Local Actions for a Curry-style Operational Semantics *

Gordon Stewart Andrew W. Appel

Princeton University, Princeton, NJ, USA

jsseven@cs.princeton.edu appel@princeton.edu

Abstract

Soundness proofs of program logics such as Hoare logics and type systems are often made easier by *decorating* the operational semantics with information that is useful in the proof. However, modifying the operational semantics to carry around such information can make it more difficult to show that the operational semantics corresponds to what actually occurs on a real machine.

In this work we present a program logic framework targeting operational semantics in *Curry-style*—that is, operational semantics without proof decorations such as separation algebras, share models, and step indexes. Although we target Curry-style operational semantics, our framework permits local reasoning via the frame rule and retains expressive assertions in the program logic. Soundness of the program logic is derived mechanically from simple properties of primitive commands and expressions.

We demonstrate our framework by deriving a separation logic for the model of a core imperative programming language with external function calls. We also apply our framework in a more realistic setting in the soundness proof of a separation logic for CompCert's Cminor. Our proofs are machine-checked in Coq.

Categories and Subject Descriptors D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory — Semantics; F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs — Logics of programs; F.4.1 [*MATHEMATICAL LOGIC AND FORMAL LANGUAGES*]: Mathematical Logic — Mechanical theorem proving

General Terms Languages, Theory, Verification

Keywords Curry-style Operational Semantics, Local Actions, Separation Logic

1. Introduction

Decorating an operational semantics with extra information often makes soundness proofs of program logics such as Hoare logics and type systems easier. This extra information might include, e.g., lock invariants, step indexes, or forms of ghost state. However, modifying the operational semantics to include this extra information comes at a price: one must show that the decorated semantics corresponds to the original intuitive semantics. Otherwise, properties proved with respect to the decorated semantics might not hold of the intuitive semantics.

We faced this problem while attempting to prove the soundness of a separation logic for Cminor, an intermediate language in the CompCert certified compiler stack [14]. CompCert's compiler correctness proof is dependent on the specific memory model used to define the operational semantics of Cminor and the other languages in the stack. Because this memory model makes the formulation of these operational semantics quite natural and well-suited to proofs by bisimulation, we found it undesirable from an engineering standpoint to modify the memory model and the operational semantics of Cminor to support the sorts of extra information useful in our proof of soundness of the separation logic. Instead, we chose to prove our separation logic sound with respect to a decorated semantics and then show that the decorated semantics corresponds to the intuitive semantics already used in CompCert.

This paper presents our solution to the general problem of constructing program logics targeting *Curry-style* operational semantics, that is, those without proof decorations. We call such semantics Curry-style by analogy to Curry-style formulations of the simply typed lambda calculus in which types are seen as decorations of untyped lambda terms. The outline of our approach is as follows: First, we isolate the components of the program logic: (1) the *operational semantics* of the programming language with respect to which the logic is proved sound; (2) *worlds* of the program logic; and, in the case of separation logic, (3) a *separation algebra* on worlds. Then, we reassemble these three components in a way that exposes the right interfaces. This principled reassembly enables us to construct models of state at the level of the program logic that are resilient to changes in the target language. Conversely, sophisticated models of state at the level of the program logic need not complicate the data model of the programming language in our framework. This has a twofold benefit: (1) the operational semantics of the programming language can be stated more simply, making it easier to understand; and (2) undecorated operational semantics are better suited to compiler correctness proofs by bisimulation, e.g. in CompCert. In Section 9, we demonstrate that our approach works for the core of an imperative programming language with external function calls. We also apply our framework in a more realistic setting in the soundness proof of a separation logic for undecorated Cminor.

Contributions.

- We present a separation logic framework targeting generic Curry-style operational semantics. Because worlds of the program logic are distinct from states of the operational semantics in our framework, the operational semantics of the target language can be given without decoration. Soundness of the separation logic is a consequence of simple erasure and safety facts proved about primitive commands and expressions.
- Our framework forms the core of the soundness proof of a separation logic for undecorated Cminor.

* This work supported in part by NSF award CNS-0910448 and AFOSR award FA9550-09-1-0138.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLPV'11, January 29, 2011, Austin, Texas, USA.

Copyright © 2011 ACM 978-1-4503-0487-0/11/01. . . $10.00

- We show how to prove functional correctness properties in a program logic for safety without requiring semantic assertions in the target language.
- We present a reusable Coq library implementing our framework: `http://www.cs.princeton.edu/~jsseven/local`.
- Our proofs are machine-checked in Coq.

2. Background

We set the stage for a more detailed description of our approach by first reviewing separation algebras and the theory of local action, both introduced by Calcagno et al. [7] to describe a class of models of separation logic. We then provide a short introduction to CompCert, share models, and indirection theory. CompCert serves as our motivating example of a case in which it is important whether we can isolate the program logic from the definition of the operational semantics. Share models and indirection theory illustrate two parts of the proof apparatus that we would like to isolate from the operational semantics, but which are useful in the construction of separation logic proofs and separation logics themselves. If they were naively added to CompCert, both share models and indirection theory would present complications for the CompCert model: share models would unduly burden CompCert with too much information, while indirection theory would complicate bisimulation proofs used to prove compiler correctness.

Separation Algebras and Local Action. Separation algebras are mathematical structures (partial, cancellative, commutative monoids) that describe in a general way what it means for two heaplets to be disjoint. They were first used by Calcagno et al. [7] to construct an abstract model of separation logic. In this work we use a formalization of separation algebras in Coq due to Dockins et al. [10], called multi-unit separation algebras, that replaces the partial binary operator of the monoid with a three-place relation called *join* and allows multiple identity elements. A three-place relation is used in order to express noncomputable join relations in Coq. We write that "x joins with y to produce z" with the notation $x \oplus y = z$.

The theory of local action provides a second useful abstraction. Local actions are commands (1) whose effects on the program state can be tracked back to "smaller" states (this is often called the frame property in the literature [7, 21]); and (2) whose safety in small states guarantees safety in extensions of safe small states. It can be shown that a command has both of these properties iff the frame rule is sound for it [7].

The CompCert Memory Model. CompCert is an optimizing C compiler developed by Leroy with formal operational semantics of source, target, and compiler intermediate languages and machine-checked proofs of correctness in Coq of each compiler stage [14]. The main features of CompCert's memory model [15] are generality (the same memory model is used at all levels of the compiler stack) and relocatability (the memory model supports a notion of *memory injection* that is critical in the proofs of many program transformations performed by the compiler).

CompCert memories are byte-addressable. Data in the CompCert model includes integers and floats, in various bit sizes, as well as pointers. To support these features, CompCert's memory model includes theorems for reasoning about encoding and decoding values to and from lists of bytes, called *memvals* in CompCert parlance, with support for big- and little-endian architectures, and a realistic model of fixed-width machine integers.

Permissions vs. Shares. We have been working with Leroy to adapt CompCert's memory model to include *permissions* at each memory cell. These permissions (*free*, *write*, etc.) are the minimal decoration needed to ensure the compiler never performs thread-unsafe optimizations, such as hoisting loads and stores past lock synchronizations. Such permissions don't, however, support the sort of resource accounting required to construct a separation algebra.[1] For example, CompCert's *read* permission, which models situations in which a program can read a location in the heap but not modify or free it, doesn't satisfy the *cancellativity* axiom of separation algebras: in a natural formulation of $\oplus$ for CompCert's permission lattice, both the empty permission and the *read* permission join with *read* to produce *read*, thus violating cancellativity (*read* $\neq$ the empty permission).

Additionally, permissions in CompCert don't have the convenient properties of share models, such as those of Dockins et al. [10], Parkinson [20], and Bornat et al. [6], that allow infinite splitting of shares and token counting. Infinite share splitting is useful in proofs of arbitrary-depth divide-and-conquer algorithms. Token counting can be used to verify that outstanding readers on a lock have all completed. We would like to support such proof idioms in our program logic but don't want to burden the operational semantics with the additional complexity of the required share models.

Indirection. An even more compelling reason to enforce a clear distinction between worlds of the program logic and states of the operational semantics is the need to reason about challenging language features such as function pointers, Pthreads-style locks, and ML-style reference types. Semantic approaches to reasoning about such features, such as indirection theory [13]—which generalizes previous step-indexing approaches to reasoning about recursion [1, 2]—and Kripke models of modal logics [4, 9] appear to work well but require significant machinery to implement. For example, in step-indexed models of locks with higher-order resource invariants in the heap, we must keep track of the level, or "age," of the world at each computation step and *age* the world appropriately in order to prove the soundness of the Hoare rule for `unlock`. Such bookkeeping is necessary in the soundness proof of the Hoare logic but there's no reason to allow this information to leak into the operational semantics and the compiler correctness proof.

3. Our Approach

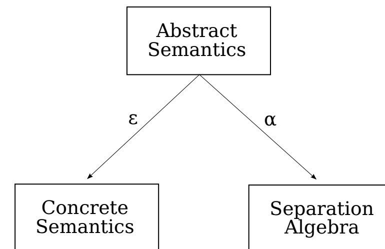


Figure 1: Overview of the Framework

The central insight of our work is synthetic: within an appropriately stratified framework (depicted in Figure 1), one can derive a separation logic for a Curry-style semantics without adapting the semantics to support the usual required machinery, such as a separation algebra. The technique, which is a form of proof by refinement, requires that one define an appropriate abstract command or expression for each primitive command and expression in the Curry-style semantics. Abstract commands and expressions are

[1] We do not mean to imply that the permission model of CompCert is deficient and should be changed, only that the minimally decorated memory model that is convenient for proofs by bisimulation and is easily understood is not the right model for use in the program logic.

appropriate when their concrete counterparts refine, or implement, them. In order to prove the soundness of the frame rule, each such pair of commands must be local in the sense of Calcagno et al. [7].

The worlds and states of the abstract and concrete semantics are related by a collection of *erasure relations* pairing worlds with states, and primitive commands and expressions of the abstract semantics with corresponding primitive commands and expressions of the concrete semantics. In Figure 1, we represent this collection of erasure relations with ε.

The worlds of the abstract semantics induce worlds of the separation logic via a *projection function*, which we call α. The function α maps abstract worlds to separation algebra elements supporting the predicates of separation logic. We choose not to define a separation algebra directly on worlds of the abstract semantics just to increase the generality of our system: there are cases in which we'd like abstract worlds to contain extra information that we don't want to pass to predicates of the separation logic: for example, the thread schedule in the model of a concurrent programming language.

This stratification—of data, primitive expressions, and primitive commands—enables us to couple Curry-style operational semantics with an expressive program logic.

Technical Development. We now describe the framework in more detail (Figure 2).

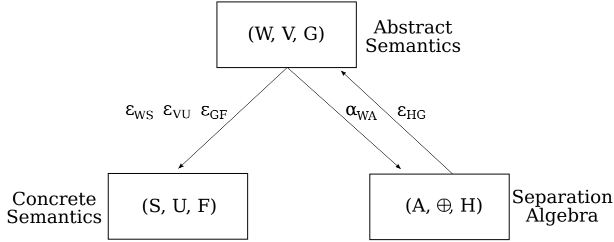


Figure 2: Detailed Overview of the Framework

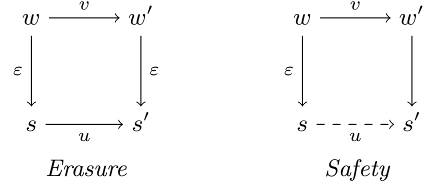
Here, W and S are (resp.) the types of worlds of the program logic and states of the operational semantics. The structure (S, U, F) defines the operational semantics of the object language (also called the concrete language): U is a set of *primitive concrete commands*, while F is a set of *primitive concrete expressions*. Primitive commands are deterministic binary relations on S; primitive expressions are partial functions from S to boolean values. On top of U and F we define the operational semantics of a conventional while-language, in continuation-passing style.

The structure (W, V, G) defines the operational semantics of the *abstract language:* V is a set of (potentially nondeterministic) *abstract primitive commands,* while G is a set of *abstract primitive expressions.* ε_{VU} is a predicate defining the set of corresponding pairs of abstract and concrete commands; ε_{GF}, in turn, defines the set of corresponding pairs of abstract and concrete expressions. We explain what we mean by *corresponding* below.

The *world erasure* projection ε_{WS}, which we'll often just call ε, maps worlds of the program logic to states of the operational semantics.

The structure $(A, \oplus)$ defines a separation algebra over elements of type A. The α-projection α_{WA} maps elements of W to elements of A. We'll often call α_{WA} just α. H defines a set of primitive expressions operating on elements of A while ε_{HG} defines the set of corresponding expressions in H and G. We need primitive expressions H in order to express separation logic assertions such as $\mathsf{assert_expr}(e)$, which is satisfied by elements $a : A$ in which e evaluates to true. Such assertions appear in the proof rules for conditionals and while loops.

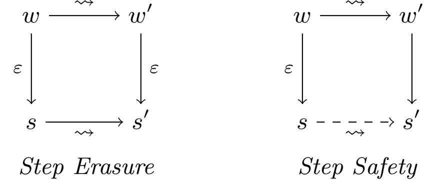
We say that two primitive commands $(v, u) : \varepsilon_{VU}$ correspond to constitute a *stratified primitive command* when v and u have the following properties, expressed as commutative diagrams:



Erasure states that if world w erases to state s, (w, w') is a state transition permitted by v, and (s, s') is a state transition permitted by u, then w' must erase to s'. Because ε is a function, primitive command erasure entails that primitive commands u are deterministic.

Safety states that if world w erases to state s and (w, w') is a state transition permitted by v, then there exists an s s.t. (s, s') is a state transition permitted by u. Primitive expressions (g, f) : ε_{GF} must satisfy corresponding *expression erasure* and *expression safety* properties.

An easy consequence of primitive command erasure and safety are the following extensions to the step relations of the abstract and concrete languages:



Step erasure states that steps in the abstract semantics commute under erasure with steps from corresponding states in the concrete semantics. Step safety states that whenever we can take a step in the abstract semantics we can take a step in the concrete semantics, i.e. abstract steps ensure corresponding concrete steps are nonstuck. We use these two properties in the proof of soundness of the Hoare logic rule for primitive command introduction. In Section 5.1, we prove a corollary simulation theorem relating abstract and concrete semantics, though this theorem is not necessary in the soundness proof of the Hoare logic.

Finally, we construct a separation logic and prove it sound with respect to (S, U, F). Facts proved in the separation logic are true, up to erasure and the α-projection, of programs in (S, U, F). Because of how we interpret the Hoare triple $\vdash \{P\}\ c\ \{Q\}$, each primitive command with a valid Hoare rule in the separation logic *must satisfy the frame rule* and is thus a local action. Because of the erasure and safety properties, the introduction rule for primitive concrete commands in the Hoare logic requires only that a corresponding abstract primitive command be exhibited as a proof witness. This greatly simplifies the soundness proofs of Hoare rules for primitive commands.

Notation. In what follows, we denote abstract worlds with $w \in W$, and concrete states with $s \in S$. Elements $a \in A$ are elements of the separation algebra $(A, \oplus)$. The function $\varepsilon : W \to S$ is an erasure function mapping abstract worlds to concrete states. The function $\alpha : W \to A$ projects a separation algebra element from an abstract world. We use the "forces" notation $\alpha(w) \vDash P$ to express the fact that the α-projection of world w satisfies separation logic predicate P. The assertion $\alpha(w) \vDash P$ is equivalent to the proposition $P(\alpha(w))$.

4. Generic Operational Semantics

We start with the rather conventional notion of *nondeterministic operational semantics.* Such semantics are well-understood and highly expressive: the dynamic behavior of most programming languages can be expressed in such a framework. We say an operational semantics is *generic* when the set of primitive commands is left abstract.

Because we are concerned primarily with modeling data and not control, we present generic operational semantics in continuation-passing style (thus isolating data from control) and we use a conventional expression language evaluating expressions to boolean values in control-flow constructs, with the standard operators for conjunction, disjunction, and negation. Unlike in some models of separation logic [16, 21], expression evaluation in our framework is partial, meaning expressions can get stuck. We assume a separation logic proof will show that the expressions in a particular code fragment are safe to evaluate. We present the syntax and excerpted semantics of our expression language in Figure 3. The elided rules are for disjunction and negation.

Syntax

$$e ::= \mathsf{val}(b) \mid \mathsf{primexpr}(f) \mid \neg e \mid e_1 \mathbin{\&\&} e_2 \mid e_1 \mathbin{||} e_2$$

Evaluation $F \vdash e \Downarrow_s \lfloor b \rfloor$

$$F \vdash \mathsf{val}(b) \Downarrow_s \lfloor b \rfloor$$
$$F \vdash \mathsf{primexpr}(f) \Downarrow_s \lfloor b \rfloor \quad \textit{if} \quad f \in F \textit{ and } f(s) = \lfloor b \rfloor$$
$$F \vdash \mathsf{primexpr}(f) \Downarrow_s \mathsf{none} \quad \textit{if} \quad f \in F \textit{ and } f(s) = \mathsf{none}$$
$$F \vdash e_1 \mathbin{\&\&} e_2 \Downarrow_s \lfloor b \rfloor \quad \textit{if} \quad F \vdash e_1 \Downarrow_s \lfloor b_1 \rfloor \textit{ and } F \vdash e_2 \Downarrow_s \lfloor b_2 \rfloor \textit{ and } b_1 \wedge b_2 = b$$

Figure 3: Expression Syntax and Evaluation (excerpted)

We write $F \vdash e \Downarrow_s \lfloor b \rfloor$ to denote that in context F (a set of primitive expressions) e evaluates in state s to boolean value b. $F \vdash e \Downarrow_s \mathsf{none}$ means e is undefined in state s and context F. We write $e \Downarrow_s \lfloor b \rfloor$ when the context is understood. Letting states be partial functions from local variables to booleans (more realistically, states might *include* stores that are partial functions from local variables to booleans), we can encode local variables in this abstract framework with a special primitive expression $\mathtt{evar}(x) \stackrel{def}{=} \lambda s.\ s(x)$. By the rule for primitive expressions, $\mathsf{primexpr}(\mathtt{evar}(x)) \Downarrow_s \lfloor b \rfloor$ whenever x is defined to equal b in state s.

Definition 1 (Generic Operational Semantics)**.** *A* generic operational semantics *is a triple* $\mathbb{S} = (S, U, F)$*, where* S *is the type of states of the programming language,* $U \subseteq \mathcal{P}(S \times S)$ *is a set of binary relations on* S *modeling the set of* primitive commands*, and* $F \subseteq \mathcal{P}(S \rightharpoonup \mathsf{bool})$ *is a set of partial functions modeling* primitive expressions.

The syntax of languages in this framework is given in Figure 4. $\mathsf{primcom}(u)$ denotes primitive commands $u \in U$. In the rest of this paper, we use the syntax $u : U$ and $u \in U$ interchangeably to denote that u is of type U or that u is an element of U. skip and seq denote nops and the sequential composition of commands, respectively. Conditionals and while loops are standard.

$$\begin{array}{rcl} c & ::= & \mathsf{primcom}(u) \\ & | & \mathsf{skip} \\ & | & c_1;\, c_2 \\ & | & \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\ & | & \mathsf{while}\ e\ \mathsf{do}\ c. \end{array}$$

Figure 4: Syntax of the While-language

Following Appel and Blazy [3], we separate data from control by defining the step relation of generic operational semantics in continuation-passing style (Figure 5).

Step Relation $\langle s, \kappa\rangle \rightsquigarrow \langle s', \kappa'\rangle$

$$\langle s, \mathsf{skip} \cdot \kappa\rangle \rightsquigarrow \langle s, \kappa\rangle$$
$$\langle s, (c_1; c_2) \cdot \kappa\rangle \rightsquigarrow \langle s, c_1 \cdot c_2 \cdot \kappa\rangle$$
$$\langle s, \mathsf{primcom}(u) \cdot \kappa\rangle \rightsquigarrow \langle s', \kappa\rangle \qquad \textit{if } u \in U \textit{ and } s \stackrel{u}{\mapsto} s'$$
$$\langle s, (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \cdot \kappa\rangle \rightsquigarrow \langle s, c_1 \cdot \kappa\rangle \qquad \textit{if } F \vdash e \Downarrow_s \lfloor \mathsf{true} \rfloor$$
$$\langle s, (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \cdot \kappa\rangle \rightsquigarrow \langle s, c_2 \cdot \kappa\rangle \qquad \textit{if } F \vdash e \Downarrow_s \lfloor \mathsf{false} \rfloor$$
$$\langle s, \mathsf{while}\ e\ \mathsf{do}\ c \cdot \kappa\rangle \rightsquigarrow \langle s, (\mathsf{if}\ e\ \mathsf{then}\ (c; \mathsf{while}\ e\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip}) \cdot \kappa\rangle$$

Figure 5: Operational Semantics in the context of $\mathbb{S}$

We write $s \stackrel{u}{\mapsto} s'$ to denote that (s, s') is a state transition permitted by u. That is, $(s, s') \in u$. Note that although we write u in lower-case to distinguish it from U, u is itself a set. A control κ is either Kstop, which indicates the successful end of execution, or $\mathsf{Kseq}(c, \kappa)$ (written $c \cdot \kappa$), which first runs statement c in the current state, then in the resulting state continues execution with control κ.

Configurations $\langle s, \kappa\rangle$ of the operational semantics, which we also call continuations or programs, are tuples of a state and the current control. Note that only primitive commands can modify the data component of a configuration; all other operational rules update just the control. Also, the only way to introduce nondeterminism in this framework is via primitive commands.

As an example of a primitive command, assume states are tuples of a store and a heap, where stores and heaps are just partial functions from program variables (resp. addresses) to values. In this setup, we can define the assign primitive command as $\mathtt{assign}(x,\ e) \stackrel{def}{=} \lambda\sigma.\ \lambda\sigma'.\ \exists b.\ e \Downarrow_\sigma \lfloor b \rfloor \ \wedge\ \sigma' = (\sigma_s[x \leftarrow b],\ \sigma_h)$. That is, $\mathtt{assign}(x, e)$ evaluates expression e in the current state to value b, then updates program variable x to b. σ_s and σ_h project the store and heap components of state σ.

4.1 Safety

We now define a notion of *safety* semantically with respect to generic operational semantics. Safety is used to define the Hoare triple of our separation logic. Intuitively, a program is safe in a generic operational semantics if either the program 1) infinitely loops, or 2) terminates normally, that is, without going wrong in some way. Calcagno et al. model wrong behavior with a distinguished *fault* state. We take a slightly more operational approach in this work and say that a program is safe if no terminating execution of the program gets stuck. Stuckness is defined in the standard way (i.e. a program is stuck if it can't take a step in the operational semantics).

Definition 2 (Immediate Safety)**.** *A program* $\langle s, \kappa\rangle$ *is* immediately safe *iff*

$$\kappa = \mathsf{Kstop}$$

or

$$\exists s'.\ \exists \kappa'.\ \langle s, \kappa\rangle \rightsquigarrow \langle s', \kappa'\rangle.$$

Definition 3 (Safety)**.** *A program* $\langle s, \kappa\rangle$ *is* safe *iff for all* s', κ',

$$\frac{\langle s, \kappa\rangle \rightsquigarrow^* \langle s', \kappa'\rangle}{\text{immediately safe } \langle s', \kappa'\rangle.}$$

This definition corresponds to the informal definition of safety we gave above: if a program $\langle s, \kappa\rangle$ is safe, it may either infinitely loop or terminate safely, but safety ensures it will *never get stuck.*

It follows immediately from the definition of safety that safe configurations can only step to safe configurations.

Lemma 1 (Step Safety)**.** *For all* safe *configurations* $\langle s,\, \kappa\rangle$, $\langle s,\, \kappa\rangle \rightsquigarrow \langle s',\, \kappa'\rangle$ *implies that* $\langle s',\, \kappa'\rangle$ *is* safe. □

The natural converse theorem, that configurations stepping to safe configurations are themselves safe, is true, however, only when the initial configuration is deterministic.

Definition 4 (Deterministic Configuration)**.** *A configuration* $\langle s,\, \kappa\rangle$ *is* deterministic *iff*

$$\langle s,\, \kappa\rangle \rightsquigarrow \langle s',\, \kappa'\rangle \wedge \langle s,\, \kappa\rangle \rightsquigarrow \langle s'',\, \kappa''\rangle \rightarrow s' = s'' \wedge \kappa' = \kappa''.$$

Lemma 2 (Deterministic Step Safety)**.** *For all* safe *configurations* $\langle s',\, \kappa'\rangle$ *and* deterministic *configurations* $\langle s,\, \kappa\rangle$, $\langle s,\, \kappa\rangle \rightsquigarrow \langle s',\, \kappa'\rangle$ *implies that* $\langle s,\, \kappa\rangle$ *is* safe. □

Lemma 2 is useful for showing that control steps, all of which are deterministic, preserve safety.

Finally, we prove that $\rightsquigarrow$ is a compatible relation, in the sense of [5], p. 50. Intuitively, compatibility means that executing a command to completion in one step in the empty continuation is the same as executing that command to completion in any larger continuation. We use compatibility later on to prove a convenient introduction rule in our separation logic for primitive commands.

Lemma 3 (Step Compatibility)**.** $\langle s,\, c \cdot \mathsf{Kstop}\rangle \rightsquigarrow \langle s',\, \mathsf{Kstop}\rangle \rightarrow \langle s,\, c \cdot \kappa\rangle \rightsquigarrow \langle s',\, \kappa\rangle$. □

5. Stratified Semantics

Generic operational semantics are adequate for modeling programs in a single language. However, we're interested in something more: showing that programs in the object (concrete) language refine programs in an abstract language operating on worlds of the program logic, at least up to erasure. It's for this purpose that we introduce the notion of *stratified semantics.*

Informally, one can think of a stratified semantics as the Cartesian product of two generic operational semantics. The construction is as follows: let (W, V, G) and (S, U, F) be generic operational semantics. (S, U, F) defines the object language while (W, V, G) defines a corresponding abstract programming language operating on worlds W. The resulting stratified semantics, to a first approximation, is the structure $(W, S, \varepsilon_{VU}, \varepsilon_{GF})$, where $\varepsilon_{VU} : \mathcal{P}(V \times U)$ and $\varepsilon_{GF} : \mathcal{P}(G \times F)$ are *sets* of command and expression pairs, respectively. The way to think about ε_{VU} and ε_{GF} is as *predicates* defining the primitive commands $v : V$ and $u : U$ that can be run in the same control.

This really is just a first approximation, however. In particular, taking any $\varepsilon_{VU} : \mathcal{P}(V \times U)$ or $\varepsilon_{GF} : \mathcal{P}(G \times F)$ isn't quite right, since a particular set of state transitions $v_{load} : V$ (modeling all possible load operations in the abstract semantics) may not correspond to another set of state transitions $u_{store} : U$ (modeling all possible stores in the concrete semantics). In this case, we wouldn't expect v_{load} to make u_{store} safe, since the resource requirements of loads are clearly different from those of stores.

The solution to this problem is to require that the pairs of relations in ε_{VU} satisfy *erasure* and *safety* properties, as we described in Section 3. The pairs of functions in ε_{GF} must satisfy corresponding *expression erasure* and *expression safety* properties.

But to formalize these properties we first need to equip stratified semantics with a little more structure: an erasure function mapping worlds W to states S. We'll call this function *world erasure* and we'll write $\varepsilon(w) = s$ to denote that world w erases to state s. We are justified in requiring that world erasure be *total* because we expect that worlds contain at least as much structure as states; each world should be able to reconstruct a corresponding state of the concrete semantics. We are justified in requiring that world erasure be *functional* because we don't expect more than one state to correspond to a single world.[2] We formalize these notions here.

Definition 5 (Primitive Command Erasure, $(v, u) : \varepsilon_{VU}$)**.** *$v$ erases to u iff*

$$\frac{\varepsilon(w) = s \quad w \overset{v}{\mapsto} w' \quad s \overset{u}{\mapsto} s'}{\varepsilon(w') = s'.}$$

Primitive command erasure requires that executions of primitive command pairs in stratified semantics preserve world erasure; this is a type of monotonicity property. An important consequence of this property is the fact that primitive commands, and by extension the step relation, of the concrete semantics must be deterministic. This follows from the fact that ε is a function.

Definition 6 (Primitive Command Safety, $(v, u) : \varepsilon_{VU}$)**.** *$v$ makes u safe iff*

$$\frac{\varepsilon(w) = s \quad w \overset{v}{\mapsto} w'}{\exists s'.\ s \overset{u}{\mapsto} s'.}$$

That is, primitive command executions in the abstract semantics imply the existence of corresponding (nonstuck) primitive command executions in the concrete semantics.

Primitive expressions must satisfy corresponding erasure and safety properties.

Definition 7 (Primitive Expression Erasure, $(g, f) : \varepsilon_{GF}$)**.** *Primitive expression g erases to f iff*

$$\frac{\varepsilon(w) = s \quad \mathsf{primexpr}(g) \Downarrow_w \lfloor b_1 \rfloor \quad \mathsf{primexpr}(f) \Downarrow_s \lfloor b_2 \rfloor}{b_1 = b_2}$$

Definition 8 (Primitive Expression Safety, $(g, f) : \varepsilon_{GF}$)**.** *Primitive expression g makes f safe iff*

$$\frac{\mathsf{primexpr}(g) \Downarrow_w \lfloor b_1 \rfloor \quad \varepsilon(w) = s}{\exists b_2.\ \mathsf{primexpr}(f) \Downarrow_s \lfloor b_2 \rfloor}$$

Primitive expression erasure and safety are especially important for preserving control flow during program erasure.

Definition 9 (Stratified Semantics)**.** *The tuple* $\mathbb{WS} = (W, S, \varepsilon_{VU}, \varepsilon_{GF}, \varepsilon)$ *is a* stratified semantics *when*

1. *$\mathbb{W} = (W, V, G)$ is a generic operational semantics;*
2. *$\mathbb{S} = (S, U, F)$ is a generic operational semantics;*
3. *$\varepsilon_{VU} : \mathcal{P}(V \times U)$ gives the pairs of corresponding abstract and concrete primitive commands;*
4. *$\varepsilon_{GF} : \mathcal{P}(G \times F)$ gives the pairs of corresponding abstract and concrete primitive expressions;*
5. *$\varepsilon : W \rightarrow S$ is a world erasure function;*
6. *Pairs $(v, u) : \varepsilon_{VU}$ satisfy* primitive command erasure *and* primitive command safety *with respect to ε; and*
7. *Pairs $(g, f) : \varepsilon_{GF}$ satisfy* primitive expression erasure *and* primitive expression safety *with respect to ε.*

Note that although we use lower-case letters for v and u to distinguish them from the sets V and U, v and u are themselves

[2] In most cases we wouldn't expect world erasure to be injective: many worlds might map to the same state if, for example, worlds tracked infinitely splittable read shares at heap cells while states tracked only a single read permission, as in CompCert. Also, we believe world erasure could be generalized to *relations* on worlds and states, although we have no reason to do so in this paper since our applications are to deterministic languages (e.g. sequential Cminor). Generalizing world erasure to relations would require a full bisimulation proof in Corollary 3 instead of the unidirectional simulation presented there, among other changes.

sets of world and state transition pairs (that is, relations on worlds and states). Also note that because $\varepsilon_{VU} : \mathcal{P}(V \times U)$, $v : V$ and $u : U$.

In the context of a stratified semantics, the following theorems are simple corollaries of primitive command and expression erasure and safety.

Corollary 1 (Primitive Command Erasure and Safety)**.** *Given a stratified semantics* $\mathbb{WS}$ *and a primitive command pair* $(v, u) : \varepsilon_{VU}$*,*

$$\frac{\varepsilon(w) = s \quad w \overset{v}{\mapsto} w'}{\exists s'.\ s \overset{u}{\mapsto} s' \ \wedge\ \varepsilon(w') = s'.}$$

□

Corollary 2 (Primitive Expression Erasure and Safety)**.** *Given a stratified semantics* $\mathbb{WS}$ *and a primitive expression pair* $(g, f) : \varepsilon_{GF}$*,*

$$\frac{\mathsf{primexpr}(g) \Downarrow_w \lfloor b \rfloor \quad \varepsilon(w) = s}{\mathsf{primexpr}(f) \Downarrow_s \lfloor b \rfloor}$$

□

5.1 Program Erasure and Refinement

In this section, we define what it means for an abstract program to correspond to a concrete one by defining an erasure relation on program syntax. Then, we extend the erasure and safety properties satisfied by primitive commands and primitive expressions to the step relations of generic semantics (step erasure and safety). Finally, we prove as a corollary of step erasure and safety that concrete programs simulate corresponding abstract programs.

Let $\mathbb{W}$ and $\mathbb{S}$ be generic operational semantics defining (resp.) abstract and concrete programming languages. Let $\mathbb{WS}$ be a stratified semantics deriving from $\mathbb{W}$ and $\mathbb{S}$, in the sense of Definition 9. Abstract program $\langle w, \kappa_w \rangle$ erases to concrete program $\langle \varepsilon(w), \overline{\kappa_w} \rangle$ under the following conditions.

Definition 10 (Program Erasure)**.** Program erasure *is the least relation satisfying:*

$$\overline{\langle w, \kappa_w \rangle} \quad = \quad \langle \varepsilon(w), \overline{\kappa_w} \rangle.$$

Here, we overload $\bar{\cdot}$ to denote program erasure and control erasure. We define control erasure, in terms of command erasure, as follows:

Definition 11 (Control Erasure)**.** Control erasure *is the least relation satisfying:*

$$\begin{array}{rcl} \overline{\mathsf{Kstop}} & = & \mathsf{Kstop} \\ \overline{c \cdot \kappa} & = & \overline{c} \cdot \overline{\kappa} \end{array}$$

Definition 12 (Command Erasure)**.** Command erasure *is the least relation satisfying:*

$$\overline{\mathsf{primcom}(v)} \quad = \quad \mathsf{primcom}(u) \qquad ((v, u) : \varepsilon_{VU}).$$

We elide the standard structural rules for skip, seq, conditionals, and while. Expression erasure $\overline{e} = e'$ is defined in the obvious way. Primitive expressions erase when $(g, f) : \varepsilon_{GF}$. Other rules are the expected search rules.

Now that we've defined program erasure, we can extend erasure and safety to the step relations of abstract and concrete semantics.

Lemma 4 (Step Erasure)**.**

$$\frac{\overline{\langle w, \kappa_w \rangle} = \langle s, \kappa_s \rangle \quad \langle w, \kappa_w \rangle \rightsquigarrow \langle w', \kappa'_w \rangle \quad \langle s, \kappa_s \rangle \rightsquigarrow \langle s', \kappa'_s \rangle}{\overline{\langle w', \kappa'_w \rangle} \quad = \quad \langle s', \kappa'_s \rangle}$$

Proof. By case analysis on $\rightsquigarrow$, primitive command erasure and expression erasure. □

Lemma 5 (Step Safety)**.**

$$\frac{\overline{\langle w, \kappa_w \rangle} = \langle s, \kappa_s \rangle \qquad \langle w, \kappa_w \rangle \rightsquigarrow \langle w', \kappa'_w \rangle}{\exists s'.\ \exists \kappa'_s.\ \langle s, \kappa_s \rangle \rightsquigarrow \langle s', \kappa'_s \rangle}$$

Proof. By case analysis on $\rightsquigarrow$, primitive command safety and expression safety. □

Step erasure and safety are used in the soundness proof of our Hoare logic rule for primitive commands. The idea is straightforward: because abstract commands imply corresponding concrete commands are safe and because abstract commands commute under erasure with corresponding abstract commands, one can prove the soundness of a Hoare rule for a *concrete* command just by showing that its *abstract* counterpart meets the intended specification.

As a corollary of step erasure and safety, we show that programs in the concrete semantics simulate corresponding abstract ones. The statement of the simulation theorem is most easily given as a commutative diagram.

Corollary 3 (Refinement Simulation)**.** *The following diagram commutes:*

$$\begin{array}{ccc} \langle w, \kappa_w \rangle & \xrightarrow{\rightsquigarrow^*} & \langle w', \kappa'_w \rangle \\ \varepsilon \downarrow & & \downarrow \varepsilon \\ \langle s, \kappa_s \rangle & \overset{\rightsquigarrow^*}{\dashrightarrow} & \langle s', \kappa'_s \rangle \end{array}$$

Proof. By induction on $\rightsquigarrow^*$, and Lemmas 4 and 5. □

Refinement simulation states that for any execution in the abstract semantics, there is an execution in the concrete semantics that commutes with erasure. (In the diagram, ε is overloaded to denote program erasure.) Because the concrete semantics is required to be deterministic by primitive command erasure, this commuting execution is the only such execution.

6. Stratified Semantics with Separation

The stratified semantics we've considered so far were derived from generic semantics $\mathbb{W}$ and $\mathbb{S}$. We said that $\mathbb{S}$ modeled programs of an object language, one we'd like to reason about, while $\mathbb{W}$ modeled programs of an abstract language whose states were also *worlds* of a program logic—more precisely, a separation logic. The product of these two generic semantics was the structure $\mathbb{WS}$, a stratified semantics. Such a structure was useful for proving simulation properties but lacked the scaffolding necessary to construct a separation logic.

In this section, we show how to add *separation algebras* to stratified semantics in order to construct *stratified semantics with separation*. A stratified semantics with separation can be used to mechanically derive a separation logic for $\mathbb{S}$ (§ 7).

Adding Separation to $\mathbb{WS}$**.** We first define the set of *separation algebra elements* by providing a type, A. On top of A we define a partial *join* operator, $\oplus$, satisfying the separation algebra axioms:

$$\begin{array}{l} x \oplus y = z_1 \ \rightarrow\ x \oplus y = z_2 \ \rightarrow\ z_1 = z_2 \\ x_1 \oplus y = z \ \rightarrow\ x_2 \oplus y = z \ \rightarrow\ x_1 = x_2 \\ x \oplus y = z \ \rightarrow\ y \oplus x = z \\ x \oplus y = a \ \rightarrow\ a \oplus z = b \ \rightarrow\ \exists c.\ y \oplus z = c \wedge x \oplus c = b \\ \forall x.\ \exists u.\ u \oplus x = x \end{array}$$

That is, $\oplus$ must be functional, cancellative, commutative, and associative, and every element of A must have a unit u.

$a \vDash \mathsf{emp}$	*iff*	$a \oplus a = a$
$a \vDash \mathsf{true}$	*iff*	*always*
$a \vDash \mathsf{false}$	*iff*	*never*
$a \vDash P * Q$	*iff*	$\exists a_0.\ \exists a_1.$
		$a_0 \oplus a_1 = a \ \wedge\ a_0 \vDash P \ \wedge\ a_1 \vDash Q$
$a_1 \vDash P \mathrel{-\!\!*} Q$	*iff*	$\forall a_0.\ \forall a.$
		$a_0 \oplus a_1 = a \rightarrow a_0 \vDash P \rightarrow a \vDash Q$
$a \vDash P \wedge Q$	*iff*	$a \vDash P$ *and* $a \vDash Q$
$a \vDash P \rightarrow Q$	*iff*	$a \vDash P \rightarrow a \vDash Q$
$a \vDash \exists b.\ P$	*iff*	$\exists b.\ a \vDash P(b)$
$a \vDash \mathsf{expr_eval}(e, b)$	*iff*	$e \Downarrow_a \lfloor b \rfloor$
$a \vDash \mathsf{safe_expr}(e)$	*iff*	$a \vDash \exists b.\ \mathsf{expr_eval}(e, b)$
$a \vDash \mathsf{assert_expr}(e)$	*iff*	$a \vDash \mathsf{expr_eval}(e, \mathsf{true})$

Figure 6: Forcing Semantics of Separation Logic Assertions (excerpted)

Next, we define a *projection* $\alpha : W \rightarrow A$ mapping worlds W to SA elements, and a set of primitive expressions, $H \subseteq \mathcal{P}(A \rightharpoonup \mathsf{bool})$. The primary purpose of H is to support separation logic assertions, such as $\mathsf{assert_expr}(e)$, that deal with expression evaluation. The meaning of the assertion $\mathsf{assert_expr}(e)$ is "in the current world (more precisely, the α-projection of the current world), e is not stuck and evaluates to a true value."

We'd like the e's in such assertions to correspond to expressions actually evaluated in the object language. In the proof of the Hoare rule for conditionals, for example, $\mathsf{assert_expr}(e)$ (and symmetrically, $\mathsf{assert_expr}(\neg e)$) must imply that evaluating some expression e' in the object language program in a corresponding control is safe, and produces the left (or right) branch of the conditional. For this purpose, we introduce a predicate $\varepsilon_{HG} : \mathcal{P}(H \times G)$ very similar to the ε_{GF} predicate of stratified semantics that determines whether two primitive expressions h and g can be evaluated in the same control. Pairs $(h, g) : \varepsilon_{HG}$ must satisfy the same primitive expression erasure and safety conditions satisfied by pairs $(g, f) : \varepsilon_{GF}$. We define the transitive closure of ε_{HG} and ε_{GF} (ε_{HF}) in the standard way.

The final, somewhat unwieldy structure is: $\mathbb{WSA} = (W, S, A, \varepsilon_{VU}, \varepsilon_{GF}, \varepsilon_{HG}, \varepsilon, \alpha, \oplus)$.

Definition 13 (Stratified Semantics with Separation)**.** *A* stratified semantics with separation *is a tuple* $\mathbb{WSA} = (W, S, A, \varepsilon_{VU}, \varepsilon_{GF}, \varepsilon_{HG}, \varepsilon, \alpha, \oplus)$ *s.t.* $(W, S, \varepsilon_{VU}, \varepsilon_{GF}, \varepsilon)$ *is a stratified semantics,* $(A, \oplus)$ *is a separation algebra,* $\alpha : W \rightarrow A$ *is a projection mapping worlds to SA elements, and* $\varepsilon_{HG} : \mathcal{P}(H \times G)$ *is a predicate defining the set of corresponding primitive expression pairs* (h, g)*.* $H \subseteq \mathcal{P}(A \rightharpoonup \mathsf{bool})$*.*

Now that we've equipped stratified semantics with a separation algebra, we can give the forcing semantics of some common separation logic assertions (Figure 6). $\mathsf{expr_eval}(e, b)$ asserts that in the α-projection of the current world, expression e evaluates to boolean value b. $\mathsf{safe_expr}(e)$ asserts that evaluation of e in the α-projection of the current world will not get stuck. The b in the assertion $\exists b.\ P$ quantifies over any Coq type, not just values. The other assertions are standard.

7. A Separation Logic for $\mathbb{S}$

We now have all the machinery we need to construct a separation logic for $\mathbb{S}$. The construction is a *semantic* one: the Hoare triple $\vdash \{P\}\ c\ \{Q\}$ is interpreted semantically as a proposition of higher-order logic. Inference rules of the form

$$\frac{\vdash \{P_i\}\ c_i\ \{Q_i\}}{\vdash \{P\}\ c\ \{Q\}}$$

are then proved as derived rules from this interpretation.

We follow Appel and Blazy [3] in giving the interpretation of the Hoare triple in continuation-passing style. In outline, we first define what it means for a separation logic predicate P to guard a control; then, we give the semantics of the Hoare triple $\vdash \{P\}\ c\ \{Q\}$ in terms of guards. Finally, we prove the soundness of some common inference rules of separation logic with respect to this interpretation.

7.1 Guards

Definition 14. *A predicate* P guards *concrete control* κ *iff*

$$\forall w.\ \alpha(w) \vDash P \rightarrow \text{safe}\ \langle \varepsilon(w),\ \kappa \rangle.$$

$\alpha(w)$ is the α-projection of abstract world w; hence the forcing assertion $\alpha(w) \vDash P$ is well-defined. Remember that $\varepsilon(w)$ erases world w to a state $s : S$.

7.2 The Hoare Triple $\vdash \{P\}\ c\ \{Q\}$

We give a semantic interpretation of $\vdash \{P\}\ c\ \{Q\}$ in terms of guards:

Definition 15. $\vdash \{P\}\ c\ \{Q\}$ *iff*

$$\forall \kappa.\ \forall F.\ (Q * F) \text{ guards } \kappa \rightarrow (P * F) \text{ guards } (c \cdot \kappa).$$

Unpacking the definitions, it should be clear that this interpretation of the Hoare triple is satisfied by nonterminating executions of c for any preconditions P and postconditions Q. Proof: nonterminating executions are safe, thus $c \cdot \kappa$ is safe, for all κ.

What should also be clear is that because the predicate F is universally quantified in the definition of the triple, and because we require that F be separating conjoined with the pre- and postconditions in the guards, the frame rule is sound for every command with a valid Hoare triple in the logic:

Theorem 1 (Soundness of the Frame Rule)**.** *For all commands* c *and frames* F*,*

$$\frac{\vdash \{P\}\ c\ \{Q\}}{\vdash \{P * F\}\ c\ \{Q * F\}.}$$

Proof. By associativity of $*$ in the guards predicates. □

One implication of this theorem is that primitive commands $\mathsf{primcom}(u)$ with a valid axiomatic semantics satisfy the frame rule. That is, if we can prove $\vdash \{P\}\ \mathsf{primcom}(u)\ \{Q\}$ for some P and Q, then it immediately follows that $\vdash \{P{*}F\}\ \mathsf{primcom}(u)\ \{Q{*}F\}$ by the soundness of the frame rule. There is no trickery here: the proof that $\mathsf{primcom}(u)$ is local is just built in to the soundness proof of $\vdash \{P\}\ \mathsf{primcom}(u)\ \{Q\}$.

Since the frame rule is sound for $\mathsf{primcom}(u)$, we are justified in calling $\mathsf{primcom}(u)$ a local action. When all the primitive commands $u : U$ of a stratified semantics with separation are local in this way, we say that the semantics is a stratified local action semantics.

Definition 16. *A* stratified local action semantics *is a stratified semantics with separation* $\mathbb{WSA}$ *deriving from* $\mathbb{W}$ *and* $\mathbb{S}$ *s.t. all primitive commands* $u : U$ *are local actions.*

7.3 Functional Correctness Specifications

The semantic interpretation we gave the Hoare triple $\vdash \{P\}\ c\ \{Q\}$ in Section 7.2 was for safety and not functional correctness. That

is, $\vdash \{P\}\ c\ \{Q\}$ did not imply that Q held after c was executed in states satisfying P. It just said that c was safe in any state satisfying P. This contravenes the usual functional correctness interpretation of Hoare triples in separation logic. Appel and Blazy hinted at how functional correctness properties could be proved in their separation logic for Cminor, but didn't make the connection between proofs of safety and proofs of full functional correctness explicit.

We demonstrate here that safety is in fact enough to achieve functional correctness. First, we take a slight detour to show that the primitive command mechanism of our framework is powerful enough to support semantic, even noncomputable assertions in the syntax of programs.

Primitive Assertions. Although the syntax of our while-language is limited, the primitive command mechanism—which makes it possible to inject arbitrary relations into the syntax of programs—is quite powerful. For example, we can easily define a semantic assertion statement for any given model of the framework. A semantic assertion is like a C `assert` statement except that the assertion language is the language of propositions of the metalogic instead of C expressions; and instead of producing a runtime error when an asserted expression evaluates to 0, as in C, semantic assertions get stuck when they are not satisfied by the state in which they are asserted.

The construction is as follows: let $\mathbb{WSA}$ be a stratified semantics with separation deriving from $\mathbb{W}$ and $\mathbb{S}$. Define $\mathsf{assert}(P)$ in the abstract semantics as $\mathsf{primcom}(v)$ and $\mathsf{assert}(P)$ in the concrete semantics as $\mathsf{primcom}(u)$ s.t.

$$v \stackrel{def}{=} \lambda w.\ \lambda w'.\ \alpha(w) \vDash P\ \wedge\ w = w',\ \text{and}$$
$$u \stackrel{def}{=} \lambda s.\ \lambda s'.\ s = s'.$$

It should be clear from these definitions that v will get stuck in any world w whose α-projection doesn't satisfy P, and that u is equivalent to skip. It should also be clear that because P is a proposition of the metalogic (i.e. Coq), the command v need not be computable. Fortunately, we can always erase v to u.

Theorem 2. *v erases to u and makes u safe. Furthermore, the construction given here generalizes to any stratified semantics with separation.*

Proof. Erasure: We assume world erasure holds for initial world w and initial state s. Therefore, erasure must hold for w' and s' since $w = w'$ and $s = s'$. Safety: skip is never stuck. Generality: We make no assumptions about $\mathbb{WSA}$. □

Such assertions are sufficient for proving functional correctness properties: the idea is simply to sequentially compose assertions with commands. For example, if $\vdash \{P\}\ c\ \{Q\}$, then we can prove that executing c in states satisfying P does actually result in states satisfying Q (assuming c terminates) by proving the slightly more involved triple $\vdash \{P\}\ c;\ \mathsf{assert}(Q)\ \{Q\}$. If c results in a state that *does not* satisfy Q, $\mathsf{assert}(Q)$ will not be safe and the extended Hoare triple will be unsound. If the Hoare triple *is* sound, then we know that the proposition Q holds immediately after c is executed (again assuming c terminates).

8. Inference Rules

We could derive Hoare rules $\vdash \{P\}\ \mathsf{primcom}(u)\ \{Q\}$ for primitive commands u directly from the definition of the Hoare triple. This would involve unpacking the definitions of the Hoare triple and guards and then proving that $\alpha(w) \vDash P$ implies $\mathsf{primcom}(u)$ can step from s, where $\varepsilon(w) = s$, to some state s' such that $\varepsilon(w') = s'$ and $\alpha(w') \vDash Q$, for some w'. Such proofs would be tedious.

We have a more direct way of deriving $\vdash \{P\}\ c\ \{Q\}$, however: exhibiting a primitive command v in the abstract semantics with a parallel specification. Indeed, this is one of the primary reasons we constructed an abstract semantics in the first place.

For this purpose, we define a new version of the Hoare triple, called the *primitive Hoare triple*. This Hoare triple is in direct style and is used only to derive standard Hoare triples for primitive commands. We give its definition here.

Definition 17 (Primitive Hoare Triple)**.** $\vdash_{prim}\ \{P\}\ v\ \{Q\}$ *iff* $\forall w.\ \forall F.$

$$\frac{\alpha(w) \vDash P * F}{\exists w'.\ w \stackrel{v}{\mapsto} w'\ \wedge\ \alpha(w') \vDash Q * F.}$$

With this definition in hand, we can state the introduction rule for primitive commands quite directly.

Theorem 3 (Primitive Command Introduction ($Primcom$))**.**

$$\frac{(v, u) : \varepsilon_{VU} \qquad \vdash_{prim} \{P\}\ v\ \{Q\}}{\vdash \{P\}\ \mathsf{primcom}(u)\ \{Q\}}$$

Proof. By step erasure, step safety, and step compatibility. □

Figures 7 and 8 present the inference rules (excerpted) of the separation logic for $\mathbb{S}$, including the introduction rule for primitive commands. All of the rules in the figures except the rule for while-loops are proved sound in Coq w.r.t. the semantic interpretation of the Hoare judgment given in § 7. We have proved the Hoare rule for while-loops sound informally and do not expect the formal proof will cause much trouble since we have already formally proved a more general rule for loops with arbitrary block nesting and breaks in the context of the larger program logic for Cminor. Rules in Figure 7 are proved directly from the definition of the Hoare triple. Rules in Figure 8 are proved as derived lemmas from the rules in Figure 7. In the proof rules prefixed with P (e.g. $PConj_1$), $\lceil \cdot \rceil$ denotes pure predicates, that is, propositions lifted into the domain of predicates via the lifting function $\lambda A.\ \lambda a.\ A$.

Most of the inference rules in the figures are straightforward. In the rules for while loops and conditionals, $\mathrm{intuitionistic}\ e$ means evaluation of expression e is invariant under extensions of the state. $\mathrm{intuitionistic}\ e$ holds for the various expression operators, e.g. $\neg$, $\&\&$, $||$, if their operands are intuitionistic but must be proven for each primitive expression. $\overline{e} = e'$ denotes the transitive closure of expression erasure under ε_{HF}.

9. A Concrete Model

In this section we construct a model of the framework for a simple programming language with external function calls, and use this model to build the core of a variables-as-resources [19] separation logic. A variables-as-resources separation logic is one in which program variables are modeled as separating resources, much like heap cells in a conventional separation logic. The primary advantage of a variables-as-resources logic is that it permits the concise statement of inference rules without side conditions concerning program variables. We model the following primitive commands: `assign`, `load`, `store`, and $\texttt{ext_fun}_R$, and one primitive expression: `evar`, the purpose of which is to look up variables in the local environment. Although this language is extremely simple, its formulation touches on every aspect of the framework.

9.1 Concrete States Σ

In order to express the operational behavior of the primitive commands and of `evar`, we first need to construct a concrete memory model Σ. The basic types of the concrete memory model are:

$$\mathsf{val} \stackrel{def}{=} \mathsf{bool}$$
$$\mathsf{var},\ \mathsf{address} \stackrel{def}{=} \mathbb{N}.$$

$$\frac{(v,u) : \varepsilon_{VU} \quad \vdash_{prim} \{P\}\ v\ \{Q\}}{\vdash \{P\}\ \mathsf{primcom}(u)\ \{Q\}}\ \mathit{Primcom} \qquad \frac{}{\vdash \{P\}\ \mathsf{skip}\ \{P\}}\ \mathit{Skip} \qquad \frac{\vdash \{P\}\ c_1\ \{P'\} \quad \vdash \{P'\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1;\ c_2\ \{Q\}}\ \mathit{Seq}$$

$$\frac{\vdash \{P \wedge \mathsf{assert_expr}(e)\}\ c_1\ \{Q\} \quad P \Rightarrow \mathsf{safe_expr}(e) \quad \overline{e} = e' \quad \text{intuitionistic } e \quad \vdash \{P \wedge \mathsf{assert_expr}(\neg e)\}\ c_2\ \{Q\}}{\vdash \{P\}\ \mathsf{if}\ e'\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \{Q\}}\ \mathit{Conditional}$$

$$\frac{\vdash \{P \wedge \mathsf{assert_expr}(e)\}\ c\ \{P\} \qquad P \Rightarrow \mathsf{safe_expr}(e) \qquad \overline{e} = e' \qquad \text{intuitionistic } e}{\vdash \{P\}\ \mathsf{while}\ e'\ \mathsf{do}\ c\ \{P \wedge \mathsf{assert_expr}(\neg e)\}}\ \mathit{While}$$

$$\frac{P \Rightarrow P' \quad \vdash \{P'\}\ c\ \{Q'\} \quad Q' \Rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}\ \mathit{Consequence} \qquad \frac{\vdash \{P\}\ c\ \{Q\}}{\vdash \{P * F\}\ c\ \{Q * F\}}\ \mathit{Frame} \qquad \frac{\vdash \{P_1\}\ c\ \{Q\} \quad \vdash \{P_2\}\ c\ \{Q\}}{\vdash \{P_1 \vee P_2\}\ c\ \{Q\}}\ \mathit{Disj}$$

$$\frac{\vdash \{P\}\ c\ \{Q\}}{\vdash \{\lceil A \rceil \wedge P\}\ c\ \{\lceil A \rceil \wedge Q\}}\ \mathit{PConj_1}$$

Figure 7: Core Inference Rules in the context of $\mathbb{WSA}$ (excerpted)

$$\frac{\vdash \{\lceil A \rceil \wedge P\}\ c\ \{\lceil A \rceil \wedge Q\} \qquad A}{\vdash \{P\}\ c\ \{Q\}}\ \mathit{PConj_2} \qquad \frac{\vdash \{P_1\}\ c_1\ \{Q_1\} \qquad \vdash \{P_2\}\ c_2\ \{Q_2\}}{\vdash \{P_1 * P_2\}\ c_1;\ c_2\ \{Q_1 * Q_2\}}\ \mathit{Compose}$$

$$\frac{\vdash \{P\}\ c\ \{Q\} \qquad Q \Rightarrow A}{\vdash \{A \wedge P\}\ c\ \{A \wedge Q\}}\ \mathit{Conj}$$

Figure 8: Derived Inference Rules in the context of $\mathbb{WSA}$ (excerpted)

$$\begin{aligned}
\mathtt{evar}(x) &\overset{def}{=} \lambda\sigma.\ \sigma_s(x) \\
\mathtt{assign}(x,\ e) &\overset{def}{=} \lambda\sigma.\ \lambda\sigma'.\ \exists b.\ e \Downarrow_\sigma \lfloor b \rfloor\ \wedge\ \sigma' = (\sigma_\Omega,\ \sigma_s[x \leftarrow b],\ \sigma_h) \\
\mathtt{load}(x,\ l) &\overset{def}{=} \lambda\sigma.\ \lambda\sigma'.\ \exists b.\ \sigma_h(l) = \lfloor b \rfloor\ \wedge\ \sigma' = (\sigma_\Omega,\ \sigma_s[x \leftarrow b],\ \sigma_h) \\
\mathtt{store}(l,\ x) &\overset{def}{=} \lambda\sigma.\ \lambda\sigma'.\ \exists b.\ \exists b'.\ \sigma_s(x) = \lfloor b' \rfloor\ \wedge\ \sigma_h(l) = \lfloor b \rfloor\ \wedge\ \sigma' = (\sigma_\Omega,\ \sigma_s,\ \sigma_h[l \leftarrow b']) \\
\mathtt{ext_fun}_R &\overset{def}{=} \lambda\sigma.\ \lambda\sigma'.\ \sigma \overset{R}{\mapsto} \sigma'
\end{aligned}$$

Figure 9: Concrete Primitive Commands and evar

$$\begin{aligned}
\hat{\mathtt{evar}}(x) &\overset{def}{=} \lambda\hat{\sigma}.\ \mathsf{case}\ \hat{\sigma}_s(x)\ \mathsf{of}\ \lfloor (b,\ \pi) \rfloor \Rightarrow \lfloor b \rfloor \mid \mathsf{none} \Rightarrow \mathsf{none} \\
\hat{\mathtt{assign}}(x,\ e) &\overset{def}{=} \lambda\hat{\sigma}.\ \lambda\hat{\sigma}'.\ \exists b.\ \exists b'.\ e \Downarrow_{\hat{\sigma}} \lfloor b' \rfloor\ \wedge\ \hat{\sigma}_s(x) = \lfloor (b, \top) \rfloor\ \wedge\ \hat{\sigma}' = (\sigma_\Omega,\ \hat{\sigma}_s[x \leftarrow (b',\ \top)],\ \hat{\sigma}_h) \\
\hat{\mathtt{load}}(x,\ l) &\overset{def}{=} \lambda\hat{\sigma}.\ \lambda\hat{\sigma}'.\ \exists b.\ \exists b'.\ \exists \pi.\ \hat{\sigma}_s(x) = \lfloor (b,\ \top) \rfloor\ \wedge\ \hat{\sigma}_h(l) = \lfloor (b',\ \pi) \rfloor\ \wedge\ \hat{\sigma}' = (\sigma_\Omega, \hat{\sigma}_s[x \leftarrow (b',\ \top)],\ \hat{\sigma}_h) \\
\hat{\mathtt{store}}(l,\ x) &\overset{def}{=} \lambda\hat{\sigma}.\ \lambda\hat{\sigma}'.\ \exists b.\ \exists b'.\ \exists \pi.\ \hat{\sigma}_s(x) = \lfloor (b',\ \pi) \rfloor\ \wedge\ \hat{\sigma}_h(l) = \lfloor (b,\ \top) \rfloor\ \wedge\ \hat{\sigma}' = (\sigma_\Omega, \hat{\sigma}_s,\ \hat{\sigma}_h[l \leftarrow (b',\ \top)]) \\
\hat{\mathtt{ext_fun}}_{\hat{R}} &\overset{def}{=} \lambda\hat{\sigma}.\ \lambda\hat{\sigma}'.\ \hat{\sigma} \overset{\hat{R}}{\mapsto} \hat{\sigma}'
\end{aligned}$$

Figure 10: Abstract Primitive Commands and $\hat{\mathtt{evar}}$

$$\begin{aligned}
h \vDash l \overset{\pi}{\mapsto} b &\overset{def}{=} \forall l'.\ \mathsf{if}\ l = l'\ \mathsf{then}\ h(l') = \lfloor (b, \pi) \rfloor\ \mathsf{else}\ h(l') = \mathsf{none} \\
s \vDash x \overset{\pi}{\mapsto} b &\overset{def}{=} \forall x'.\ \mathsf{if}\ x = x'\ \mathsf{then}\ s(x') = \lfloor (b, \pi) \rfloor\ \mathsf{else}\ s(x') = \mathsf{none} \\
s \vDash \mathsf{own}_\pi(x) &\overset{def}{=} \exists b.\ s \vDash x \overset{\pi}{\mapsto} b \\
s \vDash \mathsf{own}(x) &\overset{def}{=} s \vDash \mathsf{own}_\top(x) \\
a \vDash {\uparrow^s} P &\overset{def}{=} a_s \vDash P\ \wedge\ a_h \vDash \mathsf{emp} \\
a \vDash {\uparrow^h} P &\overset{def}{=} a_s \vDash \mathsf{emp}\ \wedge\ a_h \vDash P
\end{aligned}$$

$$\vdash \{{\uparrow^s}\mathsf{own}(x) \wedge \mathsf{expr_eval}(e, b) \wedge \lceil \overline{e} = e' \rceil \wedge \lceil \text{intuitionistic } e \rceil\}\ \mathsf{primcom}(\mathtt{assign}(x, e'))\ \{{\uparrow^s} x \overset{\top}{\mapsto} b\}$$

$$\vdash \{{\uparrow^s}\mathsf{own}(x)\ *\ {\uparrow^h} l \overset{\pi}{\mapsto} b\}\ \mathsf{primcom}(\mathtt{load}(x, l))\ \{{\uparrow^s} x \overset{\top}{\mapsto} b\ *\ {\uparrow^h} l \overset{\pi}{\mapsto} b\}$$

$$\vdash \{{\uparrow^s} x \overset{\pi}{\mapsto} b'\ *\ {\uparrow^h} l \overset{\top}{\mapsto} b\}\ \mathsf{primcom}(\mathtt{store}(l, x))\ \{{\uparrow^s} x \overset{\pi}{\mapsto} b'\ *\ {\uparrow^h} l \overset{\top}{\mapsto} b'\}$$

$$\vdash \{P_{ext}\}\ \mathsf{primcom}(\mathtt{ext_fun}_R)\ \{Q_{ext}\}$$

Figure 11: Forcing Semantics of Assertions, and Hoare Rules for the Primitive Commands in § 9

To keep the model simple, we assume boolean values and define local variables and addresses as natural numbers.

Concrete Stores. Now, we can define stores simply as partial maps from variables to values:

$$\mathsf{store} \stackrel{def}{=} \mathsf{var} \rightharpoonup \mathsf{val}.$$

Note that in this store model, we don't include any *shares*—which could be used to model the level of access to local variables—even though the assertions of our variables-as-resources separation logic will need to be able to express ownership of local variables. This is because our concrete memory model represents states of the *erased* semantics. Storing shares in the *abstract* store will be sufficient for proving the soundness of a Hoare rule for assignment in a variables-as-resources style, i.e. without side conditions.

Heaps. Heaps in this simple model are just the same as stores, except that we map addresses—instead of identifiers—to values:

$$\mathsf{heap} \stackrel{def}{=} \mathsf{address} \rightharpoonup \mathsf{val}.$$

Concrete States. Concrete states are tuples of a heap, a store, and special component that we call the *oracle state* and denote Ω. The oracle state represents the state of the external world, or the context in which concrete programs execute. Concrete programs can compute with the oracle state through a special primitive command we call $\mathtt{ext_fun}_R$, or external function call. The semantics of $\mathtt{ext_fun}_R$ is given by a relation R on concrete states. Although we leave $\mathtt{ext_fun}_R$ unspecified, the external function call mechanism could be used to implement, e.g., the system calls of an operating system, or preemptive context switching in a thread library.

$$\Sigma \stackrel{def}{=} \Omega \times \mathsf{store} \times \mathsf{heap}.$$

We equip our model with projections σ_Ω, σ_s, and σ_h for accessing the oracle state, store, and heap components of concrete state σ.

Concrete Primitive Commands. With these preliminary definitions out of the way, we are ready to give the definitions of the primitive commands (Figure 9). Read $\sigma_s[x \leftarrow 3]$ as "set the value of variable x in store σ_s to 3". We use the same notation for heaps.

$\mathtt{assign}(x, e)$ evaluates e in the current state and updates x to the resulting value. $\mathtt{load}(x, l)$ sets x to the current value at location l. $\mathtt{store}(l, x)$ stores the value of x at location l. $\mathtt{ext_fun}_R$ passes control to the program context to execute a state transition specified by the relation R. $\mathtt{ext_fun}_R$ takes no arguments; instead we assume that the client and context have agreed on an ad hoc calling convention in which the external function identifier, the arguments to the external function, and its result are passed somewhere in client-modifiable state, such as the store.

Note that $\mathtt{assign}(x, e)$ will succeed only if it's safe to evaluate e in the current state; $\mathtt{load}(x, l)$ will succeed only if the heap is defined at l; and $\mathtt{store}(l, x)$ will succeed only if x is defined and the heap is defined at l. $\mathtt{ext_fun}_R$ will succeed whenever R is defined in the state in which the external function is called.

We construct a generic semantics using these four commands.

Theorem 4 (Concrete Semantics)**.** *Let* $U = \{\mathtt{assign}, \mathtt{load}, \mathtt{store}, \mathtt{ext_fun}_R\}$ *and* $F = \{\mathtt{evar}\}$*.* (Σ, U, F) *is a generic operational semantics.* □

9.2 Abstract Worlds $\hat{\Sigma}$

Next, we construct an abstract memory model $\hat{\Sigma}$, redefine the primitive commands and expression $\mathtt{evar}$ in this model, and show that the abstract commands and $\mathtt{evar}$ correspond to their concrete counterparts.

Abstract Stores, Heaps, and Worlds. We let abstract stores and heaps be partial maps from local variables (resp. locations) to $\mathsf{val} \times \mathsf{pshare}$:

$$\hat{\mathsf{store}} \stackrel{def}{=} \mathsf{var} \rightharpoonup \mathsf{val} \times \mathsf{pshare};$$
$$\hat{\mathsf{heap}} \stackrel{def}{=} \mathsf{address} \rightharpoonup \mathsf{val} \times \mathsf{pshare}.$$

Positive shares π : pshare range over nonempty shares; pshare stands for "positive share."

Shares represent the amount of access to a location in the heap, or to a variable in the store in a variables-as-resources separation logic, and are convenient for expressing sharing patterns among threads, especially in proofs of concurrent programs. We require shares at defined locations and in defined local variables to be nonempty as a well-formedness condition. In the formalization, we use the share model of Dockins et al.

Worlds $\hat{\sigma} \in \hat{\Sigma}$ are just the products of oracle states, abstract stores, and abstract heaps.

$$\hat{\Sigma} \stackrel{def}{=} \Omega \times \hat{\mathsf{store}} \times \hat{\mathsf{heap}}.$$

Note that we use the same Ω in both the concrete and abstract semantics. This is for convenience only; we could assume two different types of oracle state and an erasure function relating them but that would complicate the example presented here.

Figure 10 presents the abstract primitive commands and expression $\hat{\mathtt{evar}}$. These commands, like their concrete counterparts, form a generic semantics.

Theorem 5 (Abstract Semantics)**.** *Let* $V = \{\hat{\mathtt{assign}}, \hat{\mathtt{load}}, \hat{\mathtt{store}}, \hat{\mathtt{ext_fun}}_{\hat{R}}\}$ *and* $G = \{\hat{\mathtt{evar}}\}$*.* $(\hat{\Sigma}, V, G)$ *is a generic operational semantics.* □

9.3 A Stratified Semantics

Constructing a stratified semantics for (Σ, U, F) and $(\hat{\Sigma}, V, G)$ is straightforward. We first define an erasure function $\varepsilon : \hat{\Sigma} \to \Sigma$ mapping worlds to states:

$$\varepsilon(s) \stackrel{def}{=} \lambda x.\ \mathsf{case}\ s(x)\ \mathsf{of}\ \lfloor(b,\ \pi)\rfloor \Rightarrow \lfloor b\rfloor \mid \mathsf{none} \Rightarrow \mathsf{none}$$
$$\varepsilon(h) \stackrel{def}{=} \lambda l.\ \mathsf{case}\ h(l)\ \mathsf{of}\ \lfloor(b,\ \pi)\rfloor \Rightarrow \lfloor b\rfloor \mid \mathsf{none} \Rightarrow \mathsf{none}$$
$$\varepsilon(\hat{\sigma}) \stackrel{def}{=} (\hat{\sigma}_\Omega,\ \varepsilon(\hat{\sigma}_s),\ \varepsilon(\hat{\sigma}_h))$$

ε is overloaded to operate on abstract stores and heaps as well as worlds.

Next, we show that commands and expressions of the abstract semantics satisfy erasure and safety when paired with corresponding commands of the concrete semantics. The predicates ε_{VU} and ε_{GF} define when commands and expressions of the two semantics correspond. For example, $\forall x.\ \forall e.\ \forall e'.$

$$\overline{e} = e' \to (\hat{\mathtt{assign}}(x,\ e), \mathtt{assign}(x,\ e')) : \varepsilon_{VU}$$

gives the ε_{VU} rule for assignment. ε_{VU} is defined inductively with similar rules for loads and stores. The same is done for ε_{GF} (with only one constructor). Erasure and safety follow, for each rule of ε_{VU} and ε_{GF} except the one for external function calls, from the definitions of the abstract and concrete primitive commands and expressions. In the case of external function calls, we assume that $\hat{R}$ and R satisfy erasure and safety. This assumption is realistic: it merely reflects the fact that $\hat{R}$ and R give consistent (abstract and concrete) definitions of the environment.

Theorem 6. $(\hat{\Sigma}, \Sigma, \varepsilon_{VU}, \varepsilon_{GF}, \varepsilon)$ *is a stratified semantics.* □

9.4 Deriving a Separation Logic for (Σ, U, F)

All that's left to do now is to construct a separation algebra $(A,\ \oplus)$, define a projection mapping worlds $\hat{\Sigma}$ to elements of A, and con-

struct the set of primitive expressions H evaluated in A that correspond to the expressions in G. We use these three components to construct the structure $(\hat{\Sigma}, \Sigma, A, \varepsilon_{VU}, \varepsilon_{GF}, \varepsilon_{HG}, \varepsilon, \alpha, \oplus)$, a stratified semantics with separation, and then derive a separation logic for (Σ, U, F).

Let $A = \hat{\text{store}} \times \hat{\text{heap}}$. Let $\alpha : \hat{\Sigma} \to A$ be the projection $\lambda\hat{\sigma}.\ \text{case}\ \hat{\sigma}\ \text{of}\ (_, s, h) \Rightarrow (s, h)$. The oracle state is intended to contain private implementation data like a thread schedule; we don't include oracle states in worlds of the program logic because the user of the logic should not be able to define separation logic predicates that refer to this private state. We define a new expression $\texttt{evar}_A$, evaluated in A and corresponding to $\hat{\texttt{evar}}$, and define ε_{HG} in the obvious way (with $H = \{\texttt{evar}_A\}$). Because the projection α preserves the store component of worlds, expressions in H and G satisfy primitive expression erasure and safety.

We define a separation algebra for A using the SA operators of Dockins et al. Unpacking its definition, we know that $A = (\text{var} \rightharpoonup \text{val} \times \text{pshare}) \times (\text{address} \rightharpoonup \text{val} \times \text{pshare})$. Therefore, our goal is to construct a separation algebra that matches this type. To do so, we first need to construct a separation algebra for the type of data, $\text{option}\ (\text{val} \times \text{pshare})$. In the Coq formalization, we use the SA bijection operator to construct a separation algebra for this type via a bijection with $\text{option}\ (\text{val} \times \text{share})_+$. The subscript $+$ notation here indicates that we're taking the subset of positive (i.e. nonidentity) elements of $\text{val} \times \text{share}$. none then becomes our distinguished unit. Once we've constructed a separation algebra for the type of data, we can lift this SA to an SA on functions via the usual extensional lifting. That is, two functions f and g join whenever $f(l)$ and $g(l)$ join for every l in the domain of f and g.

With these constructions out of the way, we can now define a stratified semantics with separation.

Theorem 7. $(\hat{\Sigma}, \Sigma, A, \varepsilon_{VU}, \varepsilon_{GF}, \varepsilon_{HG}, \varepsilon, \alpha, \oplus)$ *is a stratified semantics with separation.* □

We give Hoare rules for `assign`, `load`, and `store` in Figure 11. Since the external function call primitive $\texttt{ext_fun}_R$ is uninterpreted, we just assume it has a valid specification $\vdash \{P_{ext}\}$ $\texttt{ext_fun}_R\ \{Q_{ext}\}$ in the separation logic. Here, P_{ext} gives the precondition for making an external function call and Q_{ext} gives the external function call postcondition. To prove that an *interpreted* version of $\texttt{ext_fun}_R$ satisfies such a specification, one would have to show, e.g., that $\vdash_{prim} \{P_{ext}\}\ R\ \{Q_{ext}\}$. The framework gives us the Hoare rules in figures 7 and 8 for free, as a consequence of the fact that $(\hat{\Sigma}, \Sigma, A, \varepsilon_{VU}, \varepsilon_{GF}, \varepsilon_{HG}, \varepsilon, \alpha, \oplus)$ is a stratified semantics with separation. In the figure, $\top$ denotes the full share and models exclusive access situations. $\uparrow^s$ and $\uparrow^h$ lift separation logic predicates on stores and heaps (which are well-defined because we defined separation algebras for stores and heaps in the process of defining a separation algebra for A) to operate on elements $a \in A$. We overload the world projections defined above to operate on a's.

Every primitive command in U has a specification in the separation logic. Therefore:

Theorem 8. $(\hat{\Sigma}, \Sigma, A, \varepsilon_{VU}, \varepsilon_{GF}, \varepsilon_{HG}, \varepsilon, \alpha, \oplus)$ *is a stratified local action semantics.* □

10. Related Work

Parkinson [18] makes the case for a *core separation logic* in which it is possible to define new axiomatic semantics for new language features without recourse to meta-theoretic arguments. He suggests that deny-guarantee logics [11], which provide mechanisms for dealing with arbitrary patterns of interference in a clean way, might provide the right foundation for such an approach. Although we agree in sentiment, we take a different approach in this work by facilitating the construction of new separation logics from retargetable components such as command specifications given under an abstract model of program state. One of our primary goals was to prove our separation logics sound with respect to undecorated operational semantics; Parkinson does not address this issue. Because we never resort to proof-theoretic arguments in the soundness proof of our general logic, our logic retains a measure of extensibility.

Dinsdale-Young et al. [8] relate abstract module specifications to concrete implementations of those modules. Unlike our own work, which is more operational in nature, the translations of Dinsdale-Young et al. are performed on *axiomatic* descriptions of the source and target languages expressed in terms of *context algebras*. Context algebras generalize separation algebras by defining a set of *state contexts* and two partial noncommutative binary operators: a *context composition* operator that inserts a context into a one-hole context; and an *application* operator that inserts a state into a one-hole context. They describe both *locality-preserving* and *locality-breaking* module translations in this framework, the intuition being that locality-preserving translations faithfully translate footprints of an abstract module specification to footprints of a concrete implementation while locality-breaking translations do not. In locality-preserving translations, frame rules that hold in the abstract context are also sound in the implementation context. It could be interesting in future work to experiment with context algebras in the stratified framework we present here. In principle, context algebras could be substituted relatively easily for the separation algebras of stratified semantics with separation.

11. Application to CompCert

Appel, Hobor, and Zappa Nardelli [12] showed how a soundness proof for a Concurrent Separation Logic for Cminor could be decomposed into (1) soundness of a sequential separation logic for a decorated operational semantics with (2) soundness of a concurrency oracle. By 2009 the first of these two proofs was complete and machine-checked in Coq, and the second part was nearly complete. But the operational semantics of the "real" CompCert Cminor is undecorated, not decorated. The present work closes the gap: we have refactored the soundness proof for sequential separation logic according to the recipe given here. The refactored proof is more than 98% complete (measured by the number of admitted lemmas in the approximately 40,000 line proof); we don't foresee any problems completing the final $< 2\%$ of the proof.

Acknowledgments

We would like to thank C.J. Bell, Lennart Beringer, Robert Dockins, Christopher Monsanto, Cole Schlesinger, David Walker, and the PLPV anonymous reviewers for invaluable suggestions and comments on this work.

References

[1] A. Ahmed, M. Fluet, and G. Morrisett. A step-indexed model of substructural state. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN International Conference on Functional programming*, pages 78–91, 2005. ISBN 1-59593-064-7. doi: http://doi.acm.org/10.1145/1086365.1086376.

[2] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, Princeton, NJ, Nov. 2004. Tech Report TR-713-04.

[3] A. W. Appel and S. Blazy. Separation logic for small-step C minor. In *20th Int'l Conference on Theorem Proving in Higher-Order Logics*, pages 5–21, 2007.

[4] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th Annual Symposium on Principles of Programming Languages (POPL'07)*, pages 109–122, Jan. 2007.

[5] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1981.

[6] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL'05: The 32nd ACM Symp. on Principles of Programming Languages*, pages 259–270, 2005.

[7] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 366–378, 2007. ISBN 0-7695-2908-9. doi: http://dx.doi.org/10.1109/LICS.2007.30.

[8] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE'10: Verified Software: Theories, Tools and Experiments*, 2010.

[9] R. Dockins, A. W. Appel, and A. Hobor. Multimodal separation logic for reasoning about operational semantics. In *24th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIV)*, pages 5–20. Springer Electronic Notes in Theoretical Computer Science (ENTCS), 2008.

[10] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *The 7th Asian Symposium on Programming Languages and Systems*. Springer ENTCS, 2009. URL `http://msl.cs.princeton.edu/fresh-sa.pdf`.

[11] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *In ESOP09: European Symposium on Programming, volume 5502 of LNCS*, pages 363–377. Springer, 2009.

[12] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. European Symp. on Programming (ESOP 2008) (LNCS 4960)*, pages 353–367. Springer, 2008.

[13] A. Hobor, R. Dockins, and A. W. Appel. A theory of indirection via approximation. In *Proc. 37th Annual ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 171–185, Jan. 2010.

[14] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[15] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Automated Reasoning*, 41(1):1–31, 2008.

[16] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01: Annual Conference of the European Association for Computer Science Logic*, pages 1–19, Sept. 2001. LNCS 2142.

[17] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, May 2007.

[18] M. Parkinson. The next 700 separation logics. In G. Leavens, P. OHearn, and S. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010.

[19] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. *Proc. of 21st Annual IEEE Symp. on Logic in Computer Science*, 0:137–146, 2006. ISSN 1043-6871.

[20] M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[21] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Proc. of Foundations of Software Science and Computation Structures*, 2002.