf{e})\{\mathbf{s}\})} \\
\\
\text{FUNCTIONCALL} \\
\frac{E(\mathbf{f}) = (\vec{\mathbf{x}}, \mathbf{s}) \quad \sigma'(\vec{\mathbf{x}}) = \llbracket \vec{\mathbf{e}} \rrbracket_{\sigma}}{E \vdash (\sigma, \mathbf{x} := \mathbf{f}(\vec{\mathbf{e}})) \xrightarrow{\text{id}} (\sigma, \mathbf{x} := (\sigma', \mathbf{s}))} \\
\\
\text{LOCALRETURN} \\
\frac{}{E \vdash (\sigma, \mathbf{x} := (\sigma', \text{return } \mathbf{e}; \mathbf{s})) \xrightarrow{\text{id}} (\sigma[\mathbf{x} \mapsto \llbracket \mathbf{e} \rrbracket_{\sigma'}], \text{skip})} \\
\\
\text{RETURN} \\
\frac{}{E \vdash (\sigma, \mathbf{x} := (\sigma', \text{return } \mathbf{e})) \xrightarrow{\text{id}} (\sigma[\mathbf{x} \mapsto \llbracket \mathbf{e} \rrbracket_{\sigma'}], \text{skip})} \\
\\
\text{DEFAULTRETURN} \\
\frac{}{E \vdash (\sigma, \mathbf{x} := (\sigma', \text{skip})) \xrightarrow{\text{id}} (\sigma[\mathbf{x} \mapsto v], \text{skip})} \\
\\
\text{LOCALASSIGN} \\
\frac{}{E \vdash (\sigma, \mathbf{x} := \mathbf{e}) \xrightarrow{\text{id}} (\sigma[\mathbf{x} \mapsto \llbracket \mathbf{e} \rrbracket_{\sigma}], \text{skip})} \\
\\
\text{ATOMICWRITE} \\
\frac{\llbracket \mathbf{e}_1 \rrbracket_{\sigma} = n \quad n \in \text{Addr}}{E \vdash (\sigma, [\mathbf{e}_1] := \mathbf{e}_2) \xrightarrow{\text{write}(n, \llbracket \mathbf{e}_2 \rrbracket_{\sigma})} (\sigma, \text{skip})} \\
\\
\text{ATOMICREAD} \\
\frac{\llbracket \mathbf{e} \rrbracket_{\sigma} = n \quad n \in \text{Addr}}{E \vdash (\sigma, \mathbf{x} := [\mathbf{e}]) \xrightarrow{\text{read}(n, v)} (\sigma[\mathbf{x} \mapsto v], \text{skip})} \\
\\
\text{CAS} \\
\frac{\llbracket \mathbf{e}_1 \rrbracket_{\sigma} = n \quad n \in \text{Addr}}{E \vdash (\sigma, \mathbf{x} := \text{CAS}(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)) \xrightarrow{\text{CAS}(n, \llbracket \mathbf{e}_2 \rrbracket_{\sigma}, \llbracket \mathbf{e}_3 \rrbracket_{\sigma}, b)} (\sigma[\mathbf{x} \mapsto b], \text{skip})} \\
\\
\text{CASFAULT} \\
\frac{\llbracket \mathbf{e}_1 \rrbracket_{\sigma} \notin \text{Addr}}{E \vdash (\sigma, \mathbf{x} := \text{CAS}(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)) \xrightarrow{\not\rightarrow} (\sigma, \text{skip})} \\
\\
\text{ALLOC} \\
\frac{\llbracket \mathbf{e} \rrbracket_{\sigma} = n \quad 1 \leq n}{E \vdash (\sigma, \mathbf{x} := \text{alloc}(\mathbf{e})) \xrightarrow{\text{alloc}(n, v)} (\sigma[\mathbf{x} \mapsto v], \text{skip})} \\
\\
\text{ALLOCFULT} \\
\frac{\llbracket \mathbf{e} \rrbracket_{\sigma} < 1}{E \vdash (\sigma, \mathbf{x} := \text{alloc}(\mathbf{e})) \xrightarrow{\not\rightarrow} (\sigma, \text{skip})} \\
\\
\text{SKIP} \\
\frac{}{E \vdash (\sigma, \text{skip}; \mathbf{s}) \xrightarrow{\text{id}} (\sigma, \mathbf{s})} \\
\\
\text{IFFALSE} \\
\frac{\llbracket \mathbf{e} \rrbracket_{\sigma} = 0}{E \vdash (\sigma, \text{if}(\mathbf{e}) \text{ then } \{\mathbf{s}_1\} \text{ else } \{\mathbf{s}_2\}) \xrightarrow{\text{id}} (\sigma, \mathbf{s}_2)} \\
\\
\text{WHILEFALSE} \\
\frac{\llbracket \mathbf{e} \rrbracket_{\sigma} = 0}{E \vdash (\sigma, \text{while}(\mathbf{e})\{\mathbf{s}\}) \xrightarrow{\text{id}} (\sigma, \text{skip})} \\
\\
\text{FUNCTIONCALLSTEP} \\
\frac{E \vdash \kappa \xrightarrow{\alpha} \kappa'}{E \vdash (\sigma, \mathbf{x} := \kappa) \xrightarrow{\alpha} (\sigma, \mathbf{x} := \kappa')}
\end{array}$$

**Lemma 25** (Determinism of Thread Semantics). For any thread  $(\sigma, \mathbf{s})$ , if  $E \vdash (\sigma, \mathbf{s}) \xrightarrow{\alpha} (\sigma_1, \mathbf{s}_1)$  and  $E \vdash (\sigma, \mathbf{s}) \xrightarrow{\alpha'} (\sigma_2, \mathbf{s}_2)$  then  $\mathbf{s}_1 = \mathbf{s}_2$ .

**Definition 26** ( $- \vdash - \xrightarrow{-}$ , Threadpool Semantics). We define the operational semantics of a thread pool as a atomic action labelled transition relation.

$$- \vdash - \xrightarrow{-} \subseteq Env \times ThreadPool \times Action \times ThreadPool$$

$$\frac{E \vdash t \xrightarrow{\text{fork}(\mathbf{f}, \vec{v})} t' \quad E(\mathbf{f}) = (\vec{x}, \mathbf{s}) \quad \sigma(\vec{x}) = \vec{v}}{E \vdash T \parallel t \xrightarrow{\text{id}} T \parallel t' \parallel (\sigma, \mathbf{s})} \quad \frac{E \vdash t \xrightarrow{\alpha} t' \quad \alpha \in Action}{E \vdash T \parallel t \xrightarrow{\alpha} T \parallel t'}$$

**Definition 27** ( $- \vdash - \xrightarrow{-}$ , Program Semantics). We define the operational semantics of programs as a single step transition relation

$$- \vdash - \rightarrow - \subseteq Env \times Program \times Program$$

$$\frac{E \vdash T \xrightarrow{\alpha} T' \quad h' \in \llbracket \alpha \rrbracket (h)}{E \vdash (h, T) \rightarrow (h', T')}$$

## 5 Assertion Logic

**Definition 28** (Type, Logical Types). We define a set of primitive types over which we will allow quantification:

$$\text{Type } \tau ::= \text{Val} \mid \text{Perm} \mid \text{Region}$$

**Definition 29** (*LVar*, Logical Variables). We suppose a set *LVar* ranged over by  $x, y$  etc.

**Definition 30** (Context, Logical Contexts). Well-formed logical contexts binds logical variables to primitive types:

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

where  $x$  does not occur in the  $\Gamma$  it extends. A context  $\Gamma$  induces a partial map from variables to types, and we denote a defined look-up by  $\Gamma(x) = \tau$ .

**Definition 31** (*RegionTypeName*, Region Type Names). We suppose a set of region type names ranged over by  $\mathbb{T}$ .

**Definition 32** (*RegionType*, Region Types). We define the set of region types as parameterized region type names, according to the following definition:

$$\text{RegionType} := \{ \mathbb{T}(r, \vec{x} : \vec{\tau}) \mid r \in LVar, \vec{x} \in LVar^*, \vec{\tau} \in \text{Type}^* \}$$

**Definition 33** (*PrimitiveGuard*, Primitive Guard Symbols). We suppose an infinite set of primitive guard symbols, usually ranged over by  $G$ .

**Definition 34** (Term, Assertion Logic Terms). We define the syntax of assertions with the following grammar. The types  $\tau$  described in the grammar are simple types over which we allow quantification; this is not a higher-order logic.

$M, N, O \in \text{Term} ::=$	$x$	logical variables
	$  \top   \perp   M \wedge N   M \vee N$	propositional formulae
	$  M \Rightarrow N   \neg N$	
	$  M ? N : O$	conditionals
	$  \forall x : \tau. M   \exists x : \tau. M$	quantification
	$  M =_{\tau} N$	primitive equality
	$  M * N   \text{emp}$	separating conjunction
	$  M \mapsto N   M \mapsto [N]$	points-to assertions
	$  M@(N)   \mathbb{T}(M, N, O)$	region assertions
	$  G   G[M]   G(M)$	guard resources
	$  0p   1p   \sim M   M \cdot N$	permission expressions
	$  \text{compatible}(M, N)$	
	$  x   n   M + N$	program expressions
	$  M - N   M * N   M / N$	
	$  M \leq N   M < N$	
	$  M \geq N   M > N$	
	$  \epsilon   M, N$	list expressions

**Definition 35** ( $- \vdash - : \text{Val}$ , Program Value Expressions). Program value expressions precisely reflect arithmetic expressions and comparisons as per the object language. We do not distinguish between expressions in the assertion logic and expressions in programs; the only difference is that assertions additionally allow for logical variables.

Well-typed program value expressions are given by the following rules:

$$\begin{array}{c}
 - \vdash - : \text{Val} \subseteq \text{Context} \times \text{Term} \\
 \\
 \frac{\Gamma(x) = \text{Val}}{\Gamma \vdash x : \text{Val}} \qquad \frac{}{\Gamma \vdash x : \text{Val}} \qquad \frac{}{\Gamma \vdash n : \text{Val}} \\
 \\
 \frac{\Gamma \vdash M : \text{Val} \quad \Gamma \vdash N : \text{Val} \quad \text{op} \in \{+, -, *, /\}}{\Gamma \vdash M \text{ op } N : \text{Val}} \\
 \\
 \frac{\Gamma \vdash M : \text{Val} \quad \Gamma \vdash N : \text{Val} \quad \text{comp} \in \{<, \leq, \geq, >\}}{\Gamma \vdash M \text{ comp } N : \text{Val}}
 \end{array}$$

**Definition 36** ( $- \vdash - : \text{Perm}$ , Permission Assertions). Well-typed permission expression are given by the following rules:

$$\begin{array}{c}
 - \vdash - : \text{Perm} \subseteq \text{Context} \times \text{Term} \\
 \\
 \frac{\Gamma(x) = \text{Perm}}{\Gamma \vdash x : \text{Perm}} \qquad \frac{}{\Gamma \vdash 0p : \text{Perm}} \qquad \frac{}{\Gamma \vdash 1p : \text{Perm}} \qquad \frac{\Gamma \vdash M : \text{Perm}}{\Gamma \vdash \sim M : \text{Perm}} \\
 \\
 \frac{\Gamma \vdash M : \text{Perm} \quad \Gamma \vdash N : \text{Perm}}{\Gamma \vdash M \cdot N : \text{Perm}}
 \end{array}$$

**Definition 37** ( $- \vdash - : \text{Pure}$ , Pure Assertions). Well-typed pure assertions are given by the following rules. They are “pure” in that they do not contain spatial or region assertions.

$$- \vdash - : \text{Pure} \subseteq \text{Context} \times \text{Term}$$

$$\begin{array}{c}
\frac{}{\Gamma \vdash \perp : \text{Pure}} \qquad \frac{}{\Gamma \vdash \top : \text{Pure}} \qquad \frac{\Gamma \vdash M : \text{Pure} \quad \Gamma \vdash N : \text{Pure}}{\Gamma \vdash M \wedge N : \text{Pure}} \\
\frac{\Gamma \vdash M : \text{Pure} \quad \Gamma \vdash N : \text{Pure}}{\Gamma \vdash M \vee N : \text{Pure}} \qquad \frac{\Gamma \vdash M : \text{Pure} \quad \Gamma \vdash N : \text{Pure}}{\Gamma \vdash M \Rightarrow N : \text{Pure}} \qquad \frac{\Gamma \vdash M : \text{Pure}}{\Gamma \vdash \neg M : \text{Pure}} \\
\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \tau}{\Gamma \vdash M =_{\tau} N : \text{Pure}} \qquad \frac{\Gamma, x : \tau \vdash M : \text{Pure}}{\Gamma \vdash \forall x : \tau. M : \text{Pure}} \qquad \frac{\Gamma, x : \tau \vdash M : \text{Pure}}{\Gamma \vdash \exists x : \tau. M : \text{Pure}} \\
\frac{\Gamma \vdash M : \text{Perm} \quad \Gamma \vdash N : \text{Perm}}{\Gamma \vdash \text{compatible}(M, N) : \text{Pure}}
\end{array}$$

**Definition 38** ( $- \vdash -$  : Guard, Guard Expressions). Well-typed guard expressions are given by the following rules, wherein  $G$  ranges over the set of primitive guards:

$$\begin{array}{c}
- \vdash - : \text{Guard} \subseteq \text{Context} \times \text{Term} \\
\frac{}{\Gamma \vdash G : \text{Guard}} \qquad \frac{\Gamma \vdash M : \text{Perm}}{\Gamma \vdash G[M] : \text{Guard}} \qquad \frac{\Gamma \vdash M : \text{Val}}{\Gamma \vdash G(M) : \text{Guard}} \qquad \frac{\Gamma \vdash M : \text{Guard} \quad \Gamma \vdash N : \text{Guard}}{\Gamma \vdash M * N : \text{Guard}}
\end{array}$$

**Definition 39** ( $- \vdash -$  : Region, Region Identifiers).

$$\begin{array}{c}
- \vdash - : \text{Region} \subseteq \text{Context} \times \text{Term} \\
\frac{\Gamma(x) = \text{Region}}{\Gamma \vdash x : \text{Region}}
\end{array}$$

**Definition 40** ( $- ; - \vdash -$  : Assn, Assertions). Well-typed assertions are given by the following rules.

$$\begin{array}{c}
- ; - \vdash - : \text{Assn} \subseteq \mathcal{P}(\text{RegionType}) \times \text{Context} \times \text{Term} \\
\frac{\Gamma \vdash M : \text{Pure}}{R; \Gamma \vdash M : \text{Assn}} \qquad \frac{\Gamma \vdash M : \text{Pure} \quad R; \Gamma \vdash N : \text{Assn} \quad \text{dom}(R); \Gamma \vdash O : \text{Assn}}{R; \Gamma \vdash M ? N : O : \text{Assn}} \\
\frac{R; \Gamma \vdash M : \text{Assn} \quad R; \Gamma \vdash N : \text{Assn}}{R; \Gamma \vdash M * N : \text{Assn}} \qquad \frac{\Gamma \vdash M : \text{Val} \quad \Gamma \vdash N : \text{Val}}{R; \Gamma \vdash M \mapsto N : \text{Assn}} \\
\frac{\Gamma \vdash M : \text{Val} \quad \Gamma \vdash N : \text{Val}}{R; \Gamma \vdash M \mapsto [N] : \text{Assn}} \qquad \frac{}{R; \Gamma \vdash \text{emp} : \text{Assn}} \qquad \frac{\Gamma \vdash x : \text{Region} \quad \Gamma \vdash M : \text{Guard}}{R; \Gamma \vdash x@(M) : \text{Assn}} \\
\frac{\mathbb{T}(r, \vec{x} : \vec{\tau}) \in R \quad \Gamma \vdash x : \text{Region} \quad |\vec{M}| = |\vec{\tau}| \quad \Gamma \vdash M_i : \tau_i \quad \Gamma \vdash N : \text{Val}}{R; \Gamma \vdash \mathbb{T}(x, \vec{M}, N) : \text{Assn}}
\end{array}$$

Assertions also contain the usual first order logic connectives:

$$\begin{array}{c}
\frac{R; \Gamma \vdash M : \text{Assn} \quad R; \Gamma \vdash N : \text{Assn} \quad \text{op} \in \{\vee, \wedge, \Rightarrow\}}{R; \Gamma \vdash M \text{ op } N : \text{Assn}} \\
\frac{R; \Gamma, x : \tau \vdash M : \text{Assn} \quad \text{quan} \in \{\forall, \exists\}}{R; \Gamma \vdash \text{quan } x. M : \text{Assn}} \qquad \frac{c \in \{\top, \perp\}}{R; \Gamma \vdash c : \text{Assn}} \qquad \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \tau}{R; \Gamma \vdash M =_{\tau} N : \text{Assn}} \\
\frac{R; \Gamma \vdash M : \text{Assn}}{R; \Gamma \vdash \neg M : \text{Assn}}
\end{array}$$

## 5.1 Entailment Logic

**Definition 41** (Syntactic Entailment of Assertions). We define syntactic entailment on those as a relation:

$$-; - \mid - \vdash - \subseteq \mathcal{P}(\text{RegionType}) \times \text{Context} \times \mathcal{P}(\text{Assn}) \times \text{Assn}$$

An entailment  $R; \Gamma \mid P_1, \dots, P_n \vdash Q$  is well-formed when

$$R; \Gamma \vdash P_i : \text{Assn} \qquad R; \Gamma \vdash Q : \text{Assn}$$

The rules of the entailment logic are a standard first-order intuitionistic separation logic.

## 6 Specification Logic

### 6.1 Specification Syntax

**Definition 42** (*GAD*, Guard Algebra Declaration). A guard algebra declaration is an expression from the following grammar of guard algebra combinators, where  $G$  ranges over primitive guard symbols:

<i>GAD</i> $g$	$::=$	$G$	Indivisible Guard
		$\%G$	Divisible Guard
		$\#G$	Parameterized Guard
		$g * g$	Product Construction
		$g + g$	Sum Construction
		$0$	Nil Guard

A guard algebra declaration is well-formed when a primitive guard symbol  $G$  occurs at most once in the declaration.

**Definition 43** (*ID*, Region Interpretation Declaration). Region interpretations are a collection of clauses of the form

$$(\Delta). \Pi \mid e : P$$

where  $\Delta$  is a logical context and  $\Pi, e$  and  $P$  are terms of the assertion logic.

A single clause is well-formed with respect to a set of region types  $R$ , variable  $r$  and variables  $\vec{x}$  of types  $\vec{\tau}$  if

- $r : \text{Region}, \vec{x} : \vec{\tau}, \Delta \vdash \Pi : \text{Pure}$
- $r : \text{Region}, \vec{x} : \vec{\tau}, \Delta \vdash e : \text{Val}$
- $R; r : \text{Region}, \vec{x} : \vec{\tau}, \Delta \vdash P : \text{Assn}$

A whole region interpretation declaration is well-formed with respect to a region declaration context  $R$ , identifier  $r$  and variables  $\vec{x}$  of types  $\vec{\tau}$  when, in addition to each clause being well-formed,

**Definition 44** (*AD*, Region Action Declaration). A region action declaration is a collection of clauses of the form

$$(\Delta). \Pi \mid G : e_1 \rightsquigarrow e_2$$

where  $\Delta$  is a logical context,  $P, G, e_1$  and  $e_2$  are terms in the assertion logic.

A single clause is well-formed with respect to a variable  $r$  and variables  $\vec{x}$  of types  $\vec{\tau}$  when

- $r : \text{Region}, \vec{x} : \vec{\tau}, \Delta \vdash \Pi : \text{Pure}$
- $r : \text{Region}, \vec{x} : \vec{\tau}, \Delta \vdash G : \text{Guard}$
- $r : \text{Region}, \vec{x} : \vec{\tau}, \Delta \vdash e_1 : \text{Val}$
- $r : \text{Region}, \vec{x} : \vec{\tau}, \Delta \vdash e_2 : \text{Val}$

**Definition 45** (*RegionDecl*, Region Declarations). A region declaration is a triple of a guard algebra declaration, a region interpretation declaration and a region action declaration:

$$\text{RegionDecl} := \{(g, i, a) \mid g \in \text{GAD}, i \in \text{ID}, a \in \text{AD}\}$$

A region declaration  $(g, i, a)$  is well-formed with respect to a collection of region types  $R$ , logical variable  $r$  and logical variables  $\vec{x}$  of types  $\vec{\tau}$  when:

- $g$  is a well-formed guard algebra declaration
- $i$  is a well-formed region interpretation declaration with respect to  $R$ ,  $r$  and  $\vec{x} : \vec{\tau}$
- $a$  is a well-formed action declaration with respect to  $r$  and  $\vec{x} : \vec{\tau}$

**Definition 46** (*RegionContext*, Region Contexts). A region declaration contexts is defined by the following grammar

$$R ::= \cdot \mid R, \mathbb{T}(r, \vec{x} : \vec{\tau})(g, i, a)$$

We denote the set of region types associated in the region context  $R$  as  $\text{dom}(R)$ .

A region context  $R$  is well-formed when each individual region declaration  $(g, i, a)$  is well-formed with respect to the associated region type parameters  $r$ ,  $\vec{x}$  and  $\vec{\tau}$ , and the collection of all the region types  $\text{dom}(R)$ , and each region type is associated at most once.

These conditions are enough for a well-formed region context to induce a partial mapping from region types to well-formed region declarations, and we denote a well-defined lookup as follows:

$$R(\mathbb{T}(r, \vec{x} : \vec{\tau})) = (g, i, a)$$

**Definition 47** (*I(-)*, Interpretation Function). A well-formed region context induces a function on well-typed region assertions to spatial assertions, intuitively computing the “symbolic” interpretation of a region assertion. The region context in question will be clear from context.

For a region context  $R$  and a well-formed region assertion

$$\text{dom}(R); \Gamma \vdash \mathbb{T}(x, \vec{M}, N) : \text{Assn}$$

we define the interpretation as follows: Let  $\mathbb{T}(r, \vec{x} : \vec{\tau})$  be the associated region type in  $R$ . Then, for each clause in the region interpretation, say

$$(\Delta). \Pi \mid e : P$$

We can build the assertion

$$\text{dom}(R); \Gamma \vdash (\exists \Delta. \Pi \wedge N = e \wedge P)[x, \vec{M}/r, \vec{x}] : \text{Assn}$$

where we close the assertions in the clause by the concrete parameters at hand, and the local context is closed existentially. Then, the interpretation is the disjunction of all such clauses:

$$I(\mathbb{T}(x, \vec{M}, N)) = \bigvee_{(\Delta). \Pi \mid e : P} (\exists \Delta. \Pi \wedge N = e \wedge P)[x, \vec{M}/r, \vec{x}]$$

**Definition 48** (*SpecCtxt*, Function Specification Context). Function specification contexts assign specifications to function symbols. They are generated by the following grammar, where  $\Gamma$  is a logical context, and  $\vec{y}$  is a sequence of logical variables:

$$\Phi \in \text{SpecCtxt} ::= \cdot \mid \Phi, f : (\Gamma, \vec{y}) \{P\} \{r.Q\}$$

A function specification context is well-formed with regards to a set of region types  $R$  when

$$R; \Gamma, \vec{y} : \text{Val} \vdash P : \text{Assn} \qquad R; \Gamma, \vec{y} : \text{Val}, r : \text{Val} \vdash Q : \text{Assn}$$

where  $P$  and  $Q$  do not mention any program variables. Furthermore we require that  $f$  occurs at most once in  $\Phi$ .

These conditions are enough for a well-formed function specification context to induce a partial mapping from function names to function specifications, and we denote a successful lookup with

$$\Phi(f) = (\Gamma, \vec{y}) \{P\} \{r.Q\}$$

**Definition 49** (Spec, Statement Specifications). The set of syntactic statement specifications is defined by the following grammar, where  $P, Q, U \in \text{Term}$ ,  $r \in \text{LVar}$  and  $s \in \text{Stmt}$ .

$$s \in \text{Spec} ::= \{P\} s \{Q \mid r.U\}$$

A statement specification is well-typed wrt. a region context  $R$ , function specification context  $\Phi$  (well-typed wrt.  $\text{dom}(R)$ ) and logical context  $\Gamma$  according to the following judgement:

$$\frac{\begin{array}{l} \text{dom}(R); \Gamma \vdash P : \text{Assn} \\ \text{dom}(R); \Gamma \vdash Q : \text{Assn} \\ \text{dom}(R); \Gamma, r : \text{Val} \vdash U : \text{Assn} \end{array}}{R; \Phi; \Gamma \vdash \{P\} s \{Q \mid r.U\} : \text{Spec}}$$

**Definition 50** (– atomic, Atomic Statements). We declare a subset of  $\text{Stmt}$  as atomic:

$$\frac{}{x := \text{CAS}(e_1, e_2, e_3) \text{ atomic}} \quad \frac{}{[e_1] := e_2 \text{ atomic}} \quad \frac{}{x := [e] \text{ atomic}}$$

**Definition 51** (Atomic Statement Specifications). The set of syntactic atomic statement specifications is given by the following grammar where  $P, Q \in \text{Term}$ ,  $s \in \text{Stmt}$  with  $s$  atomic:

$$\text{Atomic} ::= \langle P \rangle s \langle Q \rangle$$

An atomic statement specification is well-typed wrt. to collection of region identifiers  $S$ , a region declaration  $R$  and a logical context  $\Gamma$  according to the following judgement:

$$\frac{R; \Gamma \vdash P : \text{Assn} \quad R; \Gamma \vdash Q : \text{Assn} \quad \forall x \in S. \Gamma \vdash x : \text{Region}}{R; \Gamma \vdash_S \langle P \rangle s \langle Q \rangle : \text{Atomic}}$$

**Definition 52** (Region Opening). We define a judgment for expressing the opening of regions, given by the following grammar where  $I$  is some finite index set  $P, Q_i$  are assertions,  $\Delta_i$  are contexts and  $\vec{r}_i$  are sequences of region identifiers:

$$[P] \text{ open } [\{(\Delta_i). (Q_i, \vec{r}_i)_{i \in I}\}]$$

An opening judgment is well-formed with regards to a set of region-types  $R$  and a logical context  $\Gamma$  according to the following judgement:

$$\frac{\begin{array}{l} R; \Gamma \vdash P : \text{Assn} \\ \forall i \in I. R; \Gamma, \Delta_i, \vec{r}_i : \text{Region} \vdash Q_i : \text{Assn} \end{array}}{R; \Gamma \vdash [P] \text{ open } [\{(\Delta_i). (Q_i, \vec{r}_i)_{i \in I}\}]}$$

**Definition 53** (Region Closing). We define a judgment for closing regions, given by the following grammar where  $P, Q$  are assertions and  $\vec{r}$  a sequence of region identifiers:

$$[P] \text{ close}(\vec{r}) [Q]$$

A closing judgement is well-formed with regards to a set of region types  $R$  and a logical context  $\Gamma$  according to the following judgement:

$$\frac{R; \Gamma \vdash P : \text{Assn} \quad R; \Gamma \vdash Q : \text{Assn}}{R; \Gamma \vdash [P] \text{ close}(\vec{r}) [Q]}$$

**Convention 54.** For two sequences of assertion logic expressions of equal length  $n$ ,  $\vec{x}$  and  $\vec{y}$ , we use the notation  $\vec{x} = \vec{y}$  as notation for the assertion  $x_1 = y_1 * \dots * x_n = y_n$ .

**Definition 55** (FunctionSpec, Function Specifications). The set of syntactic function specifications is defined by the following grammar, letting  $f \in \text{Fun}$ ,  $\vec{x} \in \text{Var}^*$ ,  $P, Q \in \text{Term}$  are assertions that does not mention program variables and  $s \in \text{Stmt}$ ,  $\Gamma \in \text{Context}$  and  $\vec{y} \in \text{LVar}^*$ .

$$\text{FunctionSpec} ::= f(\vec{x})(\Gamma, \vec{y}) \{P\} s \{r.Q\}$$

A function specification is well-typed wrt. a region context  $R$  and a function specification context  $\Phi$  if its constituents form a well-typed statement specification.

$$\frac{R; \Phi; \Gamma, \vec{y} : \text{Val} \vdash \{P * (\vec{x} = \vec{y})\} \text{ s } \{\forall r. Q \mid r. Q\} : \text{Spec} \quad |\vec{x}| = |\vec{y}|}{R; \Phi \vdash \mathbf{f}(\vec{x})(\Gamma, \vec{y})\{P\} \text{ s } \{r. Q\} : \text{FunctionSpec}}$$

**Definition 56** ( $\lceil - \rceil$ , Function Specification Erasure (Specification)). We define

$$\lceil - \rceil : \text{FunctionSpec}^* \rightarrow \text{SpecCtxt}$$

as follows:

$$\begin{aligned} \lceil \epsilon \rceil &= \cdot \\ \lceil \vec{f}, \mathbf{f}_i(\vec{x})(\Gamma, \vec{y})\{P\} \text{ s } \{r. Q\} \rceil &= \lceil \vec{f} \rceil, \mathbf{f}_i : (\Gamma, \vec{y})\{P\} \{r. Q\} \end{aligned}$$

**Definition 57** ( $\text{ProgramSpec}$ , Program Specifications). The set of syntactic program specifications is defined as sequences of region declarations and function specifications:

$$\text{ProgramSpec} := \text{RegionDecl}^*; \text{FunctionSpec}^*$$

A program specification is well-typed when each individual function specification is well-typed wrt. to the region and function specification contexts described by the region and function declarations, respectively:

$$\frac{\vec{r}; \lceil \vec{f} \rceil \vdash \mathbf{f}_i : \text{FunctionSpec}, \quad \forall \mathbf{f}_i \in \vec{f}}{\vdash \vec{r}; \vec{f} : \text{ProgramSpec}}$$

## 6.2 Specification Rules

### 6.2.1 Program Specification

We say that we can derive a program specification when we can derive each of the individual function specifications with respect to the region and function contexts specified by the program specification.

$$\frac{\vec{r}; \lceil \vec{f} \rceil \vdash \mathbf{f}_i, \quad \forall \mathbf{f}_i \in \vec{f}}{\vdash \vec{r}; \vec{f}}$$

### 6.2.2 Function Specification

We say that we can derive a function specification when we can derive its implementation with respect to the pre- and post-condition imposed by the function specification.

$$\frac{R; \Phi; \Gamma, \vec{y} : \text{Val} \vdash \{P * (\vec{x} = \vec{y})\} \text{ s } \{\forall r. Q \mid r. Q\}}{R; \Phi \vdash \mathbf{f}(\vec{x})(\Gamma, \vec{y})\{P\} \text{ s } \{r. Q\}}$$

### 6.2.3 Statement Specification

**Definition 58** ( $\text{mod}(-)$ , Modifies Sets). We define the set of local variables modified by a given statement as follows:

$$\begin{aligned}
& \text{mod} : \text{Stmt} \rightarrow \mathcal{P}(\text{Var}) \\
& \text{mod}(s_1; s_2) = \text{mod}(s_1) \cup \text{mod}(s_2) \\
& \text{mod}(\text{if}(e) \text{ then } \{s_1\} \text{ else } \{s_2\}) = \text{mod}(s_1) \cup \text{mod}(s_2) \\
& \text{mod}(\text{while}(e)\{s\}) = \text{mod}(s) \\
& \text{mod}(x := \text{alloc}(e)) = \{x\} \\
& \text{mod}(x := \text{CAS}(e_1, e_2, e_3)) = \{x\} \\
& \text{mod}(x := e) = \{x\} \\
& \text{mod}([e_1] := e_2) = \emptyset \\
& \text{mod}(x := [e]) = \{x\} \\
& \text{mod}(x := f(\vec{e})) = \{x\} \\
& \text{mod}(\text{return } e) = \emptyset \\
& \text{mod}(\text{fork } f(\vec{e})) = \emptyset \\
& \text{mod}(\text{skip}) = \emptyset
\end{aligned}$$

$$\begin{array}{c}
\frac{\Phi(f) = (\Gamma, \vec{y})\{P\}\{r.Q\}}{R; \Phi; \Gamma, \vec{y} : \text{Val} \vdash \{P * (\vec{e} = \vec{y})\} x := f(\vec{e}) \{\exists r.x = r * Q \mid r.U\}} \\
\\
\frac{}{R; \Phi; \Gamma \vdash \{\top\} \text{return } e \{Q \mid r.e = r\}} \\
\\
\frac{R; \Phi; \Gamma \vdash \{P * e \neq 0\} s_1 \{Q \mid r.U\} \quad R; \Phi; \Gamma \vdash \{P * e = 0\} s_2 \{Q \mid r.U\}}{R; \Phi; \Gamma \vdash \{P\} \text{if}(e) \text{ then } \{s_1\} \text{ else } \{s_2\} \{Q \mid r.U\}} \\
\\
\frac{R; \Phi; \Gamma \vdash \{I * e \neq 0\} s \{I \mid r.U\}}{R; \Phi; \Gamma \vdash \{I\} \text{while}(e)\{s\} \{I * e = 0 \mid r.U\}} \\
\\
\frac{}{R; \Phi; \Gamma \vdash \{e = v * v > 0\} x := \text{alloc}(e) \{\exists n.x = n * n \mapsto [v] \mid r.U\}} \\
\\
\frac{}{R; \Phi; \Gamma \vdash \{P\} \text{skip} \{P \mid r.U\}} \quad \frac{R; \Phi; \Gamma \vdash \{P\} s_1 \{S \mid r.U\} \quad R; \Phi; \Gamma \vdash \{S\} s_2 \{Q \mid r.U\}}{R; \Phi; \Gamma \vdash \{P\} s_1; s_2 \{Q \mid r.U\}} \\
\\
\frac{}{R; \Phi; \Gamma \vdash \{x = e_0\} x := e \{x = e[e_0/x] \mid r.U\}} \quad \frac{\Phi(f) = (\Gamma, \vec{y})\{P\}\{r.Q\}}{R; \Phi; \Gamma \vdash \{P * (\vec{e} = \vec{y})\} \text{fork } f(\vec{e}) \{\top \mid r.U\}} \\
\\
\frac{R; \Phi; \Gamma \vdash \{P\} s \{Q \mid r.U\} \quad \text{mod}(s) \cap F = \emptyset}{R; \Phi; \Gamma \vdash \{P * F\} s \{Q * F \mid r.U\}} \\
\\
\frac{R; \Gamma \mid P \vdash P' \quad R; \Phi; \Gamma \vdash \{P'\} s \{Q' \mid r.U'\} \quad R; \Gamma \mid Q' \vdash Q \quad R; \Gamma, r : \text{Val} \mid U' \vdash U}{R; \Phi; \Gamma \vdash \{P\} s \{Q \mid r.U\}} \\
\\
\frac{R; \Gamma \vdash [P] \text{open} [\{(\Delta_i). (P'_i, \vec{r}_i)\}_{i \in I}] \quad \forall i \in I. \quad R; \Gamma, \Delta_i \vdash_{\{\vec{r}_i\}} \langle P'_i \rangle s \langle Q_i * \text{newRegion}(\vec{n}_i) \rangle \quad R; \Gamma, \Delta_i \vdash [Q_i * \text{newRegion}(\vec{n}_i)] \text{close}(\vec{r}_i, \vec{n}_i) [Q]}{R; \Phi; \Gamma \vdash \{P\} s \{\text{stabilize}(Q) \mid r.U\}}
\end{array}$$

## 6.2.4 Atomic Statement Specifications

$$\frac{}{R; \Gamma \vdash_S \langle e_1 \mapsto \_ \rangle [e_1] := e_2 \langle e_1 \mapsto e_2 \rangle} \quad \frac{}{R; \Gamma \vdash_S \langle e = n * n \mapsto v \rangle x := [e] \langle x = v * n \mapsto v \rangle}$$

$$\frac{R; \Gamma \vdash_S \langle e_1 = a * a \mapsto v * e_2 = old * e_3 = new \rangle}{x := CAS(e_1, e_2, e_3)} \\ \langle (x \neq 0 * v = old * a \mapsto new) \vee (x = 0 * v \neq old * a \mapsto v) \rangle$$

## 7 Model

We here describe the constructions in Sets that we use to interpret the specification logic.

**Definition 59** (*Perm*, Permissions). We define the set of *Perm* as the free atomless boolean algebra over an infinite set.

**Definition 60** (*RegionId*, Region Identifiers). We assume a set of region identifiers, *RegionId*.

**Definition 61** (*LVal*, Logical Values). We define the set of logical values to be the disjoint union of program values, permissions and region identifiers:

$$LVal := Val + Perm + RegionId$$

recalling the definition of program values from Definition 11.

**Definition 62** (*AbstractState*, Abstract States). We define the set of abstract region states as the set of program values:

$$AbstractState := Val$$

**Definition 63** (PCMZ, Partial Commutative Monoid with Zero). A PCMZ  $M$  is a 4-tuple  $(|M|, \epsilon, 0, \cdot)$  consisting of an underlying set  $|M|$  with a distinguished element  $\epsilon$  (the unit), a distinguished element  $0$  (the zero) and a partial composition on that set (multiplication or simply 'composition', noted  $\cdot$ ) satisfying the following requirements for all  $a, b, c \in |M|$ , where  $x \downarrow y$  means ' $x \cdot y$  is defined'.

- $a \downarrow b$  implies  $(b \downarrow a$  and  $a \cdot b = b \cdot a)$ .
- $\epsilon \downarrow a$  and  $\epsilon \cdot a = a$ .
- $0 \downarrow a$  and  $0 \cdot a = 0$ .
- $(b \downarrow c$  and  $a \downarrow (b \cdot c))$  implies  $(a \downarrow b$  and  $(a \cdot b) \downarrow c$  and  $a \cdot (b \cdot c) = (a \cdot b) \cdot c)$ .

We note that the *free* PCMZ over a set  $X$  is the usual, initial construction of a structure over an underlying set. Note that by initiality, any function  $f$  from  $X$  into the underlying set of another PCMZ, say  $|M|$ , gives us a canonical PCMZ homomorphism from the free PCMZ over  $X$  to  $M$ , denoted  $\bar{f}$ .

**Definition 64** (*Guard*, The PCMZ of Primitive Guards). We define *Guard* as the free PCMZ over a set of expressions over *PrimitiveGuard*, call it  $|Guard|$ , built according to the following grammar wherein  $G$  ranges over *PrimitiveGuard*,  $\pi$  ranges over *Perm* and  $x$  ranges over *LVar*:

$$\begin{aligned} |Guard| ::= & G \\ & | G[\pi] \\ & | G(x) \end{aligned}$$

**Definition 65** (*GuardAlgebra*, Guard Algebras).

$$\sum_{G:PCMZ} |Guard| \rightarrow |G|$$

**Definition 66** (*GuardAlgebraAssignment*, Guard Algebra Assignment). We define a guard algebra assignment as a partial finite map from region types to guard algebras,

$$\text{GuardAlgebraAssignment} := \text{RegionType} \xrightarrow{\text{fin}} \text{GuardAlgebra}$$

**Definition 67** (*RegionAssignment*, Region Assignments). We define region assignments as partial finite maps from region identifiers to triples describing their (indexed) region type, state, and which guards we have in possession.

$$\text{RegionAssignment} := \text{RegionId} \xrightarrow{\text{fin}} ((\text{RegionType} \times \text{LVal}^*) \times \text{AbstractState} \times |\text{Guard}|)$$

We refer to the first, second and third component of the codomain as  $\alpha(r).\text{type}$ ,  $\alpha(r).\text{state}$  and  $\alpha(r).\text{guards}$  for  $\alpha \in \text{RegionAssignment}$ ,  $r \in \text{dom}(\alpha)$ .

Region assignments form a partial commutative monoid where composition is defined as follows - we use  $\perp$  to denote undefined values, i.e.  $f(a) = \perp$  if  $a \notin \text{dom}(f)$ .

$$(a_1 \cdot a_2)(r) = \begin{cases} a_1(r) & a_2(r) = \perp \\ a_2(r) & a_1(r) = \perp \\ (a_1(r).\text{type}, a_1(r).\text{state}, a_1(r).\text{guards}) \cdot a_2(r).\text{guards}) & a_1(r).\text{type} = a_2(r).\text{type} \\ & \wedge a_1(r).\text{state} = a_2(r).\text{state} \\ \perp & \text{otherwise} \end{cases}$$

Observe that the empty map is a unit. Associativity and commutativity is inherited by the properties of equality and the PCM *Guard*.

This gives rise to canonical partial order known as the extension order, here written in full: we say  $a_1 \sqsubseteq a_2$  iff  $\text{dom}(a_1) \subseteq \text{dom}(a_2)$  and for all  $r \in \text{dom}(a_1)$ :

1.  $a_1(r).\text{type} = a_2(r).\text{type}$
2.  $a_1(r).\text{state} = a_2(r).\text{state}$
3.  $a_1(r).\text{guards} \sqsubseteq a_2(r).\text{guards}$

**Definition 68** (*Abstract Configuration*). Abstract machine configurations consists of a heap and region assignment pair:

$$\text{Heap} \times \text{RegionAssignment}$$

Abstract configurations form a partial commutative monoid by pointwise lifting of both unit and composition. This also gives rise to the pointwise extension order:

$$(h_1, a_1) \leq (h_2, a_2) \stackrel{\text{def}}{\iff} \exists (h_3, a_3). h_1 \cdot h_3 = h_2 \wedge a_1 \cdot a_3 = a_2$$

**Definition 69** (*Assertion*, Assertions). We define assertions as upwards closed subsets of abstract configurations, ordered according to the canonical extension order.

$$\text{Assertion} := \mathcal{P} \uparrow (\text{Heap} \times \text{RegionAssignment})$$

**Definition 70** (*RegionInterpretation*, Region Interpretations).

$$\text{RegionInterpretation} := \text{AbstractState} \multimap \text{Assertion}$$

**Definition 71** (*LTS*, Labeled Transition Systems).

$$\text{LTS} := |\text{Guard}| \xrightarrow{\text{mon}} \mathcal{P}(\text{AbstractState} \times \text{AbstractState})$$

**Definition 72** (*RegionTypeAssignment*, Region Type Assignments). A region type assignment associates a region interpretation, labeled transition system and guard algebra with each (instantiated) region type.

$$\text{RegionTypeAssignment} := (\text{RegionType} \times \text{LVal}^*) \rightarrow \text{RegionInterpretation} \times \text{LTS} \times \text{GuardAlgebra}$$

We refer to the three components of the type assignment  $\rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v})$  as follows:

$$\rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}).\text{int} \quad \rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}).\text{Its} \quad \rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}).\text{GA}$$

The choice of guard algebra must be independent of the instantiated values, i.e. for a given  $\rho \in \text{RegionTypeAssignment}$ , for any region type  $\mathbb{T}(r, \vec{x} : \vec{\tau})$  and two choices of values  $\vec{v}$  and  $\vec{u}$  it must hold that

$$\rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}) = \rho(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{u})$$

**Definition 73** ( $\text{GA}(-)$ , RTA Algebra Assignment Erasure). We denote the erasure of region interpretation and labeled transition system from a region type assignment  $\rho$  as  $\text{GA}(\rho)$ , erasing a

$$(\text{RegionType} \times \text{LVal}^*) \xrightarrow{\text{fin}} \text{RegionInterpretation} \times \text{LTS} \times \text{GuardAlgebra}$$

to a

$$\text{RegionType} \xrightarrow{\text{fin}} \text{GuardAlgebra}$$

## 8 Interpretation

**Definition 74** (Interpretation of Primitive Types). We interpret the primitive types into the corresponding sets:

$$\begin{aligned} \llbracket \text{Val} \rrbracket &= \text{Val} \\ \llbracket \text{Perm} \rrbracket &= \text{Perm} \\ \llbracket \text{Region} \rrbracket &= \text{RegionId} \end{aligned}$$

**Definition 75** (Interpretation of Logical Variable Contexts). We interpret program variable contexts as substitutions, valuations or partial maps to values of the appropriate type:

$$\begin{aligned} \llbracket \cdot \rrbracket &= \emptyset \\ \llbracket \Gamma, x : \tau \rrbracket &= \{ \gamma[x \mapsto v] \mid \gamma \in \llbracket \Gamma \rrbracket, v \in \llbracket \tau \rrbracket \} \end{aligned}$$

**Definition 76** (Interpretation of Program Expressions). We interpret program expressions as functions from interpretations of their contexts into *Val*.

$$\llbracket \Gamma \vdash M : \text{Val} \rrbracket : \llbracket \Gamma \rrbracket \times \text{Stack} \rightarrow \text{Val}$$

$$\begin{aligned} \llbracket \Gamma \vdash x : \text{Val} \rrbracket (\gamma, \sigma) &= \gamma(x) \\ \llbracket \Gamma \vdash \mathbf{x} : \text{Val} \rrbracket (\gamma, \sigma) &= \sigma(\mathbf{x}) \\ \llbracket \Gamma \vdash n : \text{Val} \rrbracket (\gamma, \sigma) &= n \\ \llbracket \Gamma \vdash M \text{ op } N : \text{Val} \rrbracket (\gamma, \sigma) &= \llbracket \Gamma \vdash M : \text{Val} \rrbracket (\gamma, \sigma) \text{ op } \llbracket \Gamma \vdash N : \text{Val} \rrbracket (\gamma, \sigma) \\ \llbracket \Gamma \vdash M \text{ comp } N : \text{Val} \rrbracket (\gamma, \sigma) &= \begin{cases} 1 & \text{if } \llbracket \Gamma \vdash M : \text{Val} \rrbracket (\gamma, \sigma) \text{ comp } \llbracket \Gamma \vdash N : \text{Val} \rrbracket (\gamma, \sigma) \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

**Definition 77** (Interpretation of Permission Expressions). We interpret permission expressions as functions from interpretations of their contexts into  $Perm$ .

$$\llbracket \Gamma \vdash M : Perm \rrbracket : \llbracket \Gamma \rrbracket \rightarrow Perm$$

$$\begin{aligned} \llbracket \Gamma \vdash x : Perm \rrbracket (\gamma) &= \gamma(x) \\ \llbracket \Gamma \vdash 0p : Perm \rrbracket (\gamma) &= \perp \\ \llbracket \Gamma \vdash 1p : Perm \rrbracket (\gamma) &= \top \\ \llbracket \Gamma \vdash \sim M : Perm \rrbracket (\gamma) &= (\llbracket \Gamma \vdash M : Perm \rrbracket (\gamma))^C \\ \llbracket \Gamma \vdash M \cdot N : Perm \rrbracket (\gamma) &= \llbracket \Gamma \vdash M : Perm \rrbracket (\gamma) \wedge \llbracket \Gamma \vdash N : Perm \rrbracket (\gamma) \end{aligned}$$

**Definition 78** (Interpretation of Region Expressions). We interpret region expressions as functions from interpretations of their contexts to region identifiers:

$$\llbracket \Gamma \vdash M : Region \rrbracket : \llbracket \Gamma \rrbracket \rightarrow RegionId$$

$$\llbracket \Gamma \vdash x : Region \rrbracket (\gamma) = \gamma(x)$$

**Definition 79** (Interpretation of Pure Assertions). We interpret pure assertions as functions from interpretations of their contexts into  $2$ . The codomain  $2$  is a complete boolean algebra, so we omit the usual, pointwise definitions of the standard logical connectives.

$$\llbracket \Gamma \vdash M : Pure \rrbracket : \llbracket \Gamma \rrbracket \times Stack \rightarrow 2$$

$$\begin{aligned} \llbracket \Gamma \vdash M =_{\tau} N : Pure \rrbracket (\gamma, \sigma) &\iff \\ \llbracket \Gamma \vdash M : \tau \rrbracket (\gamma, \sigma) &=_{\tau} \llbracket \Gamma \vdash N : \tau \rrbracket (\gamma, \sigma) \end{aligned}$$

$$\begin{aligned} \llbracket \Gamma \vdash \forall x : \tau. M : Pure \rrbracket (\gamma, \sigma) &\iff \\ \forall v \in \llbracket \tau \rrbracket. \llbracket \Gamma, x : \tau; \sigma \vdash M : Pure \rrbracket (\gamma[x \mapsto v], \sigma) & \end{aligned}$$

$$\begin{aligned} \llbracket \Gamma \vdash \exists x : \tau. M : Pure \rrbracket (\gamma, \sigma) &\iff \\ \exists v \in \llbracket \tau \rrbracket. \llbracket \Gamma, x : \tau; \sigma \vdash M : Pure \rrbracket (\gamma[x \mapsto v], \sigma) & \end{aligned}$$

$$\begin{aligned} \llbracket \Gamma \vdash \text{compatible}(M, N) : Pure \rrbracket (\gamma, \sigma) &\iff \\ \llbracket \Gamma \vdash M : Perm \rrbracket (\gamma) \wedge \llbracket \Gamma \vdash N : Perm \rrbracket (\gamma) &\neq \perp \end{aligned}$$

**Definition 80** (Interpretation of Guard Expressions). We interpret guard expressions as expressions in  $Guard$ , the free PCMZ over primitive guard symbols:

$$\llbracket \Gamma \vdash G : Guard \rrbracket : \llbracket \Gamma \rrbracket \times Stack \rightarrow |Guard|$$

$$\begin{aligned} \llbracket \Gamma \vdash G : Guard \rrbracket (\gamma, \sigma) &= G \\ \llbracket \Gamma \vdash G[M] : Guard \rrbracket (\gamma, \sigma) &= G[\llbracket \Gamma \vdash M : Perm \rrbracket (\gamma)] \\ \llbracket \Gamma \vdash G(M) : Guard \rrbracket (\gamma, \sigma) &= G(\llbracket \Gamma \vdash M : Val \rrbracket (\gamma, \sigma)) \\ \llbracket \Gamma \vdash M * N : Guard \rrbracket (\gamma, \sigma) &= \llbracket \Gamma \vdash M : Guard \rrbracket (\gamma, \sigma) \cdot \llbracket \Gamma \vdash N : Guard \rrbracket (\gamma, \sigma) \end{aligned}$$

**Definition 81** (Region Type Guard Algebra Assignment). We denote the set of guard algebra assignments corresponding to a well-formed region type context  $R$  as  $GA(R)$ , and define it as follows:

$$GA(R) := \{\rho \mid \text{dom}(\rho) = R\}$$

**Definition 82** (Interpretation of Assertions). We interpret assertions as functions from interpretations of their contexts into upwards closed subsets of heaps and region assignments. Upwards closed subsets of elements drawn from a partial commutative monoid has enough structure to support standard intuitionistic separation logic, so we omit the standard, pointwise interpretations for now.

$$\llbracket R; \Gamma \vdash M : \text{Assn} \rrbracket : \text{GA}(R) \times \llbracket \Gamma \rrbracket \times \text{Stack} \rightarrow \mathcal{P} \uparrow (\text{Heap} \times \text{RegionAssignment})$$

$$\begin{aligned} \llbracket R; \Gamma \vdash M : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &= \begin{cases} \top & \text{if } \llbracket \Gamma \vdash M : \text{Pure} \rrbracket (\gamma, \sigma) \\ \perp & \text{otherwise} \end{cases} \\ \llbracket R; \Gamma \vdash M ? N : O : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &= \begin{cases} \llbracket R; \Gamma \vdash N : \text{Assn} \rrbracket (\rho, \gamma, \sigma) & \text{if } \llbracket \Gamma \vdash M : \text{Pure} \rrbracket (\gamma, \sigma) \\ \llbracket R; \Gamma \vdash O : \text{Assn} \rrbracket (\rho, \gamma, \sigma) & \text{otherwise} \end{cases} \\ \llbracket R; \Gamma \vdash x@(M) : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &= \{(h, \alpha) \mid \forall r, G, f. \\ &\quad \wedge \bar{f}(\llbracket \Gamma \vdash M : \text{Guard} \rrbracket (\gamma, \sigma)) \sqsubseteq_G \bar{f}(\alpha(r).\text{guards}) \\ &\quad \wedge r = \llbracket \Gamma \vdash x : \text{Region} \rrbracket (\gamma) \\ &\quad \wedge (G, f) = \rho(\pi_1(\alpha(r).\text{type}))\} \\ \llbracket R; \Gamma \vdash \mathbb{T}(x, \vec{M}, N) : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &= \{(h, \alpha) \mid \forall r, s, \vec{v}. \exists \vec{x}, \vec{\tau}. \\ &\quad \wedge \alpha(r).\text{type} = (\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}) \wedge \alpha(r).\text{state} = s \\ &\quad \wedge r = \llbracket \Gamma \vdash x : \text{Region} \rrbracket (\gamma) \\ &\quad \wedge v_i = \llbracket \Gamma \vdash M_i : \tau_i \rrbracket (\gamma, \sigma) \\ &\quad \wedge s = \llbracket \Gamma \vdash N : \text{Val} \rrbracket (\gamma, \sigma)\} \\ \llbracket R; \Gamma \vdash M \mapsto N : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &= \{(h, \alpha) \mid x \in \text{Addr} \wedge h(x) = \llbracket \Gamma \vdash N : \text{Val} \rrbracket (\gamma, \sigma) \\ &\quad \wedge x = \llbracket \Gamma \vdash M : \text{Val} \rrbracket (\gamma, \sigma)\} \\ \llbracket R; \Gamma \vdash M \mapsto [N] : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &= \{(h, \alpha) \mid \exists \vec{v}. x \in \text{Addr} \wedge n > 0 \wedge h(x, \dots, x + (n - 1)) = \vec{v} \\ &\quad \wedge x = \llbracket \Gamma \vdash M : \text{Val} \rrbracket (\gamma, \sigma) \\ &\quad \wedge n = \llbracket \Gamma \vdash N : \text{Val} \rrbracket (\gamma, \sigma)\} \\ \llbracket R; \Gamma \vdash M * N : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &= \llbracket R; \Gamma \vdash M : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * \llbracket R; \Gamma \vdash N : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \\ \llbracket R; \Gamma \vdash \text{emp} : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &= \{(\text{emp}, \alpha) \mid \forall \alpha\} \\ \llbracket R; \Gamma \vdash M \wedge N : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &= \llbracket R; \Gamma \vdash M : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \cap \llbracket R; \Gamma \vdash N : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \\ \llbracket R; \Gamma \vdash M \vee N : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &= \llbracket R; \Gamma \vdash M : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \cup \llbracket R; \Gamma \vdash N : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \end{aligned}$$

**Convention 83.** For particular  $\rho$  and  $\gamma$  we write  $\llbracket R; \Gamma \vdash M : \text{Assn} \rrbracket (\rho, \gamma)$  to denote the function

$$\lambda \sigma. \llbracket R; \Gamma \vdash M : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$$

**Definition 84** (Interpretation of Entailments). We interpret a well-typed derivation of entailment of assertions as functions from the interpretations of their environment into 2. The domain of interpretation of assertions is a complete BI-algebra, and entailment corresponds to subset inclusion.

$$\llbracket R; \Gamma \mid \Delta \vdash Q \rrbracket : \text{GA}(R) \times \llbracket \Gamma \rrbracket \times \text{Stack} \rightarrow 2$$

$$\begin{aligned} \llbracket R; \Gamma \mid \Delta \vdash Q \rrbracket (\rho, \gamma, \sigma) &\iff \\ \bigwedge_{P \in \Delta} \llbracket R; \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &\subseteq \llbracket R; \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \end{aligned}$$

## 8.1 Interpretation of Specifications

We need some auxilliary definitions before we can define the interpretation of specifications.

**Definition 85** ( $\text{GADef}(-, -)$ , Definedness of Guard w.r.t. Guard Algebra).

$$\begin{aligned} \text{GADef} &\subseteq \text{GuardAlgebra} \times \text{Guard} \\ \text{GADef}((G; f), gs) &\stackrel{\text{def}}{\iff} \bar{f}(gs) \neq \perp_G \end{aligned}$$

**Definition 86** ( $\text{wf}(-, -, -)$ , Well-Formedness of Region Assignments).

$$\begin{aligned} \text{wf} &\subseteq \text{GuardAlgebraAssignment} \times \text{RegionAssignment} \\ \text{wf}(\rho, a) &\stackrel{\text{def}}{\iff} \forall r \in \text{dom}(a), \text{GADef}(\rho(\pi_1(a(r).\text{type})), a(r).\text{guards}) \end{aligned}$$

**Definition 87** ( $\text{collapse}_s^o(-, -)$ , Abstract Configuration Collapse). Collapsing combines the information in open regions and the current heap into a single assertion. It uses a region type assignment to interpret regions and a set of regions that are not opened in order to collapse only the open regions.

$$\begin{aligned} \text{collapse}_s^o &: (\text{Heap} \times \text{RegionAssignment}) \rightarrow \mathcal{P}(\text{Heap} \times \text{RegionAssignment}) \\ \text{collapse}_s^o(h, a) &= \{(h, a)\} * \otimes_{r \in (\text{dom}(a) \setminus s)} \rho(r).\text{int}(a(r).\text{type}) \end{aligned}$$

**Definition 88** ( $\lfloor - \rfloor_s$ , Erasure). We define a notion of erasure - the concrete heaps that are all described by the same abstract configuration. It is defined with respect to a region type assignment and a collection of region as that are not to be collapsed:

$$\begin{aligned} \lfloor - \rfloor_s &: (\text{Heap} \times \text{RegionAssignment}) \times \text{RegionTypeAssignment} \times \mathcal{P}(\text{RegionId}) \rightarrow \mathcal{P}(\text{Heap}) \\ \lfloor (h, a) \rfloor_s^o &:= \{h' \mid (h', a') \in \text{collapse}_s^o(h, a) \wedge \text{wf}(\text{GA}(\rho), a')\} \end{aligned}$$

**Definition 89** (Interpretation of Guard Algebra Declaration). We interpret a guard algebra declaration as a guard algebra: a PCMZ  $M$  and a function  $f : |\text{Guard}| \rightarrow |M|$ , as follows:

$$\begin{aligned} \llbracket 0 \rrbracket &= (2, \{\}) \quad (\text{the empty function}) \\ \llbracket G \rrbracket &= (3, \{G \mapsto \text{point}\}) \\ \llbracket \%G \rrbracket &= (\text{Perm}, \{G[\pi] \mapsto \pi\}) \\ \llbracket \#G \rrbracket &= (3^{\text{Val}}, \{G(v) \mapsto [v \mapsto \text{point}]\}) \\ \llbracket X * Y \rrbracket &= (M \times N, (i \circ f) \cup (j \circ g)) \end{aligned}$$

where

$$\begin{aligned} i &: M \rightarrow M \times N \\ i(m) &= (m, \epsilon_N) \\ j &: N \rightarrow M \times N \\ j(n) &= (\epsilon_M, n) \end{aligned}$$

$$(M, f) = \llbracket X \rrbracket$$

$$(N, g) = \llbracket Y \rrbracket$$

and  $f$  and  $g$  have disjoint domains

$$\llbracket X + Y \rrbracket = (M + N, [f, 0] \cup [g, 1])$$

where

$$(M, f) = \llbracket X \rrbracket$$

$$(N, g) = \llbracket Y \rrbracket$$

and  $f$  and  $g$  have disjoint domains

We can extend  $f : |\text{Guard}| \rightarrow |M|$  to a total function by mapping everything else to zero. From this, we have a PCMZ homomorphism  $\bar{f} : \text{Guard} \rightarrow M$ .

Moreover, if generated from the above constructions, it will be surjective, and so  $\text{im}(\bar{f}) \cong M$  (as PCMZs).  $\bar{f}$  defines an equivalence relation  $\ker(\bar{f})$  on  $\text{Guard}$ , with  $\text{Guard} / \ker(\bar{f}) \cong \text{im}(\bar{f})$ .

**Definition 90** (Interpretation of Region Interpretation Declarations). We interpret a region interpretation declaration, well-formed with respect to a set of region types  $R$ , identifier  $r$  and variables  $\vec{x} : \vec{\tau}$  as a function from interpretations of these contexts, to region interpretations:

$$GA(R) \times \llbracket r : \text{Region}, \vec{x} : \vec{\tau} \rrbracket \rightarrow \text{RegionInterpretation}$$

Given a guard algebra assignment  $\rho \in GA(R)$ , a substitution  $\gamma \in \llbracket r : \text{Region}, \vec{x} : \vec{\tau} \rrbracket$ , for each clause  $(\Delta).\Pi \mid e : P$  we *conditionally* extend the resulting map from *AbstractState* to *Assertion* by the following:

$$v \mapsto \llbracket R; r : \text{Region}, \vec{x} : \vec{\tau}, \Delta \vdash P : \text{Assn} \rrbracket (\rho, \gamma \cdot \delta, \sigma)$$

if there is a  $\delta \in \llbracket \Delta \rrbracket$  such that

$$\llbracket R; r : \text{Region}, \vec{x} : \vec{\tau}, \Delta \vdash \Pi : \text{Pure} \rrbracket (\gamma \cdot \delta, \sigma)$$

and  $v = \llbracket \Gamma \vdash e : \text{Val} \rrbracket (\gamma \cdot \delta, \sigma)$  hold, for any  $\sigma$  at all, as none of these assertions can mention program variables.

Since no clause overlap, there will be at most one clause for which the condition is satisfied, hence this is a well-defined partial map.

**Definition 91** (Interpretation of Action Declarations). We can region interpret an action declaration, well-formed with respect to a set of region types  $R$ , identifier  $r$  and variables  $\vec{x} : \vec{\tau}$ , as describing an *LTS* as function of interpretations of its context:

$$GA(R) \times \llbracket r : \text{Region}, \vec{x} : \vec{\tau} \rrbracket \rightarrow \text{LTS}$$

Given a guard algebra assignment  $\rho \in GA(R)$ , a substitution  $\gamma \in \llbracket r : \text{Region}, \vec{x} : \vec{\tau} \rrbracket$ , for each clause  $(\Delta).P \mid G : e_1 \rightsquigarrow e_2$  we extend the mapping as follows:

$$F \mapsto \{ (\llbracket \Gamma \vdash e_1 : \text{Val} \rrbracket (\gamma \cdot \delta, \sigma), \llbracket \Gamma \vdash e_2 : \text{Val} \rrbracket (\gamma \cdot \delta, \sigma)) \mid \llbracket \Gamma \vdash G : \text{Guard} \rrbracket (\rho, \gamma \cdot \delta) = F \}^*$$

if there exists a  $\delta \in \llbracket \Delta \rrbracket$  such that  $\llbracket R; r : \text{Region}, \vec{x} : \vec{\tau}, \Delta \vdash P : \text{Assn} \rrbracket (\rho, \gamma \cdot \delta, \sigma)$  for any stack  $\sigma$  at all, where  $*$  indicates the reflexive, transitive closure of the set.

**Definition 92** (Guard Compatability Complement). We can define the “compatible complement” to a collection of guards  $M \subseteq |\text{Guard}|$  with respect to a particular guard algebra  $G$  as follows:

$$\overline{M} = \{ g' \in |\text{Guard}| \mid \forall g \in M, G\text{Def}(G, g \cdot g') \}$$

**Definition 93** (Rely( $-$ ), The Rely Relation). We can describe the interference allowed by other threads by help of a region type assignment  $\rho$ , considering all the transitions allowed by guards compatible with ours:

$$\text{Rely}(\rho) \subseteq (\text{Heap} \times \text{RegionAssignment}) \times (\text{Heap} \times \text{RegionAssignment})$$

We say two abstract configurations  $(h_1, a_1)$  and  $(h_2, a_2)$  are related iff

- $h_1 = h_2$
- $\text{dom}(a_1) \subseteq \text{dom}(a_2)$
- $\forall r \in \text{dom}(a_1). a_1(r).\text{guards} = a_2(r).\text{guards}$
- $\forall r \in \text{dom}(a_1), (a_1(r).\text{state}, a_2(r).\text{state}) \in \rho(a_1(r).\text{type}).\text{Its}(\overline{a_1(r).\text{guards}}))$ , where the guard compatibility complement is with respect to the guard algebra  $\rho(a_1(r).\text{type}).GA$

**Definition 94** (Interpretation of Region Contexts). We interpret a well-formed region context  $R$  into a region type assignment:

$$\llbracket R \rrbracket : \text{RegionTypeAssignment}$$

For each associated  $\mathbb{T}(r, \vec{x} : \vec{\tau})(g, i, a)$ , let  $\llbracket g \rrbracket$  be the guard algebra corresponding to  $g$ . Then, let  $\llbracket i \rrbracket (\llbracket g \rrbracket, \vec{v})$  and  $\llbracket a \rrbracket (\llbracket g \rrbracket, \vec{v})$  be the region interpretation and LTS corresponding to  $i$  and  $a$ , respectively, under valuation  $\vec{v}$ , such that  $v_i \in \llbracket \tau_i \rrbracket$ , of the region parameters. Then we extend the resulting map as follows:

$$(\mathbb{T}(r, \vec{x} : \vec{\tau}), \vec{v}) \mapsto (\llbracket i \rrbracket (\llbracket g \rrbracket, \vec{v}), \llbracket a \rrbracket (\llbracket g \rrbracket, \vec{v}), \llbracket g \rrbracket)$$

This definition is well-formed as a map as each region type is bound once in the region context.

**Definition 95** (Stabilisation). We define the stabilisation of an assertion  $P$  as intuitively the strongest weaker stable assertion:

$$\text{stabilize}_\rho(P) := \{q \mid \exists p \in P. (p, q) \in \text{Rely}(\rho)\}$$

We remark that by reflexivity of the rely relation, it is always the case that  $P \leq \text{stabilize}_\rho(P)$ .

**Definition 96** ( $\text{stable}_-( - )$ , Assertion Stability). An assertion  $P$  is stable with respect to region type assignment  $\rho$  when

$$\text{stabilize}_\rho(P) \leq P$$

Hence, stable assertions are equivalent to their stabilizations.

**Definition 97** ( $- \Vdash - \{ - \} \{ - \}$ , Semantic Action Judgement). We define  $\alpha \Vdash_\rho \{P\} \{Q\}$  to hold if and only if, for all  $R \in \text{Assertion}$  such that  $\text{stable}(R)$  we have  $\llbracket \alpha \rrbracket (\llbracket P * R \rrbracket_\emptyset^p) \subseteq \llbracket Q * R \rrbracket_\emptyset^p$ .

**Lemma 98** (Locality of Action Judgement). For all stable assertions  $R$ ,

$$\alpha \Vdash_\rho \{P\} \{Q\} \Rightarrow \alpha \Vdash_\rho \{P * R\} \{Q * R\}$$

*Proof.* Chose the frame in the assumption to be  $R * R'$  where  $R'$  is the given frame in the conclusion of the Lemma. Stable assertions are closed under  $*$ .  $\square$

**Definition 99** ( $\preceq$ , View Shift).

$$P \preceq Q \stackrel{\text{def}}{\iff} \text{id} \Vdash_\rho \{P\} \{Q\}$$

**Lemma 100.** Assertions ordered by semantic entailment forms a partial order.

*Proof.* The action of the identity action on sets of heap is the identity. Hence, assertions under semantic entailment forms a partial order because sets ordered by set inclusion does.  $\square$

**Lemma 101** (Frame Property of Semantic Entailment).

$$P \preceq Q \Rightarrow (\forall R. \text{stable}(R) \Rightarrow P * R \preceq Q * R)$$

*Proof.* Directly from Lemma 98.  $\square$

**Lemma 102** (Entailment is Semantic Entailment).

$$\begin{aligned} R; \Gamma \mid \Delta \vdash Q &\Rightarrow \forall \rho \in \llbracket R \rrbracket, \forall \gamma \in \llbracket \Gamma \rrbracket, \forall \sigma. \\ \bigwedge_{P \in \Delta} \llbracket R; \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) &\preceq \llbracket R; \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \end{aligned}$$

**Lemma 103** ( $\preceq$ -Closure of Action Judgement). If

$$P \preceq P' \qquad \alpha \Vdash_\rho \{P'\} \{Q'\} \qquad Q' \preceq Q$$

then

$$\alpha \Vdash_\rho \{P\} \{Q\}$$

*Proof.* We proceed by direct proof. Suppose a stable frame  $R$  and suppose a configuration  $(h, a)$  in the assertion  $P * R$ . By the definition of assertions, that means we can split  $(h, a)$  into  $(h_P, a_P) \in P$  and  $(h_R, a_R) \in R$ . By  $P \preceq P'$ , we get that  $(h_P, a_P) \in P'$ . Hence, by choosing the frame  $R$  in the assumption, we get that  $\llbracket \alpha \rrbracket((h_P, a_P)) \in Q'$ , and again, in  $Q$ , precisely as desired.  $\square$

**Definition 104** (safe, Execution Safety). We define safe to be a recursively defined relation on

$$\begin{aligned} & \text{RegionTypeAssignment} \times \mathbb{N} \times \text{Env} \times \text{Assertion} \times \text{Stack} \times \\ & \text{Cont} \times (\text{Stack} \rightarrow \text{Assertion}) \times (\text{Stack} \rightarrow \text{Val} \rightarrow \text{Assertion}), \end{aligned}$$

with  $\text{safe}_n^0(E, P, \sigma, \kappa, Q, U)$  intuitively capturing that execution of  $\kappa$  in stack  $\sigma$  satisfying  $P$  will run and, within  $n$  steps, safely halt execution in a stack satisfying  $Q$  or return a value  $v$  satisfying  $U(v)$ .

We say  $\text{safe}_0(E, P, \sigma, \kappa, Q, U)$  always holds, and  $\text{safe}_{n+1}(E, P, \sigma, \kappa, Q, U)$  holds iff ...

1. ... when  $\kappa = \text{skip}$  then
  - (a)  $\text{id} \Vdash_\rho \{P\}\{Q(\sigma)\}$
2. ... when  $\kappa = \text{return } e$  or  $\kappa = \text{return } e; s$ , for some  $e$  and  $s$ , then
  - (a)  $\text{id} \Vdash_\rho \{P\}\{U(\sigma)(\llbracket e \rrbracket_\sigma)\}$
3. ... when there is a forking step  $E \vdash (\sigma, \kappa) \xrightarrow{\text{fork}(f, \vec{v})} (\sigma', \kappa')$  for some  $f$  and  $\vec{v}$ , there exists  $P', F : \text{Assertion}$  and  $(\vec{x}, s)$  such that,
  - (a)  $E(f) = (\vec{x}, s)$
  - (b)  $\text{id} \Vdash_\rho \{P\}\{P' * F\}$
  - (c)  $\text{safe}_n^0(E, P', \sigma', \kappa', Q, U)$
  - (d)  $\text{safe}_n^0(E, F, [\vec{x} \mapsto \vec{v}], s, \lambda_.\top, \lambda_.\lambda_.\top)$
4. ... when there is a non-forking step  $E \vdash (\sigma, \kappa) \xrightarrow{\alpha} (\sigma', \kappa')$  there is a  $P' : \text{Assertion}$  such that
  - (a)  $\alpha \Vdash_\rho \{P\}\{P'\}$
  - (b)  $\text{safe}_n^0(E, P', \sigma', \kappa', Q, U)$

We omit the super-script  $\rho$  when it is clear from context. At any one time there will only be a single such region type assignment in play.

**Definition 105** (Environment/Function Spec Agreement).

$$- \Vdash_- - : - \subseteq \text{Env} \times \text{RegionTypeAssignment} \times \mathbb{N} \times \text{Fun} \times \text{FunctionSpec}$$

An environment implements a function specification of  $f$  for  $n$  steps, noted

$$E \Vdash_n^0 f : (\Gamma, \vec{y})\{P\}\{r.Q\}$$

if and only if there exists  $\vec{x} \in \text{Var}^*$  and  $s \in \text{Stmt}$  such that  $E(f) = (\vec{x}, s)$  and, for all  $\gamma \in \llbracket \Gamma, \vec{y} : \text{Val} \rrbracket$ ,  $\sigma \in \text{Stack}$ ,

$$\begin{aligned} & \text{safe}_n^0(E, \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{x} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \sigma, \\ & \quad s, \\ & \quad \lambda\sigma. \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash \forall r.Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \lambda\sigma.\lambda v. \llbracket R; \Gamma, \vec{y} : \text{Val}, r : \text{Val} \vdash Q : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v], \sigma)) \end{aligned}$$

**Definition 106** (Environment/Specification Context Agreement).

$$- \models - \subseteq Env \times RegionTypeAssignment \times \mathbb{N} \times SpecCtxt$$

An environment  $E$  implements a specification context  $\Phi$  for  $n$  steps, noted  $E \models_n^o \Phi$  when, for each individual  $\mathbf{f} \in \text{dom}(\Phi)$ ,  $E \models_n^o \mathbf{f} : \Phi(\mathbf{f})$ .

As with the safe predicate, we elide the region type assignment when clear from the context.

**Definition 107** (Interpretation of “Triples”). We define the interpretation of a Hoare “triple” as a function

$$\llbracket R; \Phi; \Gamma \vdash \{P\} \mathbf{s} \{Q \mid r.U\} : \text{Spec} \rrbracket : \llbracket R \rrbracket \times \llbracket \Gamma \rrbracket \rightarrow \mathcal{P} \downarrow (\mathbb{N})$$

defined as follows:

$$\begin{aligned} \llbracket R; \Phi; \Gamma \vdash \{P\} \mathbf{s} \{Q \mid r.U\} : \text{Spec} \rrbracket (\rho, \gamma) = & \\ & \{n \in \mathbb{N} \mid \forall E : Env. (\forall n' < n. E \models_{n'}^o \Phi) \Rightarrow \forall \sigma : Stack. \\ & \text{safe}_n^o(E, \\ & \quad \llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (GA(\rho), \gamma, \sigma), \\ & \quad \sigma, \\ & \quad \mathbf{s}, \\ & \quad \lambda \sigma. \llbracket \text{dom}(R); \Gamma \vdash Q : \text{Assn} \rrbracket (GA(\rho), \gamma, \sigma), \\ & \quad \lambda \sigma. \lambda v. \llbracket \text{dom}(R); \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (GA(\rho), \gamma[r \mapsto v], \sigma))\} \end{aligned}$$

**Lemma 108** (Interpretation of Triples is Well-Defined). For all  $\rho \in \llbracket R \rrbracket$ ,  $\gamma \in \llbracket \Gamma \rrbracket$ ,

$$\llbracket R; \Phi; \Gamma \vdash \{P\} \mathbf{s} \{Q \mid r.U\} : \text{Spec} \rrbracket (\rho, \gamma)$$

is downwards closed.

**Definition 109** (Interpretation of Atomic Triples). We define the interpretation of atomic triples as a function of well-typed triples as follows:

$$\llbracket R; \Gamma \vdash_S \langle P \rangle \mathbf{s} \langle Q \rangle : \text{Atomic} \rrbracket : \llbracket R \rrbracket \times \llbracket \Gamma \rrbracket \times Stack \rightarrow 2$$

where  $\llbracket R; \Gamma \vdash_S \langle P \rangle \mathbf{s} \langle Q \rangle \rrbracket (\rho, \gamma, \sigma)$  holds if and only if there is  $\alpha, \sigma'$  such that

1.  $E \vdash (\sigma, \mathbf{s}) \xrightarrow{\alpha} (\sigma', \text{skip})$
2. For all  $\vec{r} \in \llbracket S \rrbracket$  and stable assertions  $R$ ,

$$\llbracket \alpha \rrbracket (\llbracket R; \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * R \uparrow_{\vec{r}}^o) \subseteq \llbracket R; \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma') * R \uparrow_{\vec{r}}^o$$

**Definition 110** (Interpretation of Open Region Judgment). We define the interpretation of open region judgments as a function of well-typed judgments as follows:

$$\llbracket R; \Gamma \vdash [P] \text{open} [\{(\Delta_i). (Q_i, \vec{r}_i)\}_{i \in I}] \rrbracket : \llbracket R \rrbracket \times \llbracket \Gamma \rrbracket \times Stack \rightarrow 2$$

where  $\llbracket R; \Gamma \vdash [P] \text{open} [\{(\Delta_i). (Q_i, \vec{r}_i)\}_{i \in I}] \rrbracket (\rho, \gamma, \sigma)$  holds if and only if for all  $p \in \llbracket P \rrbracket$  and frame  $r$ , there is an index  $i \in I$  with  $\delta \in \llbracket \Delta_i \rrbracket$  and  $q \in \llbracket Q_i \rrbracket (\gamma \delta)$  with  $\vec{y} \in \llbracket \vec{r}_i \rrbracket (\gamma \delta)$  such that there is an updated frame  $r'$  with

1.  $(r, r') \in \text{Rely}(\rho)$
2.  $\llbracket p \cdot r \rrbracket_{\emptyset}^o \subseteq \llbracket q \cdot r' \rrbracket_{\vec{y}}^o$

**Definition 111** (Interpretation of Close Region Judgment). We define the interpretation of close region judgments as a function of well-typed judgments as follows:

$$\llbracket \Gamma \vdash [P] \text{close}(\vec{r}) [Q] \rrbracket : \llbracket R \rrbracket \times \llbracket \Gamma \rrbracket \times \text{Stack} \rightarrow 2$$

where  $\llbracket \Gamma \vdash [P] \text{close}(\vec{r}) [Q] \rrbracket (\rho, \gamma, \sigma)$  holds if and only if, for all  $p \in \llbracket R; \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$  and any choice of frame  $r$ , there is a  $q \in \llbracket R; \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$  and an updated frame  $r'$  such that

1.  $(r, r') \in \text{Rely}(\rho)$
2.  $\lfloor p \cdot r \rfloor_{\vec{r}}^{\rho} \subseteq \lfloor q \cdot r' \rfloor_{\emptyset}^{\rho}$

**Definition 112** (Interpretation of Function Specifications). We interpret well-typed function specifications as functions from interpretations of a function context into down-closed sets of natural numbers

$$\llbracket R; \Phi \vdash \mathbf{f}(\vec{x})(\Gamma, \vec{y})\{P\}\mathbf{s}\{r.Q\} : \text{FunctionSpec} \rrbracket : \llbracket R \rrbracket \rightarrow \mathcal{P} \downarrow (\mathbb{N})$$

$$\begin{aligned} \llbracket R; \Phi \vdash \mathbf{f}(\vec{x})(\Gamma, \vec{y})\{P\}\mathbf{s}\{r.Q\} : \text{FunctionSpec} \rrbracket (\rho) = \\ \bigcap_{\gamma \in \llbracket \Gamma, \vec{y} : \text{Val} \rrbracket} \llbracket R; \Phi; \Gamma, \vec{y} : \text{Val} \vdash \{P * (\vec{x} = \vec{y})\} \mathbf{s} \{\forall r.Q \mid r.Q\} : \text{Spec} \rrbracket (\rho, \gamma) \end{aligned}$$

**Definition 113** (Interpretation of Program Specifications). We interpret well-typed program specifications into 2:

$$\llbracket \vdash \vec{r}; \vec{f} : \text{ProgramSpec} \rrbracket \iff \forall n \in \mathbb{N}. \lfloor \vec{f} \rfloor \vDash_n^{\llbracket \vec{r} \rrbracket} \lfloor \vec{f} \rfloor$$

## 9 Soundness

### 9.1 Soundness of Specification Logic

#### 9.1.1 Soundness of Statement Specifications

**Lemma 114** (Required for “Soundness of Function Call”). Suppose  $P : \text{Assertion}$  and  $Q : \text{Term}$  that does not mention program variables. If

$$\begin{aligned} \text{safe}_n(E, P, \sigma', \mathbf{s}, \\ \llbracket \text{dom}(R); \Gamma \vdash \forall r : \text{Val}. Q : \text{Assn} \rrbracket (\rho, \gamma), \\ \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash Q : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v])) \end{aligned}$$

then, for any stack  $\sigma$ , program variable  $x$  and  $U : \text{Stack} \rightarrow \text{Val} \rightarrow \text{Assertion}$ :

$$\text{safe}_n(E, P, \sigma, x := (\sigma', \mathbf{s}), \llbracket \text{dom}(R); \Gamma \vdash \exists r : \text{Val}. x = r * Q : \text{Assn} \rrbracket (\rho, \gamma), U).$$

*Proof.* Proceed by Induction on  $n$ ; assume i.e. the lemma holds at all  $n' < n$ .

Suppose the antecedent along with a  $\sigma$ ,  $x$  and  $U$ . We now have to show.

$$\text{safe}_n(E, P, \sigma, x := (\sigma', \mathbf{s}), \llbracket \text{dom}(R); \Gamma \vdash \exists r : \text{Val}. x = r * Q : \text{Assn} \rrbracket (\gamma), U).$$

To show  $\text{safe}$  is to show the code in question  $\text{safe}$  when it halts, returns and steps. We know that the code  $x := (\sigma', \mathbf{s})$  neither immediately returns or halts, it must take a (possibly forking) execution step. By the operational semantics, which step depends on the shape of the code of the running function,  $\mathbf{s}$ . By case analysis of the operational semantics, there are 4 cases, and we handle each in sequence.

**Halt** Suppose  $s = \text{skip}$ . By assumption on  $s$ , we then know that

$$P \preceq \llbracket \text{dom}(R); \Gamma \vdash \forall r. Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma')$$

We can now step

$$\frac{}{E \vdash (\sigma, \mathbf{x} := (\sigma', \text{skip})) \xrightarrow{\text{id}} (\sigma[\mathbf{x} \mapsto v], \text{skip})}$$

for some  $v$  and thus have to find a  $P' : \text{Assertion}$  such that

(a)  $P \preceq P'$

(b)  $\text{safe}_{n-1}(E, P', \sigma[\mathbf{x} \mapsto v], \text{skip}, \llbracket \text{dom}(R); \Gamma \vdash \exists r : \text{Val}. \mathbf{x} = r * Q : \text{Assn} \rrbracket (\rho, \gamma))$

Pick  $P' := \llbracket \text{dom}(R); \Gamma \vdash \exists r : \text{Val}. \mathbf{x} = r * Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma[\mathbf{x} \mapsto v])$ , and (b) is immediate. To show (a) is to show:

$$P \preceq \llbracket \text{dom}(R); \Gamma \vdash \exists r : \text{Val}. \mathbf{x} = r * Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma[\mathbf{x} \mapsto v])$$

By transitivity of  $\preceq$  it suffices to show that

$$\llbracket \text{dom}(R); \Gamma \vdash \forall r. Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma') \preceq \llbracket \text{dom}(R); \Gamma \vdash \exists r : \text{Val}. \mathbf{x} = r * Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma[\mathbf{x} \mapsto v])$$

Since  $Q$  does not mention program variables,

$$\llbracket \text{dom}(R); \Gamma \vdash \forall r. Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma') = \llbracket \text{dom}(R); \Gamma \vdash \forall r. Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma[\mathbf{x} \mapsto v])$$

and thus, by Lemma 102 it suffices to show that

$$R; \Gamma \mid \forall r. Q \vdash \exists r. \mathbf{x} = r * Q,$$

which is a simple derivation:

$$\frac{\frac{\frac{}{R; \Gamma \mid \forall r. Q \vdash \mathbf{x} = \mathbf{x}}{R; \Gamma \mid \forall r. Q \vdash \mathbf{x} = \mathbf{x} * Q}}{R; \Gamma \mid \forall r. Q \vdash \exists r. \mathbf{x} = r * Q}}$$

**Return** This case is completely analogous to the previous, except we now have a specific value rather than  $v$ ; observe the previous case made no explicit use of  $v$ .

**Non-forking step** If  $s$  is such that

$$E \vdash (\sigma', s) \xrightarrow{\alpha} (\sigma'', s')$$

then we know from the safety of  $s$  that there is a  $P'$  such that  $P \preceq P'$  and

$$\text{safe}_{n-1}(E, P', \sigma'', s', \llbracket \text{dom}(R); \Gamma \vdash \forall r. Q : \text{Assn} \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash Q : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$$

holds. Further, we know by the operational semantics that

$$E \vdash (\sigma, \mathbf{x} := (\sigma', s)) \xrightarrow{\alpha} (\sigma, \mathbf{x} := (\sigma'', s'))$$

and thus we need to find a  $P'$  such that

(a)  $P \preceq P'$

(b)  $\text{safe}_{n-1}(E, P', \sigma, \mathbf{x} := (\sigma', s'), \llbracket \exists r : \text{Val}. \mathbf{x} = r * Q \rrbracket (\gamma))$

We choose the  $P'$  given by assumption, and (a) is immediate.

(b) follows by induction hypothesis, with the necessary premise given by the assumption on  $s$ .

**Forking step** The case of forking steps is analogous to non-forking steps, and hinges on the same observations on the assumption that  $s$  is safe and the hypothesis.  $\square$

**Lemma 115** (Argument-Parameter Substitution). For any  $P$  that does not contain program variables,  $\sigma$  and  $\sigma'$  such that  $\sigma'(\vec{x}) = \llbracket \vec{e} \rrbracket_\sigma$  it's the case that

$$\llbracket R; \Gamma \vdash P * (\vec{e} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq \llbracket R; \Gamma \vdash P * (\vec{x} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma')$$

*Proof.* First, we observe that by the interpretation of assertions, we can compute on the left hand side:

$$\begin{aligned} & \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{e} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \\ &= \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash (\vec{e} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \end{aligned}$$

and then the right:

$$\begin{aligned} & \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{x} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma') \\ &= \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma') * \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash (\vec{x} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma') \end{aligned}$$

Observing that

$$\llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) = \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma')$$

since  $P$  does not refer to program variables, we now have an entailment of the shape  $R * P \preceq R * Q$ , so by Lemma 101 it suffices to show that:

$$\llbracket R; \Gamma, \vec{y} : \text{Val} \vdash (\vec{e} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash (\vec{x} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma')$$

By another appeal to Lemma 101 it suffices to show that

$$\llbracket R; \Gamma, \vec{y} : \text{Val} \vdash e_i = y_i : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash x_i = y_i : \text{Assn} \rrbracket (\rho, \gamma, \sigma')$$

for each  $i$ . This follows easily by computation, which reveals that this indeed holds since  $\preceq$  is reflexive:

$$\begin{aligned} & \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash e_i = y_i : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \\ &= \llbracket e_i \rrbracket_{\sigma} =_{\text{val}} \gamma(y_i) \\ &= \llbracket x_i \rrbracket_{\sigma'} =_{\text{val}} \gamma(y_i) \\ &= \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash x_i = y_i : \text{Assn} \rrbracket (\rho, \gamma, \sigma') \end{aligned}$$

and we are done.  $\square$

**Lemma 116** (Soundness of Function Call). Suppose that

$$\Phi(\mathbf{f}) = (\Gamma, \vec{y}) \{P\} \{r.Q\}.$$

Then for any  $n \in \mathbb{N}$ ,  $\rho \in \llbracket R \rrbracket$  and  $\gamma \in \llbracket \Gamma, \vec{y} : \text{Val} \rrbracket$ ,

$$n \in \llbracket R; \Phi; \Gamma, \vec{y} : \text{Val} \vdash \{P * (\vec{e} = \vec{y})\} \text{x} := \mathbf{f}(\vec{e}) \{\exists r : \text{Val}. x = r * Q \mid r.U\} \rrbracket (\rho, \gamma).$$

*Proof.* Suppose an environment  $E$  such that for all  $n' < n$  we have  $E \models_{n'}^{\rho} \Phi$ . Suppose further a stack  $\sigma$ . We now need to show that

$$\begin{aligned} & \text{safe}_n(E, \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{e} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \sigma, \\ & \quad \text{x} := \mathbf{f}(\vec{e}), \\ & \quad \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash \exists r. x = r * Q : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, \vec{y} : \text{Val}, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]) \end{aligned}$$

We thus need to show Case 4 of Definition 104, where we take a regular execution step as the only applicable rule is

$$\frac{E(\mathbf{f}) = (\vec{x}, \mathbf{s}) \quad \sigma'(\vec{x}) = \llbracket \vec{e} \rrbracket_{\sigma}}{E \vdash (\sigma, \mathbf{x} := \mathbf{f}(\vec{e})) \xrightarrow{\text{id}} (\sigma, \mathbf{x} := (\sigma', \mathbf{s}))}$$

for some  $\sigma' : \text{Stack}$  such that  $\sigma'(\vec{x}) = \llbracket \vec{e} \rrbracket_{\sigma}$ .

Hence  $\mathbf{s}' = \mathbf{x} := (\sigma', \mathbf{s})$  and  $\alpha = \text{id}$ . We now need to choose a suitable  $P'$ , such that

- (a)  $\llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{e} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq P'$ .
- (b)  $\text{safe}_{n-1}(E, P', \sigma, \mathbf{x} := (\sigma', \mathbf{s})), \llbracket \exists r. \mathbf{x} = r * Q \rrbracket (\rho, \gamma), \lambda v. \llbracket U \rrbracket (\rho, \gamma[r \mapsto v])$

We choose  $P' := \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{x} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma')$ .

The statement (b) follows by appeal to Lemma 114, where the premise is obtained by instantiating  $\forall n' < n. E \models_{n'}^{\rho} \Phi$  at  $n-1$ , giving us that  $\mathbf{s}$  is safe to run from  $\llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{x} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma')$ .

(b) is then immediate from the lemma.

(a) follows by Lemma 115. □

**Lemma 117** (Soundness of Value Return). For all  $n \in \mathbb{N}$ ,  $\rho \in \llbracket R \rrbracket$ ,  $\gamma \in \llbracket \Gamma \rrbracket$  and assertion  $Q$ ,

$$n \in \llbracket R; \Phi; \Gamma \vdash \{\top\} \text{ return } \mathbf{e} \{Q \mid r. \mathbf{e} = r\} : \text{Spec} \rrbracket (\rho, \gamma)$$

*Proof.* Suppose an environment  $E$  such that for all  $n' < n$ ,  $E \models_{n'}^{\rho} \Phi$ . Suppose furthermore a stack  $\sigma$ . We need to show that

$$\begin{aligned} & \text{safe}_n(E, \llbracket \text{dom}(R); \Gamma \vdash \top : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \\ & \quad \sigma, \\ & \quad \text{return } \mathbf{e}, \\ & \quad \lambda \sigma'. \llbracket \text{dom}(R); \Gamma \vdash Q : \text{Assn} \rrbracket (\gamma, \sigma') \\ & \quad \lambda \sigma'. \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash \mathbf{e} = r : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v], \sigma')) \end{aligned}$$

We thus need to show that

$$\llbracket \text{dom}(R); \Gamma \vdash \top : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq \llbracket R; \Gamma, r : \text{Val} \vdash \mathbf{e} = r : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto \llbracket \mathbf{e} \rrbracket_{\sigma}], \sigma)$$

which is immediate by calculation:

$$\begin{aligned} & \llbracket R; \Gamma, r : \text{Val} \vdash \mathbf{e} = r : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto \llbracket \mathbf{e} \rrbracket_{\sigma}], \sigma) \\ & = \llbracket \mathbf{e} \rrbracket_{\sigma} =_{\text{val}} \llbracket \mathbf{e} \rrbracket_{\sigma} \end{aligned}$$

which always holds. □

**Lemma 118** (Soundness of If). For all assertions  $P, Q, U$ , and for all  $n \in \mathbb{N}$ ,  $\gamma \in \llbracket \Gamma \rrbracket$  and  $\rho \in \llbracket Rho \rrbracket$ , if

1.  $n \in \llbracket R; \Phi; \Gamma \vdash \{P * \mathbf{e} \neq 0\} \mathbf{s}_1 \{Q \mid r. U\} \rrbracket (\rho, \gamma)$  and
2.  $n \in \llbracket R; \Phi; \Gamma \vdash \{P * \mathbf{e} = 0\} \mathbf{s}_2 \{Q \mid r. U\} \rrbracket (\rho, \gamma)$

then

$$n \in \llbracket R; \Phi; \Gamma \vdash \{P\} \text{ if } (\mathbf{e}) \text{ then } \{\mathbf{s}_1\} \text{ else } \{\mathbf{s}_2\} \{Q \mid r. U\} \rrbracket (\rho, \gamma)$$

*Proof.* Suppose an environment  $E$  such that for all  $n' < n$  we have  $E \models_{n'}^{\rho} \Phi$ . Suppose further a stack  $\sigma$ . We now need to show that

$$\begin{aligned} & \text{safe}_n(E, \llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \sigma, \\ & \quad \text{if } (\mathbf{e}) \text{ then } \{\mathbf{s}_1\} \text{ else } \{\mathbf{s}_2\}, \\ & \quad \llbracket \text{dom}(R); \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v])) \end{aligned}$$

Proceed by cases on the whether result of  $\llbracket e \rrbracket_\sigma$  is different from or equal to zero.

In either case we have to show Case 4 of Definition 104, where we take a regular execution step. In the case where  $\llbracket e \rrbracket_\sigma \neq 0$ , there is only one applicable rule,

$$\frac{\llbracket e \rrbracket_\sigma \neq 0}{E \vdash (\sigma, \text{if}(e) \text{ then } \{s_1\} \text{ else } \{s_2\}) \xrightarrow{\text{id}} (\sigma, s_1)}$$

Hence, we need to show that there is a  $P'$  such that  $\llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq P'$  and

$$\begin{aligned} & \text{safe}_{n-1}(E, P', \\ & \quad \sigma, \\ & \quad s_1, \\ & \quad \llbracket \text{dom}(R); \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v])) \end{aligned}$$

We chose  $P' := \llbracket \text{dom}(R); \Gamma \vdash P * e \neq 0 : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$ . To show the entailment, we compute on both sides. First the right, exploiting that  $e$  does not refer to logical variables.

$$\llbracket \text{dom}(R); \Gamma \vdash P * e \neq 0 : \text{Assn} \rrbracket (\rho, \gamma, \sigma) = \llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * \llbracket e \rrbracket_\sigma \neq_{\text{val}} 0$$

Observing that  $\top$  is unit to  $*$ , we can frame the interpretation of  $P$  away by Lemma 101, and it remains to show that

$$\top \preceq \llbracket e \rrbracket_\sigma \neq_{\text{val}} 0$$

which holds as  $\llbracket e \rrbracket_\sigma \neq 0$  by assumption.

The safety requirement on  $s_1$  follows precisely by the assumption on  $s_1$ .

The case of  $\llbracket e \rrbracket = 0$  is analogous. □

**Lemma 119** (Soundness of Sequencing). For all  $n \in \mathbb{N}$ , if, for any  $\gamma \in \llbracket \Gamma \rrbracket$  and  $\rho \in \llbracket R \rrbracket$ ,

$$n \in \llbracket R; \Phi; \Gamma \vdash \{P\} s_1 \{R \mid r.U\} : \text{Spec} \rrbracket (\rho, \gamma) \quad n \in \llbracket R; \Phi; \Gamma \vdash \{R\} s_2 \{Q \mid r.U\} : \text{Spec} \rrbracket (\rho, \gamma)$$

then

$$n \in \llbracket R; \Phi; \Gamma \vdash \{P\} s_1; s_2 \{Q \mid r.U\} : \text{Spec} \rrbracket (\rho, \gamma)$$

*Proof.* Proceed by induction: assume the lemma holds for all  $n' < n$ . We now show it holds for  $n$ .

Suppose an environment  $E$  such that for all  $n' < n$ ,  $E \models_{n'}^\rho \Phi$ . Suppose further a stack  $\sigma$ . We now need to show that

$$\begin{aligned} & \text{safe}_n(E, \llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \sigma, \\ & \quad s_1; s_2, \\ & \quad \llbracket \text{dom}(R); \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket U \rrbracket (\rho, \gamma[r \mapsto v])) \end{aligned}$$

We case on whether  $s_1 = \text{skip}$ ,  $s_1 = \text{return } e$  or not; whether to skip, return or step.

**Skip** If  $s_1 = \text{skip}$ , we obtain from the first assumption that

$$\llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq \llbracket \text{dom}(R); \Gamma \vdash R : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$$

holds. We also observe that the only possible step is

$$\frac{}{E \vdash (\sigma, \text{skip}; s_2) \xrightarrow{\text{id}} (\sigma, s_2)}$$

We thus now need to show that there is a  $P' : \text{Assertion}$  such that

(a)  $\llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq P'$

(b)  $\text{safe}_{n-1}(E, P', \sigma, s_2, \llbracket \text{dom}(R); \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$

We chose  $P' := \llbracket \text{dom}(R); \Gamma \vdash R : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$  and we have both by assumption.

**Return** If  $s_1 = \text{return } e$  or  $\text{return } e; s'_1$ , we obtain by assumption on  $s_1$  that

$$\llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto \llbracket e \rrbracket_\sigma], \sigma)$$

which is precisely the requirement for

$$\text{safe}_n(E, \llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \sigma, \text{return } e; s_2, \llbracket \text{dom}(R); \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$$

**Forking Step** If

$$E \vdash (\sigma, s_1) \xrightarrow{\text{fork}(f, \vec{v})} (\sigma', s'_1)$$

we obtain by assumption on  $s_1$  variables and code  $(\vec{x}, s)$ , assertions  $P$  and  $F'$  such that

- (a)  $E(f) = (\vec{x}, s)$
- (b)  $\llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq P' * F$
- (c)  $\text{safe}_{n-1}(E, P', \sigma', s'_1, \llbracket \text{dom}(R); \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$
- (d)  $\text{safe}_{n-1}(E, F, [\vec{x} \mapsto \vec{v}], s, \lambda_. \top, \lambda_. \lambda_. \top)$

Hence, the compound statement can step as follows:

$$\frac{E \vdash (\sigma, s_1) \xrightarrow{\text{fork}(f, \vec{v})} (\sigma', s'_1)}{E \vdash (\sigma, s_1; s_2) \xrightarrow{\text{fork}(f, \vec{v})} (\sigma', s'_1; s_2)}$$

and by the definition of safety we now have to show the existence of  $(\vec{x}, s)$  and  $P, F'$  such that

- (a)  $E(f) = (\vec{x}, s)$
- (b)  $\llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq P' * F$
- (c)  $\text{safe}_{n-1}(E, P', \sigma', s'_1; s_2, \llbracket \text{dom}(R); \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$
- (d)  $\text{safe}_{n-1}(E, F, [\vec{x} \mapsto \vec{v}], s, \lambda_. \top, \lambda_. \lambda_. \top)$

We choose precisely the givens obtained by the assumption  $s_1$ . Items (a), (b) and (d) are thus given directly. (c) follows by the induction hypothesis.

**Non-Forking Step** If  $s_1$  can step according to

$$E \vdash (\sigma, s_1) \xrightarrow{\alpha} (\sigma', s'_1)$$

we obtain by assumption on  $s_1$  an assertion  $P'$  such that

- (a)  $\alpha \Vdash_\rho \{ \llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \} \{ P' \}$
- (b)  $\text{safe}_{n-1}(E, P', \sigma', s'_1, \llbracket \text{dom}(R); \Gamma \vdash R : \text{Assn} \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$

That  $s_1$  can step implies that the compound statement can step according to

$$\frac{E \vdash (\sigma, s_1) \xrightarrow{\alpha} (\sigma', s'_1)}{E \vdash (\sigma, s_1; s_2) \xrightarrow{\alpha} (\sigma', s'_1; s_2)}$$

and hence we need to show the existence of  $P'$  such that

- (a)  $\alpha \Vdash_\rho \{ \llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \} \{ P' \}$
- (b)  $\text{safe}_{n-1}(E, P', \sigma', s'_1; s_2, \llbracket \text{dom}(R); \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$

We chose  $P'$  obtained before and get (a) immediately. (b) follows by the induction hypothesis.  $\square$

**Lemma 120** (Soundness of While). For all  $n \in \mathbb{N}$ ,  $\rho \in \llbracket R \rrbracket$  and  $\gamma \in \llbracket \Gamma \rrbracket$ , if

$$n \in \llbracket R; \Phi; \Gamma \vdash \{P * e \neq 0\} \text{ s } \{I \mid r.U\} \rrbracket (\rho, \gamma)$$

then

$$n \in \llbracket R; \Phi; \Gamma \vdash \{I\} \text{ while}(e)\{s\} \{I * e = 0 \mid r.U\} \rrbracket (\rho, \gamma)$$

*Proof.* Proceed by Induction. Assume that the lemma as stated holds for all  $n' < n$ . We now show it holds for  $n$ .

Suppose an environment  $E$  such that for all  $n' < n$  we have  $E \models_{n'}^{\rho} \Phi$ . Suppose further a stack  $\sigma$ . We now need to show that

$$\begin{aligned} & \text{safe}_n(E, \llbracket \text{dom}(R); \Gamma \vdash I : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \sigma, \\ & \quad \text{while}(e)\{s\}, \\ & \quad \llbracket \text{dom}(R); \Gamma \vdash I * e = 0 : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v])). \end{aligned}$$

Proceed by cases on the whether result of  $\llbracket e \rrbracket_{\sigma}$  is different from or equal to zero. In either case we have to show Case 4 of Definition 104, where we take a regular execution step.

**Non-Zero** In the case where  $\llbracket e \rrbracket_{\sigma} \neq 0$ , there is only one applicable rule,

$$\frac{\llbracket e \rrbracket_{\sigma} \neq 0}{E \vdash (\sigma, \text{while}(e)\{s\}) \xrightarrow{\text{id}} (\sigma, \text{s}; \text{while}(e)\{s\})}$$

Hence, we need to show that there is a  $P'$  such that  $\llbracket \text{dom}(R); \Gamma \vdash I : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq P'$  and

$$\begin{aligned} & \text{safe}_{n-1}(E, P' \\ & \quad \sigma, \\ & \quad \text{s}; \text{while}(e)\{s\}, \\ & \quad \llbracket \text{dom}(R); \Gamma \vdash I * e = 0 : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v])). \end{aligned}$$

We chose  $P' := \llbracket \text{dom}(R); \Gamma \vdash I * e \neq 0 : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$ . We thus need to show that

$$\llbracket \text{dom}(R); \Gamma \vdash I : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq \llbracket \text{dom}(R); \Gamma \vdash I * e \neq 0 : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$$

Which follows knowing  $\llbracket e \rrbracket_{\sigma} \neq 0$ . Finally we need to show that

$$\begin{aligned} & \text{safe}_{n-1}(E, \llbracket \text{dom}(R); \Gamma \vdash I * e \neq 0 : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \sigma, \\ & \quad \text{s}; \text{while}(e)\{s\}, \\ & \quad \llbracket \text{dom}(R); \Gamma \vdash I * e = 0 : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v])). \end{aligned}$$

which precisely corresponds to showing that

$$n - 1 \in \llbracket R; \Phi; \Gamma \vdash \{I * e \neq 0\} \text{ s}; \text{while}(e)\{s\} \{I * e = 0 \mid r.U\} \rrbracket (\rho, \gamma)$$

We have by assumption on  $s$  that

$$n - 1 \in \llbracket R; \Phi; \Gamma \vdash \{I * e \neq 0\} \text{ s } \{I \mid r.U\} \rrbracket (\rho, \gamma)$$

and the induction hypothesis gives us

$$n - 1 \in \llbracket R; \Phi; \Gamma \vdash \{I\} \text{ while}(e)\{s\} \{I * e = 0 \mid r.U\} \rrbracket (\rho, \gamma)$$

The two preceding statements are by the Soundness of Sequencing enough to give us precisely the desired conclusion.

**Zero** In the case where  $\llbracket e \rrbracket_\sigma = 0$ , there is only one applicable rule,

$$\frac{\llbracket e \rrbracket_\sigma = 0}{E \vdash (\sigma, \text{while}(e)\{s\}) \xrightarrow{\text{id}} (\sigma, \text{skip})}$$

Hence, we need to show that there is a  $P'$  such that  $\llbracket I \rrbracket (\rho, \gamma, \sigma) \preceq P'$  and

$$\begin{aligned} & \text{safe}_{n-1}(E, P', \\ & \quad \sigma, \\ & \quad \text{skip}, \\ & \quad \llbracket \text{dom}(R); \Gamma \vdash I * e = 0 : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]). \end{aligned}$$

We chose  $P' := \llbracket I * e = 0 \rrbracket (\rho, \gamma, \sigma)$ , and the safety requirement holds by reflexivity of  $\preceq$ . To show the entailment we argue as in the looping case: We can frame  $I$  on each side of the entailment by Lemma 101, and it remains to show that

$$\top \preceq \llbracket \text{dom}(R); \Gamma \vdash e = 0 : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$$

which always holds as  $\llbracket \Gamma \vdash e : \text{Val} \rrbracket (\gamma, \sigma) = \llbracket e \rrbracket_\sigma$  when  $e$  is free of program variables, as here.  $\square$

**Lemma 121** (Soundness of Skip). For all assertions  $P$  and  $U$  and  $\rho \in \llbracket R \rrbracket$  and  $\gamma \in \llbracket \Gamma \rrbracket$ , for all  $n \in \mathbb{N}$ ,

$$n \in \llbracket R; \Phi; \Gamma \vdash \{P\} \text{ skip } \{P \mid r.U\} : \text{Spec} \rrbracket (\rho, \gamma)$$

*Proof.* Suppose an environment  $E$  such that for all  $n' < n$  we have  $E \models_{n'}^\rho \Phi$ . Suppose further a stack  $\sigma$ . We now need to show that

$$\begin{aligned} & \text{safe}_n(E, \llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \\ & \quad \sigma, \\ & \quad \text{skip}, \\ & \quad \llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v])) \end{aligned}$$

which means we need to show that

$$\llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq \llbracket \text{dom}(R); \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma).$$

This follows by reflexivity of  $\preceq$ .  $\square$

**Lemma 122** (Soundness of Local Assignment). For all  $n, \rho \in \llbracket R \rrbracket$ ,  $\gamma \in \llbracket \Gamma \rrbracket$ ,

$$n \in \llbracket R; \Phi; \Gamma \vdash \{x = n\} x := e \{x = e[n/x] \mid r.U\} : \text{Spec} \rrbracket (\rho, \gamma).$$

*Proof.* Suppose an environment  $E$  such that for all  $n' < n$  we have  $E \models_{n'}^\rho \Phi$ . Suppose further a stack  $\sigma$ . We now need to show

$$\begin{aligned} & \text{safe}_n(E, \llbracket \text{dom}(R); \Gamma \vdash x = n : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \sigma, \\ & \quad x := e, \\ & \quad \llbracket \text{dom}(R); \Gamma \vdash x = e[n/x] : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket U \rrbracket (\rho, \gamma[r \mapsto v])). \end{aligned}$$

There is only one applicable transition,

$$\frac{}{E \vdash (\sigma, x := e) \xrightarrow{\text{id}} (\sigma[x \mapsto \llbracket e \rrbracket_\sigma], \text{skip})}$$

So we pick  $P' := \llbracket \text{dom}(R); \Gamma \vdash x = e[n/x] : \text{Assn} \rrbracket (\rho, \gamma, \sigma[x \mapsto \llbracket e \rrbracket_\sigma])$  and have to show that

- (a)  $\llbracket \text{dom}(R); \Gamma \vdash x = n : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq \llbracket \text{dom}(R); \Gamma \vdash x = e[n/x] : \text{Assn} \rrbracket (\rho, \gamma, \sigma[x \mapsto \llbracket e \rrbracket_\sigma])$
- (b)  $\text{safe}_{n-1}(E, \llbracket \text{dom}(R); \Gamma \vdash x = e[n/x] : \text{Assn} \rrbracket (\rho, \gamma, \sigma[x \mapsto \llbracket e \rrbracket_\sigma]), \sigma[x \mapsto \llbracket e \rrbracket_\sigma], \text{skip}, \llbracket \text{dom}(R); \Gamma \vdash x = e[n/x] : \text{Assn}[n/x] \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$

(b) is immediate by Lemma 121.

(a) follows by computation; first on the left hand side:

$$\begin{aligned} & \llbracket \text{dom}(R); \Gamma \vdash x = n : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \\ \iff & \sigma(x) =_{\text{val}} n \end{aligned}$$

Then on the right hand side:

$$\begin{aligned} & \llbracket \text{dom}(R); \Gamma \vdash x = e[n/x] : \text{Assn} \rrbracket (\rho, \gamma, \sigma[x \mapsto \llbracket e \rrbracket_\sigma]) \\ \iff & \llbracket e \rrbracket_\sigma =_{\text{val}} \llbracket \Gamma \vdash e[n/x] : \text{Val} \rrbracket (\gamma, \sigma[x \mapsto \llbracket e \rrbracket_\sigma]) \end{aligned}$$

Hence, by definition of semantic entailment, it suffices to show that, assuming  $\sigma(x) = n$ , for any expression  $e$  and any value  $v$  it holds that

$$\llbracket e \rrbracket_\sigma =_{\text{val}} \llbracket \Gamma \vdash e[n/x] : \text{Val} \rrbracket (\gamma, \sigma[x \mapsto v])$$

We proceed by induction on the structure of  $e$ :

**Constant** If  $e = m$ , for a constant  $m$ , the result is immediate.

**Variable** If  $e = y$  for some program variable  $y$ , there are two cases: if  $x = y$ , we calculate

$$\begin{aligned} \llbracket \Gamma \vdash e[n/x] : \text{Val} \rrbracket (\gamma, \sigma[x \mapsto v]) &= \llbracket \Gamma \vdash x[n/x] : \text{Val} \rrbracket (\gamma, \sigma[x \mapsto v]) \\ &= n \\ &= \sigma(x) \\ &= \llbracket x \rrbracket_\sigma \\ &= \llbracket e \rrbracket_\sigma \end{aligned}$$

as desired. If  $x \neq y$  we calculate

$$\begin{aligned} \llbracket \Gamma \vdash e[n/x] : \text{Val} \rrbracket (\gamma, \sigma[x \mapsto v]) &= \llbracket \Gamma \vdash y[n/x] : \text{Val} \rrbracket (\gamma, \sigma[x \mapsto v]) \\ &= \llbracket \Gamma \vdash y : \text{Val} \rrbracket (\gamma, \sigma[x \mapsto v]) \\ &= \sigma(y) \\ &= \llbracket y \rrbracket_\sigma \\ &= \llbracket e \rrbracket_\sigma \end{aligned}$$

**Binary Operator** If  $e = e_1 \otimes e_2$  for some binary operation, we can calculate as follows:

$$\begin{aligned} \llbracket \Gamma \vdash e : \text{Val}[n/x] \rrbracket (\gamma, \sigma[x \mapsto v]) &= \llbracket \Gamma \vdash e_1 \otimes e_2 : \text{Val}[n/x] \rrbracket (\gamma, \sigma[x \mapsto v]) \\ &= \llbracket \Gamma \vdash e_1[n/x] : \text{Val} \rrbracket (\gamma, \sigma[x \mapsto v]) \otimes \llbracket \Gamma \vdash e_2 : \text{Val}[n/x] \rrbracket (\gamma, \sigma[x \mapsto v]) \\ &= \llbracket e_1 \rrbracket (\sigma) \otimes \llbracket e_2 \rrbracket (\sigma) \\ &= \llbracket e_1 \otimes e_2 \rrbracket_\sigma \\ &= \llbracket e \rrbracket_\sigma \end{aligned}$$

where the third equality holds by the induction hypotheses of  $e_1$  and  $e_2$ . □

**Lemma 123** (Soundness of Frame). For any  $\rho \in \llbracket R \rrbracket$ ,  $\gamma \in \llbracket \Gamma \rrbracket$  and  $\sigma \in \text{Stack}$ , assuming  $\text{mod}(\mathbf{s}) \cap F = \emptyset$ , if

$$n \in \llbracket R; \Phi; \Gamma \vdash \{P\} \mathbf{s} \{Q \mid r.U\} : \text{Spec} \rrbracket (\rho, \gamma, \sigma)$$

then

$$n \in \llbracket R; \Phi; \Gamma \vdash \{P * F\} \mathbf{s} \{Q * F \mid r.U\} : \text{Spec} \rrbracket (\rho, \gamma, \sigma).$$

*Proof.* We proceed by strong induction on  $n$ . Suppose an  $n$  such that it lies in the interpretation of  $R; \Phi; \Gamma \vdash \{P\} \mathbf{s} \{Q \mid r.U\}$ . We now have to show that  $n$  lies in the interpretation of  $R; \Phi; \Gamma \vdash \{P * F\} \mathbf{s} \{Q * F \mid r.U\}$ , which we proceed to do so according to definition:

Suppose an environment  $E$  such that for all  $n' < n$  we have  $E \vDash_{n'}^{\rho} \Phi$ . Suppose further a stack  $\sigma$ . We now need to show

$$\begin{aligned} & \text{safe}_n(E, \llbracket R; \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \sigma, \\ & \quad \mathbf{s}, \\ & \quad \llbracket R; \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v])) \end{aligned}$$

There are 4 cases, according to the definition of safe:

**Skip** If  $\mathbf{s} = \text{skip}$ , then we must show that

$$\text{id} \Vdash_{\rho} \{ \llbracket R; \Gamma \vdash P * F : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \} \{ \llbracket R; \Gamma \vdash Q * F : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \}$$

but by the assumption on  $n$ , we also get that

$$\text{id} \Vdash_{\rho} \{ \llbracket R; \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \} \{ \llbracket R; \Gamma \vdash Q : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \}$$

By the semantics of assertions, the interpretation of assertions commutes with  $*$ , and we then have by Lemma 101 precisely what we need to show.

**Return** This case is analogous to the case of skip.

**Forking Step** In this case, we assume that  $E \vdash (\sigma, \mathbf{s}) \xrightarrow{\text{fork}((\cdot), f, \vec{v})} (\sigma', \kappa')$  for some  $f$  and  $\vec{v}$ . We now have to provide  $P', F'$  and  $(\vec{x}, \mathbf{s})$  such that

1.  $E(f) = (\vec{x}, \mathbf{s})$
2.  $\text{id} \Vdash_{\rho} \{ \llbracket R; \Gamma \vdash P * F : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \} \{ P' * F' \}$
3.  $\text{safe}_{n-1}(E, P', \sigma', \kappa', \llbracket R; \Gamma \vdash Q * F : \text{Assn} \rrbracket (\rho, \gamma), \llbracket U \rrbracket)$
4.  $\text{safe}_{n-1}(E, F, \sigma', \mathbf{s}, \lambda_{\cdot}. \top, \lambda_{\cdot}. \lambda_{\cdot}. \top)$

By appeal to the assumption on  $n$ , we get precisely the pieces we need, however, we get that  $\text{id} \Vdash_{\rho} \{ \llbracket R; \Gamma \vdash P : \text{Assn} \rrbracket \} \{ P' * F' \}$ . Here, we instead of just  $P'$ , we chose  $P' * \llbracket R; \Gamma \vdash F : \text{Assn} \rrbracket (\rho, \gamma, \sigma')$ , our original frame. Hence, by Lemma 98 we get Item 2. Item 4 is then still immediate by assumption. It remains to show Item 3, which now amounts to showing that

$$\text{safe}_{n-1}(E, P' * \llbracket R; \Gamma \vdash F : \text{Assn} \rrbracket (\rho, \gamma, \sigma'), \kappa', \llbracket Q * F \rrbracket (\rho, \gamma), \llbracket U \rrbracket)$$

which we get by assumption on  $n$  combined with the induction hypothesis.

**Non-Forking Step** Analogous to the forking case without the complication of the forked thread.  $\square$

**Lemma 124.** If for any  $\sigma$ ,  $Q'(\sigma) \preceq Q(\sigma)$  and  $\text{safe}_n(E, P, \sigma, s, Q', U)$  then  $\text{safe}_n(E, P, \sigma, s, Q, U)$ .

*Proof.* Proceed by strong induction on  $n$ .

**Skip** If  $s = \text{skip}$ , we must show  $P \preceq Q(\sigma)$  knowing  $P \preceq Q'(\sigma)$  by assumption. This follows by transitivity of  $\preceq$ .

**Return** If  $s$  returns some expression  $e$ , we must show  $P \preceq U(\sigma)(\llbracket e \rrbracket_\sigma)$ , but we get this immediately by assumption.

**Forking Step** Here we must find assertions  $P', F$  such that

1.  $P \preceq P' * F$
2.  $\text{safe}_{n-1}(E, P', \sigma', s', Q, U)$
3.  $\text{safe}_{n-1}(E, F, [\vec{x} \mapsto \llbracket \vec{e} \rrbracket_\sigma], s, \top, \top)$

We get the choice of assertions by the assumption on  $s$ , and Items 1 and 3 are immediate while Item 2 follows by the induction hypothesis.

**Non-forking step** Here we must find assertion  $P'$  such that

1.  $\alpha \Vdash_\rho \{P\} \{P'\}$
2.  $\text{safe}_{n-1}(E, P', \sigma', s', Q, U)$

Both again follow immediately by assumption and the induction hypothesis.  $\square$

**Lemma 125** (Soundness of Consequence). For any  $\rho \in \llbracket R \rrbracket$ ,  $\gamma \in \llbracket \Gamma \rrbracket$  and  $\sigma \in \text{Stack}$ , we have that, assuming

$$R; \Gamma \mid P \vdash P' \qquad R; \Gamma \mid Q' \vdash Q \qquad R; \Gamma \mid U' \vdash U,$$

then for any  $n$  and  $\gamma \in \llbracket \Gamma \rrbracket$ , if

$$n \in \llbracket R; \Phi; \Gamma \vdash \{P'\} s \{Q' \mid r.U'\} : \text{Spec} \rrbracket (\rho, \gamma, \sigma)$$

then

$$n \in \llbracket R; \Phi; \Gamma \vdash \{P\} s \{Q \mid r.U\} : \text{Spec} \rrbracket (\rho, \gamma, \sigma)$$

*Proof.* We proceed according to the proof of the Soundness of Fork: by strong induction on  $n$ , and then by case on the definition of safety.

In each case, we appeal Lemma 103 in order to weaken or strengthen the assertions involved as needed, remarking that the assumed syntactic entailments give us e.g.  $\llbracket R; \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq \llbracket R; \Gamma \vdash P' : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$ .

Again, the return and skipping cases are alike, so we show one. The same is true of the forking and the non-forking case so we show the more complicated of the two.

**Skip** In the case of  $s = \text{skip}$ , we are to show that  $P \preceq Q$ , knowing  $P' \preceq Q'$ ,  $P \preceq P'$  and  $Q' \preceq Q$ , which is directly the statement of Lemma 103.

**Forking Step** In the case that  $s = \text{fork } f(\vec{e})$ , we are to give an implementation  $(\vec{x}, s)$  and assertions  $R$  and  $F$  such that

1.  $P \preceq R * F$
2.  $\text{safe}_{n-1}(E, R, \sigma, s', Q, U)$
3.  $\text{safe}_{n-1}(E, F, [\vec{x} \mapsto \llbracket \vec{e} \rrbracket_\sigma], s, \top, \top)$

The choices of  $(\vec{x}, s)$ ,  $R$  and  $F$  are given by the assumption of the lemma. The first item follows from transitivity of  $\preceq$ , knowing  $P \preceq P'$  and  $P' \preceq R * F$ . The remaining two items follow by assumption with an appeal to Lemma 124.  $\square$

**Lemma 126** (Soundness of Fork). Assuming  $\Phi(\mathbf{f}) = (\Gamma, \vec{y})\{P\}\{r.Q\}$ , it is the case that, for all  $n$ , for all  $\rho \in \llbracket R \rrbracket$ ,  $\gamma \in \llbracket \Gamma \rrbracket$ :

$$n \in \llbracket R; \Phi; \Gamma, \vec{y} : \text{Val} \vdash \{P * (\vec{e} = \vec{y})\} \text{fork } f(\vec{e}) \{ \top \mid r.U \} : \text{Spec} \rrbracket (\rho, \gamma)$$

*Proof.* Suppose an environment  $E$  such that for all  $n' < n$  we have  $E \models_{n'}^\rho \Phi$ . Suppose further a stack  $\sigma$ . We now need to show

$$\begin{aligned} & \text{safe}_n(E, \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{e} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \sigma, \\ & \quad \text{fork } f(\vec{e}), \\ & \quad \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash \top : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, \vec{y} : \text{Val}, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v])) \end{aligned}$$

By case analysis of first the statement in question and then the operational rules, we see that the only applicable case of  $\text{safe}_n$  is the case where

$$E \vdash (\sigma, \text{fork } f(\vec{e})) \xrightarrow{\text{fork}(f, \llbracket \vec{e} \rrbracket_\sigma)} (\sigma, \text{skip}).$$

This means showing that there exists a  $P', F$  and  $(\vec{x}, s)$  such that the following four items hold:

- (a)  $E(\mathbf{f}) = (\vec{x}, s)$ .
- (b)  $\llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{e} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \preceq P' * F$
- (c)  $\text{safe}_{n-1}(E, P', \sigma, \text{skip}, \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash \top : \text{Assn} \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, \vec{y} : \text{Val}, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$
- (d)  $\text{safe}_{n-1}(E, F, [\vec{x} \mapsto \llbracket \vec{e} \rrbracket_\sigma], s, \lambda \_ . \top, \lambda \_ . \top)$

By assumption that  $\mathbf{f}$  is specified by  $\Phi$ , we obtain  $(\vec{x}, s)$  that satisfy (a). We choose

$$P' := \top \quad F := \llbracket R; \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{x} = \vec{y}) : \text{Assn} \rrbracket (\rho, \gamma, [\vec{x} \mapsto \llbracket \vec{e} \rrbracket_\sigma])$$

(c) follows easily: the interpretation  $\llbracket \top \rrbracket (\rho, \gamma, \sigma)$  for any arguments is  $\top$ , and  $\top \preceq \top$ , as required by safety of  $\text{skip}$ . (d) follows from the assumption that  $E \models_{n-1}^\rho \Phi$  by appeal to weakening of the conclusions: we can always weaken to  $\top$ .

Left is (b), which follows precisely from 115.  $\square$

**Lemma 127** (Soundness of Allocation). For all  $\rho \in \llbracket R \rrbracket$ ,  $\gamma \in \llbracket \Gamma \rrbracket$  and  $\sigma$ , for any  $n$  it holds that

$$n \in \llbracket R; \Gamma \vdash \{e = v * v > 0\} \mathbf{x} := \text{alloc}(e) \{ \exists n.x = n * n \mapsto [v] \mid r.U \} : \text{Spec} \rrbracket (\rho, \gamma, \sigma)$$

*Proof.* Suppose an environment  $E$  such that for all  $n' < n$  we have  $E \models_{n'}^{\rho} \Phi$ . Suppose further a stack  $\sigma$ . We now need to show

$$\begin{aligned} & \text{safe}_n(E, \llbracket R; \Gamma \vdash e = v * v > 0 : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \sigma, \\ & \quad x := \text{alloc}(e), \\ & \quad \llbracket R; \Gamma \vdash \exists n. x = n * n \mapsto [v] : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]) \end{aligned}$$

According to the operational semantics, the allocation can either complete successfully or fault, depending on whether  $e$  denotes a non-zero natural number.

If it does not, i.e.  $\llbracket e \rrbracket_{\sigma} < 1$  it is the case that  $E \vdash (\sigma, x := \text{alloc}(e)) \xrightarrow{\zeta} (\sigma, \text{skip})$  and we have to pick a  $P'$  according to Case 4 of the definition of  $\text{safe}_n$ . We chose  $\perp$  and have to show:

1.  $\zeta \Vdash_{\rho} \{ \llbracket R; \Gamma \vdash e = v * v > 0 : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \} \{ \perp \}$
2.  $\text{safe}_n(E, \perp, \sigma, \text{skip}, \llbracket R; \Gamma \vdash \exists n. x = n * n \mapsto [v] : \text{Assn} \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$

Item 2 is vacuously true as  $\perp \preceq P$  holds of any  $P$ . To argue Item 1 we observe that any configuration in the in the interpretation of  $e = v$  and  $v > 0$  would contradict that  $v < 1$ , hence there are none. Therefore, that interpretation is empty, and the inclusion required in Item 1 is empty. (The action of faulting on a set of heaps is thus also irrelevant - the lifting of actions to sets of heaps preserve the empty set).

If  $e > 0$ , it is the case that  $E \vdash (\sigma, x := \text{alloc}(e)) \xrightarrow{\text{alloc}(\llbracket e \rrbracket_{\sigma}, n)} (\sigma[x \mapsto n], \text{skip})$  for some address  $n$  and we thus need to find a  $P'$ , for which we pick the postcondition as stated in the lemma, such that

1.  $\text{alloc}(\llbracket e \rrbracket_{\sigma}, n) \Vdash_{\rho} \{ \llbracket R; \Gamma \vdash e = v * v > 0 : \text{Assn} \rrbracket (\rho, \gamma, \sigma) \} \{ \llbracket R; \Gamma \vdash \exists n. x = n * n \mapsto [v] : \text{Assn} \rrbracket (\rho, \gamma, \sigma[x \mapsto n]) \}$
2.  $\text{safe}_n(E, \llbracket R; \Gamma \vdash \exists n. x = n * n \mapsto [v] : \text{Assn} \rrbracket (\rho, \gamma, \sigma[x \mapsto n]), \sigma, \text{skip}, \llbracket R; \Gamma \vdash \exists n. x = n * n \mapsto [v] : \text{Assn} \rrbracket (\rho, \gamma), \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v]))$

Item 2 holds by the definition of safety of  $\text{skip}$ . To show the first item, we proceed directly by definition of the semantic action judgment: suppose a stable frame, and suppose a heap  $h$  in the erasure of the conjunction of said frame and  $\llbracket R; \Gamma \vdash e = v * v > 0 : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$ . For  $h$  we know that  $\llbracket e \rrbracket_{\sigma} = v$  and  $v > 0$ , consistent with the assumption on  $e$ .

Any  $h'$  in the result of applying the action of the allocation action would satisfy that  $m \in \text{dom}(h')$  if  $m$  is such that  $n \leq m < n + (v - 1)$ . This is ensured knowing  $v > 0$ .

Finally, we need to show that  $h'$  lies in the erasure of  $\llbracket R; \Gamma \vdash \exists n. x = n * n \mapsto [v] : \text{Assn} \rrbracket (\rho, \gamma, \sigma[x \mapsto n])$ . For this to be the case, there needs to be an  $n$  such  $n = n$  and  $n \mapsto v$  is satisfied by the heap. This is precisely the case for  $h'$ . Hence we are done.  $\square$

**Lemma 128** (Soundness of Atomic Embedding). Assuming

$$\llbracket R; \Gamma \vdash [P] \text{ open } [\{(\Delta_i). (P'_i, \vec{r}_i)\}] \rrbracket (\rho, \gamma, \sigma)$$

and, for all  $i \in I$  and  $\delta_i \in \llbracket \Delta_i \rrbracket$ ,

$$\llbracket R; \Gamma, \Delta_i \vdash_{\{\vec{r}_i\}} \langle P'_i \rangle \mathfrak{s} \langle Q_i * \text{newRegion}(\vec{n}_i) \rangle \rrbracket (\rho, \gamma \delta_i, \sigma)$$

$$\llbracket R; \Gamma, \Delta_i \vdash [Q_i * \text{newRegion}(\vec{n}_i)] \text{ close}(\vec{r}_i, \vec{n}_i) [Q] \rrbracket (\rho, \gamma \delta_i, \sigma)$$

then for all  $n \in \mathbb{N}$ ,

$$n \in \llbracket R; \Phi; \Gamma \vdash \{P\} \mathfrak{s} \{\text{stabilize}(Q) \mid r.U\} \rrbracket (\rho, \gamma, \sigma)$$

*Proof.* Assume

1.  $\llbracket R; \Gamma \vdash [P] \text{ open } [\{(\Delta_i) \cdot (P'_i, \vec{r}_i)\}] \rrbracket (\rho, \gamma, \sigma)$
2.  $\llbracket R; \Gamma, \Delta_i \vdash_{\{\vec{r}_i\}} \langle P'_i \rangle \text{ s } \langle Q_i * \text{newRegion}(\vec{n}_i) \rangle \rrbracket (\rho, \gamma \cdot \delta_i, \sigma)$
3. For all  $i \in I$ ,  $\llbracket R; \Gamma, \Delta_i \vdash [Q_i * \text{newRegion}(\vec{n}_i)] \text{ close}(\vec{r}_i, \vec{n}_i) [Q] \rrbracket (\rho, \gamma \cdot \delta_i, \sigma)$

Suppose an environment  $E$  such that for all  $n' < n$  we have  $E \models_{n'}^{\rho} \Phi$ . Suppose further a stack  $\sigma$ . We now need to show that

$$\begin{aligned} & \text{safe}_n(E, \llbracket R; \Gamma \vdash P : \text{Assn} \rrbracket (\rho, \gamma, \sigma), \\ & \quad \sigma, \\ & \quad \mathbf{s}, \\ & \quad \llbracket R; \Gamma \vdash \text{stabilize}(Q) : \text{Assn} \rrbracket (\rho, \gamma), \\ & \quad \lambda v. \llbracket R; \Gamma, r : \text{Val} \vdash U : \text{Assn} \rrbracket (\rho, \gamma[r \mapsto v])). \end{aligned}$$

Since  $\mathbf{s}$  is atomic by Assumption 2, there is only the possibility that the code take a non-forking step according to the operational semantics, one that immediately reaches the skip configuration.

Hence, we can only step according to

$$E \vdash (\sigma, \mathbf{s}) \xrightarrow{\alpha} (\sigma', \text{skip})$$

and we need to find a  $P'$  such that

1.  $\alpha \Vdash_{\rho} \{\llbracket R; \Gamma \vdash P : \text{Assn}(\rho, \gamma, \sigma) \rrbracket\} \{P'\}$
2.  $\text{safe}_{n-1}(E, P', \sigma', \text{skip}, \llbracket R; \Gamma \vdash \text{stabilize}(Q) : \text{Assn} \rrbracket (\rho, \gamma), \llbracket U \rrbracket)$

We chose  $P' := \llbracket R; \Gamma \vdash \text{stabilize}(Q) : \text{Assn} \rrbracket (\rho, \gamma, \sigma')$ , and 2 is immediate. To show 1, assume a stable frame  $R$ . We now need to argue that

$$\llbracket \alpha \rrbracket (\llbracket [P] (\rho, \gamma, \sigma) * R \rrbracket_{\emptyset}^{\rho} \subseteq \llbracket [\text{stabilize}(Q)] (\rho, \gamma, \sigma') * R \rrbracket_{\emptyset}^{\rho})$$

Suppose an abstract configuration  $(h, a)$  in  $\llbracket [P] * R \rrbracket$ . Hence, we can split  $(h, a)$  into  $p = (h_p, a_p) \cdot (h_R, a_R) = r$ . By Assumption 1, we thus get an index  $i \in I$ , a  $\delta \in \llbracket \Delta \rrbracket_i$  and  $p' \in \llbracket [P'_i] \rrbracket$  and  $\vec{y} \in \llbracket [\vec{r}_i] \rrbracket$  and an  $r'$  such that  $(r, r') \in \text{Rely}(\rho)$  and  $\lfloor p \cdot r \rfloor_{\emptyset}^{\rho} \subseteq \lfloor p' \cdot r' \rfloor_{\vec{y}}^{\rho}$ .

By Assumption 2 and since  $p' = (h_{p'}, a_{p'}) \in \llbracket [P'_i] \rrbracket$ , we get  $(\llbracket \alpha \rrbracket (h_{p'}, a_{p'})) \in \llbracket [Q_i * \text{newRegion}(\vec{n}_i)] \rrbracket$

By Assumption 3 we thus get that there is a frame  $r''$  and assertion  $q \in \llbracket [Q] \rrbracket$  such that  $\lfloor (\llbracket \alpha \rrbracket (h_{p'}, a_{p'})) \cdot r \rfloor_{\vec{r}_i, \vec{n}_i}^{\rho} \subseteq \lfloor q \cdot r'' \rfloor_{\emptyset}^{\rho}$ .

By the transitivity of the rely relation, we know that  $r''$  is in the assertion  $R$  as  $R$  is stable, i.e. closed under the rely relation.

Since this is shown for any configuration in  $\llbracket [P] * R \rrbracket$  we have that  $\llbracket [P] * R \rrbracket_{\emptyset}^{\rho} \subseteq \llbracket [Q] * R \rrbracket_{\emptyset}^{\rho}$ . It remains to observe that  $Q \subseteq \text{stabilize}(Q)$  as remarked in Definition 95, and hence, by Lemma 101 we get  $\llbracket [P] * R \rrbracket_{\emptyset}^{\rho} \subseteq \llbracket [\text{stabilize}(Q)] * R \rrbracket_{\emptyset}^{\rho}$  as desired.  $\square$

### 9.1.2 Soundness of Atomic Statement Specifications

**Lemma 129** (Soundness of Atomic Write). For all  $\rho \in R$ ,  $\gamma \in \Gamma$  and  $\sigma$ ,

$$\llbracket R; \Gamma \vdash_S \langle e_1 \mapsto \_ \rangle [e_1] := e_2 \langle e_1 \mapsto e_2 \rangle : \text{Atomic} \rrbracket (\rho, \gamma, \sigma)$$

*Proof.* There are two possible execution steps for the atomic write. If  $e_1$  does not denote a valid address, the write does not succeed, and the code steps according to

$$E \vdash (\sigma, [e_1] := e_2) \xrightarrow{\zeta} (\text{skip}, \sigma)$$

and we have to argue that for any  $s \in \llbracket [S] \rrbracket$  and stable frame  $R$

$$\llbracket \zeta \rrbracket (\llbracket [R; \Gamma \vdash e_1 \mapsto \_ : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^{\rho} \subseteq \llbracket [R; \Gamma \vdash e_1 \mapsto e_2 : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^{\rho})$$

which is trivial as any heap satisfying the precondition will satisfy that  $\llbracket e_1 \rrbracket_\sigma \in \text{Addr}$ , which is a contradiction. There are hence no heaps satisfying the precondition, and the inclusion in the semantic action judgment is thus vacuously satisfied.

Hence, there is only one transition allowed by the operational semantics. The code  $[e_1] := e_1$  can step according to

$$\frac{\llbracket e_1 \rrbracket_\sigma \in \text{Addr}}{E \vdash (\sigma, [e_1] := e_2) \xrightarrow{\text{write}(\llbracket e_1 \rrbracket_\sigma, \llbracket e_2 \rrbracket_\sigma)} (\sigma, \text{skip})}$$

We thus have to argue that the following holds for all  $s \in \llbracket S \rrbracket$  and stable frames  $R$ :

$$\llbracket \text{write}(\llbracket e_1 \rrbracket_\sigma, \llbracket e_2 \rrbracket_\sigma) \rrbracket (\llbracket R; \Gamma \vdash e_1 \mapsto \_ : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^\rho) \subseteq \llbracket R; \Gamma \vdash e_1 \mapsto e_2 : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^\rho$$

We proceed according to definition. Suppose a stable frame  $R$ . And suppose a heap  $h$  in  $\llbracket \llbracket e_1 \rrbracket_\sigma \mapsto \_ \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_\emptyset^\rho$ . We now have to argue that  $\llbracket \text{write}(\llbracket e_1 \rrbracket_\sigma, \llbracket e_2 \rrbracket_\sigma) \rrbracket (h)$  lies in  $\llbracket \llbracket e_1 \rrbracket_\sigma \mapsto e_2 \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_\emptyset^\rho$ . Since  $\llbracket e_1 \rrbracket_\sigma$  is an address by assumption, we know that the heap update is successful, and we know  $h[\llbracket e_1 \rrbracket_\sigma \mapsto \llbracket e_2 \rrbracket_\sigma]$  is in the assertion  $\llbracket R; \Gamma \vdash e_1 \mapsto e_2 : \text{Assn} \rrbracket (\rho, \gamma, \sigma)$  and hence in its erasure.  $\square$

**Lemma 130** (Soundness of Atomic Read). For all  $\rho \in \llbracket R \rrbracket$ ,  $\gamma \in \llbracket \Gamma \rrbracket$  and  $\sigma$ ,

$$\llbracket R; \Gamma \vdash_S \langle e = n * n \mapsto v \rangle x := [e] \langle x = v * n \mapsto v \rangle : \text{Atomic} \rrbracket (\rho, \gamma, \sigma)$$

*Proof.* There are two possible transitions for an atomic write, depending on whether the expression  $e$  denotes a valid address or not.

In the case that  $\llbracket e \rrbracket_\sigma \notin \text{Addr}$ , the code transitions according to

$$E \vdash (\sigma, x := [e]) \xrightarrow{\not\vdash} (\text{skip}, \sigma)$$

and we have to argue that for any  $s \in \llbracket S \rrbracket$  and stable frame  $R$  we have

$$\llbracket \not\vdash \rrbracket (\llbracket R; \Gamma \vdash e = n * n \mapsto v : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^\rho) \subseteq \llbracket R; \Gamma \vdash x = v * n \mapsto v : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^\rho$$

which is trivial as any heap satisfying the precondition will satisfy that  $\llbracket e \rrbracket_\sigma \in \text{Addr}$ , which is a contradiction. There are hence no heaps satisfying the precondition, and the inclusion in the semantic action judgment is thus vacuously satisfied.

Hence, there is only one transition allowed by the operational semantics. The code  $x := [e]$  can step as follows for some value  $v'$ :

$$\frac{\llbracket e \rrbracket_\sigma \in \text{Addr}}{E \vdash (\sigma, x := [e]) \xrightarrow{\text{read}(\llbracket e \rrbracket_\sigma, v')} (\sigma[x \mapsto v'], \text{skip})}$$

Now it remains to show that

$$\llbracket \text{read}(\llbracket e \rrbracket_\sigma, v') \rrbracket (\llbracket R; \Gamma \vdash e = n * n \mapsto v : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^\rho) \subseteq \llbracket R; \Gamma \vdash x = v * n \mapsto v : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^\rho$$

We proceed directly by definition of the semantic action judgment. Suppose a stable  $R$ , and suppose further a heap  $h$  in  $\llbracket R; \Gamma \vdash e = n * n \mapsto v : \text{Assn} \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_\emptyset^\rho$ . We know that  $h(\llbracket e \rrbracket_\sigma) = v$ , hence  $v' = v$ . Then, by the action interpretation,  $\llbracket \text{read}(\llbracket e \rrbracket_\sigma, v) \rrbracket (h) = \{h\}$ .

Hence it suffices to show that  $h \in \llbracket R; \Gamma \vdash x = v * n \mapsto v : \text{Assn} \rrbracket (\rho, \gamma, \sigma[x \mapsto v]) \rrbracket_\emptyset^\rho$ .

By calculation, the semantics of the postcondition, it suffices to show that  $h$  satisfies that  $\sigma[x \mapsto v](x) = v$ , which is trivially satisfied by  $h$ , and  $h(n) = v$ , which it does, as we know  $h(\llbracket e \rrbracket_\sigma) = v$  and  $\llbracket e \rrbracket_\sigma = n$ .

Hence,  $h$  is in the erasure of that assertion.  $\square$

**Lemma 131** (Soundness of CAS). For all  $\rho \in \llbracket R \rrbracket$ ,  $\gamma \in \llbracket \Gamma \rrbracket$  and  $\sigma$ ,

$$\begin{aligned} & \llbracket R; \Gamma \vdash_S \langle e_1 = a * a \mapsto v * e_2 = old * e_3 = new \rangle \\ & \quad x := CAS(e_1, e_2, e_3) \\ & \langle (x \neq 0 * v = old * a \mapsto new) \vee (x = 0 * v \neq old * a \mapsto v) \rangle \rrbracket (\rho, \gamma, \sigma) \end{aligned}$$

*Proof.* Whether the CAS operation succeeds or not is independent of whether the CAS operation completes successfully or not. This depends on whether  $e_1$  denotes a legal address.

In the case that  $\llbracket e_1 \rrbracket_\sigma \notin Addr$ , the code transitions according to

$$E \vdash (\sigma, x := CAS(e_1, e_2, e_3)) \xrightarrow{\not\hookrightarrow} (skip, \sigma)$$

and we have to argue that for any  $s \in S$  and stable assertion  $R$

$$\begin{aligned} & \llbracket \not\hookrightarrow \rrbracket (\llbracket R; \Gamma \vdash e_1 = a * a \mapsto v * e_2 = old * e_3 = new : Assn \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^\rho) \subseteq \\ & \llbracket R; \Gamma \vdash (\dots) \vee (\dots) : Assn \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^\rho \end{aligned}$$

which is trivial as any heap satisfying the precondition will satisfy that  $\llbracket e \rrbracket_\sigma \in Addr$ , which is a contradiction. There are hence no heaps satisfying the precondition, and the inclusion in the semantic action judgment is thus vacuously satisfied.

Hence, there is only one transition allowed by the operational semantics, namely that the CAS operation completes. The code can step as follows for some value  $v'$ :

$$\frac{\llbracket e_1 \rrbracket_\sigma \in Addr}{E \vdash (\sigma, x := CAS(e_1, e_2, e_3)) \xrightarrow{CAS(b, \llbracket e_1 \rrbracket_\sigma, \llbracket e_2 \rrbracket_\sigma, \llbracket e_3 \rrbracket_\sigma)} (\sigma[x \mapsto b], skip)}$$

for some value  $b$ . We proceed in two analogous cases according to whether  $b = 0$  - we show the negative case in which the CAS succeeds.

We have to show that for any  $s \in \llbracket S \rrbracket$  and stable frames  $R$

$$\begin{aligned} & \llbracket CAS(b, \llbracket e_1 \rrbracket_\sigma, \llbracket e_2 \rrbracket_\sigma, \llbracket e_3 \rrbracket_\sigma) \rrbracket (\llbracket R; \Gamma \vdash e_1 = a * a \mapsto v * e_2 = old * e_3 = new : Assn \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^\rho) \subseteq \\ & \llbracket R; \Gamma \vdash (\dots) \vee (\dots) : Assn \rrbracket (\rho, \gamma, \sigma) * R \rrbracket_s^\rho \end{aligned}$$

We proceed directly by definition of the semantic action judgment. Suppose a heap  $h$  that lies in the initial assertion.

We know  $\llbracket e_1 \rrbracket_\sigma \in Addr$ , and furthermore, we know that  $\llbracket e_1 \rrbracket_\sigma \in \text{dom}(h)$ . We also know that  $h(\llbracket e_1 \rrbracket_\sigma) = v$ . Hence, the interpretation of the heap effect depends on whether  $v = old$  or not. In this case it must since we know by assumption that  $b \neq 0$ .

If it does,  $\llbracket CAS(\llbracket e_1 \rrbracket_\sigma, \llbracket e_2 \rrbracket_\sigma, \llbracket e_3 \rrbracket_\sigma, b) \rrbracket (h) = h[\llbracket e_1 \rrbracket_\sigma \mapsto \llbracket e_2 \rrbracket_\sigma]$ . Hence, we need to show that, knowing  $b \neq 0$ ,

$$(h[\llbracket e_1 \rrbracket_\sigma \mapsto \llbracket e_2 \rrbracket_\sigma], a) \in$$

$$\llbracket R; \Gamma \vdash (x \neq 0 * v = old * a \mapsto new) \vee (x = 0 * v \neq old * a \mapsto v) : Assn \rrbracket (\rho, \gamma, \sigma[x \mapsto b])$$

By the semantics of assertions, it is sufficient to demonstrate that it lies in one of the disjuncts, where we obviously choose to show

$$(h[\llbracket e_1 \rrbracket_\sigma \mapsto \llbracket e_2 \rrbracket_\sigma], a) \in \llbracket R; \Gamma \vdash x \neq 0 * v = old * a \mapsto new : Assn \rrbracket (\rho, \gamma, \sigma[x \mapsto b])$$

We know  $b \neq 0$ ,  $\llbracket e_1 \rrbracket_\sigma = a$  and  $\llbracket e_2 \rrbracket_\sigma = v$ , hence the three constraints are immediate.

The alternate case is analogous. □

### 9.1.3 Soundness of Program Specifications

**Lemma 132** (Soundness of Program Specifications). Given a well-typed program specification  $\vdash \vec{r}; \vec{f}$ , assume for each  $f_i$  that, for all  $n \in \mathbb{N}$ ,

$$n \in \llbracket \vec{r}; \vec{f} \rrbracket \vdash f_i : \text{FunctionSpec}.$$

Then,  $\llbracket \vdash \vec{r}; \vec{f} : \text{ProgramSpec} \rrbracket$  holds.

*Proof.* We are to show that, for any given  $n \in \mathbb{N}$ ,  $\llbracket \vec{f} \rrbracket \models_n \llbracket \vec{f} \rrbracket$ .

By assumption, for every function spec  $f_i = \mathbf{g}(\vec{x})(\Gamma, \vec{y})\{P\}\mathbf{s}\{r.Q\}$ , for any  $n$  and  $\gamma \in \llbracket \Gamma, \vec{y} : \text{Val} \rrbracket$ ,  $\rho \in \llbracket R \rrbracket$ ,

$$n \in \llbracket \vec{r}; \vec{f} \rrbracket ; \Gamma, \vec{y} : \text{Val} \vdash \{P * (\vec{x} = \vec{y})\} \mathbf{s} \{\forall r.Q \mid r.Q\} : \text{Spec} \llbracket \rho, \gamma \rrbracket.$$

This in particular means that for any  $E$  such that for any  $n' < n$  we have  $E \models_{n'}^\rho \llbracket \vec{f} \rrbracket$ , it holds for any  $\sigma \in \text{Stack}$  that

$$\begin{aligned} & \text{safe}_n(E, \llbracket \text{dom}(R); \Gamma, \vec{y} : \text{Val} \vdash P * (\vec{x} = \vec{y}) : \text{Assn} \rrbracket \llbracket \rho, \gamma, \sigma \rrbracket, \\ & \quad \sigma, \\ & \quad \mathbf{s}, \\ & \quad \lambda \sigma'. \llbracket \text{dom}(R); \Gamma, \vec{y} : \text{Val} \vdash \forall r.Q : \text{Assn} \rrbracket \llbracket \rho, \gamma, \sigma' \rrbracket, \\ & \quad \lambda \sigma'. \lambda v. \llbracket \text{dom}(R); \Gamma, \vec{y} : \text{Val}, r : \text{Val} \vdash Q : \text{Assn} \rrbracket \llbracket \rho, \gamma[r \mapsto v], \sigma' \rrbracket) \end{aligned}$$

This is precisely Definition 105, of environment/function spec agreement, and in summary we have

$$\forall E. E \models_{n'}^\rho \llbracket \vec{f} \rrbracket \Rightarrow E \models_n^\rho \mathbf{g} : (\Gamma, \vec{y})\{P\}\{r.Q\}$$

Since we have this for every  $\mathbf{g} \in \text{dom}(\llbracket \vec{f} \rrbracket)$ , we have that

$$\forall E. E \models_{n'}^\rho \llbracket \vec{f} \rrbracket \Rightarrow E \models_n^\rho \llbracket \vec{f} \rrbracket.$$

If we instantiate  $E$  to  $\llbracket \vec{f} \rrbracket$  we obtain precisely

$$\llbracket \vec{f} \rrbracket \models_{n'}^\rho \llbracket \vec{f} \rrbracket \Rightarrow \llbracket \vec{f} \rrbracket \models_n^\rho \llbracket \vec{f} \rrbracket.$$

Since we have this for any  $n' < n$ , and for any  $n$ , we can generalize to

$$\forall n. (\forall n' < n. \llbracket \vec{f} \rrbracket \models_{n'}^\rho \llbracket \vec{f} \rrbracket) \Rightarrow \llbracket \vec{f} \rrbracket \models_n^\rho \llbracket \vec{f} \rrbracket$$

which by induction lets us conclude

$$\forall n. \llbracket \vec{f} \rrbracket \models_n^\rho \llbracket \vec{f} \rrbracket$$

as desired. □

**Theorem 133** (Soundness of Program Logic). If  $\vdash \vec{r}; \vec{f} : \text{ProgramSpec}$  is derivable in the program logic, then  $\llbracket \vdash \vec{r}; \vec{f} : \text{ProgramSpec} \rrbracket$  holds.