A Theory of Termination via Indirection

Robert Dockins [*]
Princeton University
rdockins@cs.princeton.edu

Aquinas Hobor [†]
National University of Singapore
hobor@comp.nus.edu.sg

Abstract

Step-indexed models provide approximations to a class of domain equations and can prove type safety, partial correctness, and program equivalence; however, a common misconception is that they are inapplicable to liveness problems. We disprove this by applying step-indexing to develop the first Hoare logic of total correctness for a language with function pointers and semantic assertions. In fact, from a liveness perspective, our logic is stronger: we verify explicit time resource bounds. We apply our logic to examples containing nontrivial "higher-order" uses of function pointers and we prove soundness with respect to a standard operational semantics. Our core technique is very compact and may be applicable to other liveness problems. Our results 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 Step-indexed models, Termination

1. Introduction

In the last ten years, a technique called "step indexing" has built semantic models for a wide variety of complex program logics [HDA10, §2]. Step-indexed models **approximate** a class of contravariant domain equations while supporting impredicative polymorphism and contravariant equirecursion. Some power is lost during the approximation, but the consequences are unclear.

Most applications of step-indexing to date have been to problems of *safety*—"nothing bad ever happens"—*e.g.*, partial correctness. A widely-held belief among those familiar with step-indexed models has been that one cost of the approximation is that step-indexed models are inapplicable to problems of *liveness*—"something good eventually happens"—*e.g.*, total correctness.

Our major result is to demonstrate that this widely-held belief is wrong by applying step-indexed models to program termination.

[*] Supported in part by NSF CNS-0910448 and AFOSR FA9550-09-1-0138.

[†] Supported by a Lee Kuan Yew Postdoctoral Fellowship.

November 2, 2010

We define a minimal language with two distinctive features: function pointers and semantic assertions. Semantic assertions are program commands that assert the truth of a formula in logic at a program point. Although at run time semantic assertions are equivalent to `skip`, they are useful during static analysis (*e.g.*, [BCD+05]). Semantic assertions may seem benign, but their inclusion in a language with function pointers leads to the kind of unpleasant contravariant circularity associated with step-indexing.

We design a Hoare logic of total correctness for our Halting Assert Language (HAL). In fact our logic is stronger: it verifies an explicit upper bound on the number of function calls, making it a logic of resource bounds. Since the only source of nontermination in HAL is recursion through function pointers, termination is a direct consequence of an upper bound on the number of calls.

We focus on a program logic for a language containing the combination of function pointers and semantic assertions both because there is a natural liveness problem (program termination) and because it forms a minimal set of program features which exhibits the semantic modeling problems for which step-indexing is well suited. Other domains containing a similar contravariant circularity in their semantic models (*e.g.*, concurrency with first-class locks) are often quite complex in ways unrelated to the circularity. Therefore, HAL is a test bed for semantic techniques that we believe will be applicable in richer settings in the future.

However, our logic for HAL is novel in its own right, since we are not aware of any logic of total correctness for a setting that includes the kind of contravariant circularity present in HAL. Indeed, logics of resource bounds for languages containing function pointers are quite rare, even without semantic assertions.

Our Hoare logic is able to reason about programs that exhibit nontrivial use of function pointers including mutually recursive function groups and higher-order functions. Each recursive group is verified as a whole and combined into proofs of whole-program termination, which makes the logic compositional. Higher-order functions are verified independently of the context in which they will be used, and we are able to apply such functions to themselves without trouble (*e.g.*, `map` of `map`).

Contributions.

- We design a language with two complex features: function pointers and semantic assertions embedded in syntax.
- We develop a series of Hoare axioms that can verify the total correctness of programs written in this language.
- We apply the logic to some example programs.
- We build a semantic model for the Hoare judgment and prove the logic sound with respect to the model. We prove that a program verified in the logic actually terminates with the expected postcondition and connect to a standard operational semantics.
- Our results are machine checked in Coq and available at:

`http://msl.cs.princeton.edu/termination/`

Dagstuhl Seminar Proceedings 10351
Modelling, Controlling and Reasoning About State
http://drops.dagstuhl.de/opus/volltexte/2010/2805

$\chi(\tau)$	$\equiv$	$x := \ell$	load constant ℓ into x
	$\mid$	$x_3 := (x_1, x_2)$	allocate a fresh pair
	$\mid$	$x_2 := x_1.1$	project first component
	$\mid$	$x_2 := x_1.2$	project second component
	$\mid$	$\chi_1(\tau)\,;\ \chi_2(\tau)$	sequence two commands
	$\mid$	`ifnil` x `then` $\chi_1(\tau)$ `else` $\chi_2(\tau)$	test if $x = 0$ and branch
	$\mid$	`call` x	call function pointer x
	$\mid$	`return`	return from function
	$\mid$	$\boxed{\texttt{assert}\ (P : \tau)}$	semantic assertion
$\phi(\tau)$	$\equiv$	$\mathsf{label} \rightharpoonup \chi(\tau)$	parametrized program

Figure 1. Parameterized commands and programs

variable	x	$\equiv$	$\mathbb{N}$	
label	ℓ	$\equiv$	$\mathbb{N}$	
value	v	$\equiv$	label $+$ (value $\times$ value)	
store	ρ	$\equiv$	variable $\rightharpoonup$ value	
measure	t	$\equiv$	store $\rightharpoonup$ $\mathbb{N}$	
predicate	P	$\approx$	(program $\times$ store) $\to \mathcal{T}$	see §6.1
command	c	$\equiv$	χ(predicate)	
stack	s	$\equiv$	list command	
program	Ψ	$\equiv$	label $\rightharpoonup$ command	$= \phi$(predicate)

Figure 2. Basic semantic definitions

$\top,\ \bot$	constants for truth and falsehood
$P \wedge Q,\ P \vee Q$	conjunction and disjunction
$P \Rightarrow Q,\ \neg P$	implication and negation
$\forall a : \tau,\ \exists a : \tau$	impredicative quantification
$\mu X.\ P$	equirecursive predicate
$x \Downarrow v$	variable x evaluates to value v
$[x \leftarrow v]P$	P will hold if x is updated to v
closed(P)	P holds on all stores
$P \vdash Q$	entailment
$\langle\!\vert t \vert\!\rangle$	measure t evaluated on the current store
funptr $\ell\ t\ [\mathbb{P}]\ [\mathbb{Q}]$	ℓ is a pointer to a terminating function with termination measure t, precondition $\mathbb{P}$, and postcondition $\mathbb{Q}$

Figure 3. A variety of predicates (assertions in our Hoare logic)

2. An Introduction to HAL

Here we define the syntax and basic semantic definitions for HAL, and present the core assertions used in our Hoare rules.

We present definitions for commands and programs in Figure 1. Our syntax is parameterized over the type of assertions τ. Most of our commands are unexciting: load a constant into a program variable, create (allocate) a fresh pair, project the first and second components of a pair, sequence two commands, and test if a variable contains the constant 0 and branch accordingly. Note that the subcommands for sequences and branches are parameterized over the same type variable τ. Our `call` instruction is noteworthy because x is a **variable** instead of a constant function label—*i.e.*, x is a function pointer. Our functions do not take explicit arguments; instead, a programmer must establish an ad-hoc calling convention using the store. Our `return` command is standard. The unusual command is the semantic assertion `assert`; here P has the type of the argument τ. We define a parameterized program $\phi(\tau)$ as a partial function from code labels to parameterized commands.

We give the basic semantic definitions for HAL in Figure 2. We use natural numbers for program variables (for readability we use r_i instead of just i for concrete program variables in our examples). We also use natural numbers for code labels (addresses). We define values as trees having labels as leaves. A store (*a.k.a.* register bank) is a partial function from variables to values. A measure is a partial function from stores to natural numbers; we will prove termination by requiring measures to decrease during function calls. A predicate is (essentially) a function from pairs of program and store to truth values $\mathcal{T}$ (`Prop` in Coq). A command is a specialization of a parametrized command χ with predicate; a stack is a list of commands. A program is a partial function from labels to commands—*i.e.*, program $= \phi$(predicate).

Notice that the metatypes predicate, command, and program contain a contravariant cycle. The real semantic definition for predicate, which is similar in flavor to what is given here but with the pleasing addition of being sound, is the subject of §6.1.

We give a variety of predicates in Figure 3. We have constants ($\top$, $\bot$) and the standard logical connectives ($\wedge$, $\vee$, $\Rightarrow$, $\neg$). Of note is that our quantification ($\forall$, $\exists$) is **impredicative**—that is, the metavariable τ ranges over all of the types in our metalogic (τ : `Type` in Coq), including of course the type predicate itself. We provide equirecursive μ to describe recursive program invariants.

The assertion $x \Downarrow v$ means that the variable x evaluates to value v in the current store. We write $[x \leftarrow v]P$ to mean that the predicate P will be true if the current store is updated so that variable x maps to value v; $[x \leftarrow v]$ is therefore a kind of modal operator—the modality of store update. We define another modal operator, closed(P), meaning P holds on all stores.

We write $P \vdash Q$ for predicate entailment. We also introduce a notational convenience for reasoning about measures in the context of a predicate. Since a predicate is more-or-less a function taking (among other things) a store ρ as an argument, and since a measure t is a **partial** function from stores to $\mathbb{N}$, it is straightforward to evaluate $t(\rho)$ and then compare the result against other naturals with the usual operators $=$, $<$, etc. To indicate this kind of evaluation and comparison, we will write *e.g.*, "$\langle\!\vert t \vert\!\rangle < n$"—that is, evaluate t with the current store and require that the result be less than n. When $t(\rho)$ is not defined, terms containing $\langle\!\vert t \vert\!\rangle$ are equivalent to $\bot$.

The assertion of particular interest to the present work is the terminating function pointer assertion "funptr $\ell\ t\ [\mathbb{P}]\ [\mathbb{Q}]$", wherein ℓ is a function address, t is a termination measure, $\mathbb{P}$ is a precondition, and $\mathbb{Q}$ is a postcondition. When funptr $\ell\ t\ [\mathbb{P}]\ [\mathbb{Q}]$ holds:

1. The program has code c at address ℓ (recall that programs are partial functions from code labels to commands); this is exactly why we want predicates to take program*s* as an argument.
2. When c is called with an initial store ρ, if $t(\rho)$ is defined, then c makes at most $t(\rho)$ function calls before returning to its caller.
3. The precondition $\mathbb{P}$ and postcondition $\mathbb{Q}$ are actually functions from some shared type A to predicate, *i.e.*, $\mathbb{P} = \lambda a : A.\ (\ldots)$ and $\mathbb{Q} = \lambda a : A.\ (\ldots)$. The function pointer assertion is actually of form "funptr $\ell\ A\ t\ [\mathbb{P}]\ [\mathbb{Q}]$" but the type variable A has been elided to simplify the presentation.
4. If $t(\rho)$ is defined, then for all a, if the assertion $\mathbb{P}(a)$ holds prior to executing c, then the assertion $\mathbb{Q}(a)$ will hold when c returns. The parameter a is thus used to relate pre- and postconditions to each other over the function call. If functions had explicit arguments and return values, a would relate these too.

The datum a which is shared between pre- and postconditions allows us to specify precice function specifications without requiring auxiliary state. Stated another way, a *is* the auxilary state, but is presented in a way that does not affect the operational semantics.

$$\frac{\begin{array}{ccc} n \le n' & \Gamma' \wedge R \vdash R' & \Gamma' \wedge P' \vdash P \\ \Gamma' \vdash \Gamma & \Gamma' \wedge Q \vdash Q' & \Gamma,\, R \vdash_n \{P\}\, c\, \{Q\} \end{array}}{\Gamma',\, R' \vdash_{n'} \{P'\}\, c\, \{Q'\}}\ \text{Hweaken}$$

$$\frac{\Gamma \wedge P \vdash Q}{\Gamma,\, R \vdash_0 \{P\}\, \texttt{assert}\ Q\, \{P\}}\ \text{Hassert}$$

$$\frac{}{\Gamma,\, R \vdash_0 \{[x \leftarrow \ell]Q\}\, x\ :=\ \ell\, \{Q\}}\ \text{Hlabel}$$

$$\frac{P\ \equiv\ x_1 \Downarrow v_0\ \wedge\ x_2 \Downarrow v_1\ \wedge\ [x_3 \leftarrow (v_0, v_1)]Q}{\Gamma,\, R \vdash_0 \{P\}\, x_3\ :=\ (x_1,\, x_2)\, \{Q\}}\ \text{Hcons}$$

$$\frac{P\ \equiv\ x_1 \Downarrow (v_1, v_2)\ \wedge\ [x_2 \leftarrow v_1]Q}{\Gamma,\, R \vdash_0 \{P\}\, x_2\ :=\ x_1.1\, \{Q\}}\ \text{Hfetch1}$$

$$\frac{P\ \equiv\ x_1 \Downarrow (v_1, v_2)\ \wedge\ [x_2 \leftarrow v_2]Q}{\Gamma,\, R \vdash_0 \{P\}\, x_2\ :=\ x_1.2\, \{Q\}}\ \text{Hfetch2}$$

$$\frac{\begin{array}{c} \Gamma,\, R \vdash_n \{P\}\, c_1\, \{Q\} \\ \Gamma,\, R \vdash_{n'} \{Q\}\, c_2\, \{S\} \end{array}}{\Gamma,\, R \vdash_{n+n'} \{P\}\, c_1\, ;\ c_2\, \{S\}}\ \text{Hseq}$$

$$\frac{\begin{array}{c} \Gamma,\, R \vdash_n \{P_1\}\, c_1\, \{Q\} \qquad \Gamma,\, R \vdash_n \{P_2\}\, c_2\, \{Q\} \\ P\ \equiv\ (x \Downarrow 0\ \wedge\ P_1) \vee (x \Downarrow (v_1, v_2)\ \wedge\ P_2) \end{array}}{\Gamma,\, R \vdash_n \{P\}\, \texttt{ifnil}\ x\ \texttt{then}\ c_1\ \texttt{else}\ c_2\, \{Q\}}\ \text{Hif}$$

$$\frac{}{\Gamma,\, R \vdash_0 \{R\}\, \texttt{return}\, \{\bot\}}\ \text{Hreturn}$$

$$\frac{P\ \equiv\ \begin{cases} x \Downarrow \ell\ \wedge\ \mathsf{funptr}\, \ell\, t\, [\mathbb{P}_\ell]\, [\mathbb{Q}_\ell]\ \wedge \\ \langle\!| t |\!\rangle = n\ \wedge\ \mathbb{P}_\ell(a)\ \wedge\ \mathsf{closed}(\mathbb{Q}_\ell(a) \Rightarrow Q) \end{cases}}{\Gamma,\, R \vdash_{n+1} \{P\}\, \texttt{call}\ x\, \{Q\}}\ \text{Hcall}$$

Figure 4. Hoare rules

3. Total Correctness for HAL

Our program logic is divided into two parts. We provide a set of Hoare rules in §3.1 for verifying commands in HAL A second set of rules, in §3.2, show how to use the verification of a function's body to show that the function satisfies its specification, and can thus be called by other functions.

3.1 Hoare Rules

Our Hoare judgment is a six-place relation written as follows:

$$\Gamma,\, R \vdash_n \{P\}\, c\, \{Q\}$$

Here P, Q, and R are predicates (assertions), Γ is a closed predicate that only looks at programs, n is a natural number and c is a command. We defer the formal semantic model until §7, but the informal meaning is straightforward. P, c, and Q are the standard precondition, instruction, and postcondition tripe common in Hoare logics. The return assertion R is the postcondition of the current function; R must hold before the function can `return`. We collect assertions about function pointers in Γ. Finally, starting from precondition P, n is an upper bound on the number of function calls c will execute before it terminates.

We present the Hoare rules for total correctness in Figure 4. In fact, the rules are mostly unsurprising. The weakening/consequence rule Hweaken allows covariance in the preconditions (P, P') and contravariance in the postconditions (Q, Q') and return conditions (R,R'). The function assertions (Γ, Γ') are related covariantly and incorporated in the other entailments in the most general way. We allow the upper bound on the number of function calls (n,n') to increase during weakening since the bound is not strict.

Although semantic assertions caused significant headaches in the semantic model due to the contravariance outlined in §2, the Hassert rule is pleasingly direct. We simply ensure that the precondition P (including the function assertions in Γ) entails the assertion Q. Since the `assert` command does not make any function calls, we can use $n = 0$ for the upper bound.

The four rules Hlabel, Hcons, Hfetch1, and Hfetch2 are the standard weakest precondition forms for local variable updates for constants, fresh pairs, and first/second projections respectively. Since these rules do not make any function calls $n = 0$.

The sequence rule Hseq and conditional rule Hif look standard. The only point of interest in Hseq is that the upper bounds on the subcommands c_1 and c_2 are summed for the sequence. In the Hif rule, we require that both c_1 and c_2 share the same bound n, which is then used for the conditional. If the natural bounds differ, one increases the lower bound of the lower via weakening.

The rule for function return Hreturn requires that the precondition match the return assertion. After a function `return`s the remainder of the function is not executed, so we provide the postcondition $\bot$. Since `return` does not make any function calls $n = 0$.

The most important rule is Hcall, for verifying a function pointer call. The precondition P has five conjuncts. First, the variable x must point to a code label ℓ. Second, ℓ must be a function pointer to some code with termination measure t, function precondition $\mathbb{P}_\ell$, and function postcondition $\mathbb{Q}_\ell$. Third, the termination measure t must be defined on the current store, and evaluate to some n. That is, starting from the current store, the function ℓ will make no more then n function calls before `return`ing. Fourth, the function precondition $\mathbb{P}_\ell$ must hold when applied to some a. Finally, we require that the function postcondition $\mathbb{Q}_\ell$, when applied to the same a, implies the postcondition Q **in all stores**. It is insufficient to assert that $Q_\ell(a) \Rightarrow Q$ in the current store (*i.e.*, the store before the function call); we must know that the implication will still hold after the function call is completed.

The metavariable a is chosen to relate the function pre- and postconditions to each other over the call. Consider the pair

$$\begin{array}{rcl} \mathbb{P}_\ell & \equiv & \lambda(x, v).\ (r_0 \Downarrow 4)\ \wedge\ \big((x \neq r_0) \Rightarrow x \Downarrow v\big) \\ \mathbb{Q}_\ell & \equiv & \lambda(x, v).\ (r_0 \Downarrow 8)\ \wedge\ \big((x \neq r_0) \Rightarrow x \Downarrow v\big) \end{array}$$

If the caller needs to be sure that the invariant $r_{15} \Downarrow (16, (23, 42))$ is preserved over the function then he sets $a = (r_{15}, (16, (23, 42)))$.

The key point of the HCall rule is that if we satisfy P then we can verify a function pointer call with a bound of $n + 1$ calls.

Precondition generator. Our update rules are stated in weakest-precondition style and our predicates include general quantification. Thus, we can easily define (see Coq development) a precondition generator pg that computes P from R, n, c, and Q such that

$$\text{If}\quad (\Gamma \wedge P) \vdash \mathsf{pg}(R, n, c, Q)\quad \text{then}\quad \Gamma,\, R \vdash_n \{P\}\, c\, \{Q\}$$

We use such a generator to cut down on the tedium of mechanically verifying the example programs presented in §4.

$$\frac{}{\Psi\ :\ \top}\ \text{Vstart}$$

$$\frac{\Psi\ :\ \Gamma \qquad \forall a, n.\ \Big(\big(\Gamma \wedge \mathsf{funptr}\,\ell\,t\,\big[\lambda a'.\,\mathbb{P}(a') \wedge \llparenthesis t \rrparenthesis < n\big]\,\big[\mathbb{Q}\big]\big), \mathbb{Q}(a)\ \vdash_n\ \{\mathbb{P}(a) \wedge \llparenthesis t \rrparenthesis = n\}\ \ \Psi(\ell)\ \ \{\bot\}\Big)}{\Psi\ :\ (\Gamma \wedge \mathsf{funptr}\,\ell\,t\,\big[\mathbb{P}\big]\,\big[\mathbb{Q}\big])}\ \text{Vsimple}$$

$$\frac{\begin{array}{c}\Gamma'(b,n)\ \equiv\ \forall(\ell,t,\mathbb{P},\mathbb{Q}) \in \Phi(b).\ \mathsf{funptr}\,\ell\,t\,\big[\lambda a.\,\mathbb{P}(a) \wedge \llparenthesis t \rrparenthesis < n\big]\,\big[\mathbb{Q}\big] \\ \Psi\ :\ \Gamma \qquad \forall b.\,\forall(\ell,t,\mathbb{P},\mathbb{Q}) \in \Phi(b).\ \Big(\forall a, n.\ \big((\Gamma \wedge \Gamma'(b,n)), \mathbb{Q}(a)\ \vdash_n\ \{\mathbb{P}(a) \wedge \llparenthesis t \rrparenthesis = n\}\ \ \Psi(\ell)\ \ \{\bot\}\big)\Big)\end{array}}{\Psi\ :\ (\Gamma \wedge \forall b.\,\forall(\ell,t,\mathbb{P},\mathbb{Q}) \in \Phi(b).\ \mathsf{funptr}\,\ell\,t\,\big[\mathbb{P}\big]\,\big[\mathbb{Q}\big])}\ \text{Vfull}$$

Figure 5. Single and mutually-recursive function verification

3.2 Function Verification

The whole-function verification rules given in Figure 5 form the heart of our program logic. Although the symbol count is daunting, the core idea is natural and we will cover the details one at a time.

Functions are normally verified one at a time, although mutually recursive function groups are verified as a set. One begins with rule Vstart, which says that program Ψ has specification $\top$ (*i.e.*, that none of the functions in Ψ have been verified to terminate). The Vsimple and Vfull rules verify the addition of terminating function specifications into the context Γ. Vsimple is sufficient to handle simple recursive functions that take non-polymorphic function pointers as arguments. Vfull handles mutually-recursive function groups and polymorphic function pointers; Vsimple is just a special case of Vfull. After verifying the first function/group, one continues with the next function/group with another application of Vsimple/Vfull until all of Ψ has been verified.

The Vsimple rule starts with the assumption that the program Ψ has been proved to have specification Γ; the goal is to add the specification for the function at ℓ using termination measure t, precondition $\mathbb{P}$, and postcondition $\mathbb{Q}$. The main requirement is the second premise. We must verify, using the H-rules, that for any n and a, the function body $\Psi(\ell)$ meets the specification

$$\ldots,\ \mathbb{Q}(a) \vdash_n \{\mathbb{P}(a)\ \wedge\ \llparenthesis t \rrparenthesis = n\}\ \Psi(\ell)\ \{\bot\}$$

That is, starting from a state that satisfies the precondition $\mathbb{P}(a)$ and in which the termination measure t evaluates to n, the function will `return` in a state satisfying $\mathbb{Q}(a)$ after having made no more than n function calls. We use $\bot$ as the postcondition of the Hoare tuple since the function is not allowed to "fall off the bottom". The key to doing recursive functions is how we set up the function specifications: we verify $\Psi(\ell)$ using the previously-verified function specifications in Γ as well as a modified specification for ℓ itself:

$$\mathsf{funptr}\,\ell\,t\,\big[\lambda a'.\,\mathbb{P}(a')\ \wedge\ \llparenthesis t \rrparenthesis < n\big]\ \big[\mathbb{Q}\big]$$

That is, the function body $\Psi(\ell)$ is allowed to contain a recursive call *as long as the termination measure decreases*.

The Vfull rule is more general, and correspondingly, more complex. It generalizes the Vsimple rule in two orthogonal ways. First, Vfull is able to verify a mutually-recursive set of functions. Second, Vfull is able to verify function specifications where the specifications take *parameters*. The universally-quantified variable b in the Vfull rule represents the specification parameters; b can range over an arbitrary type chosen by the verifier. The variable Φ appearing in the Vfull rule represents a finite set of function specifications, i.e., a set of tuples with a label, a termination measure and a pre- and postcondition. The specifications in Φ represent the set of mutually-recursive functions we are going to verify. The quantification over $\Phi(b)$ in the premise of the rule means that we will have to construct a Hoare derivation for each function body represented in Φ. Correspondingly, the quantification in the conclusion means that subsequent verifications may rely on each of the specifications in Φ. In other words, the Vfull rule establishes the specifications of a set of mutually-recursive functions simultaneously.

Note that Φ takes an argument; thus the function specifications can depend on the parameter b.[1] In the premise of the Vfull rule, the value b is bound *once* and the same b is used to construct the recursive assumptions as is used to construct the verification obligations. In other words, the value of the parameter, b, is a constant throughout the recursion. Contrast this with the value a which connects pre- and postconditions, which is allowed to vary at each recursive call. An interesting case occurs when b is allowed to range over function specifications. In this case, the specifications in Φ take on a *higher-order* flavor. We shall see several examples using this power in the following section.

4. Examples of Verified Programming in HAL

Although HAL is Turing complete, it lacks many features enabling abstraction and ease-of-use. The simplicity of our examples does not indicate a theoretical limitation, but is rather a practical consequence of programming in such an impoverished language.

Example 1: unary addition. Here we examine a simple recursive function which "adds" two lists representing natural numbers in unary notation. The basic idea of this function is that there are two unary-encoded naturals (lists terminated by the 0 label) in registers r_1 and r_2. Cons cells are stripped from r_1 and added to r_2 until there are no cells left in r_1, at which point the function returns.

To state the specification of this function, we need a predicate listnat which relates natural numbers to their unary encoding. See Figure 6. Essentially, the number of nested cons cells captures the intended natural number.

Using the listnat predicate we can define the pre- and postconditions of the addition function, which are parametrized by the pair of numbers to be added, as well as the termination measure. Note that we allow termination measures to be partial functions; we use that power here because addt is only defined when the value in r_1 properly encodes some natural number.

Line 1 of the addition function simply asserts the precondition of the function. Line 2 tests if the value in register r_1 is nil (the 0 label). If so, the function returns; otherwise, we perform one unit of work, which involves shifting one cons cell from r_1 to r_2. Note lines 7 and 8, where we load the constant label 1 into r_0 and jump to it; this sequence is typical of "static" function calls. The code is loaded at label 1, so this is the recursive call.

The addition function is a simple self-recursive function, so we can to verify it according to its specification using the Vsimple

[1] This may be quite a strong dependency; even the type of a which connects the pre- and postconditions can depend on the value of b.

Fig. 6a: Encoding naturals

$\mathsf{listnat}(0, 0)$
$\mathsf{listnat}(n, v) \to$
$\quad \mathsf{listnat}(n + 1, (0, v))$

Number	Encoding
0	0
1	(0, 0)
2	(0, (0, 0))
3	(0, (0, (0, 0)))

Fig. 6b: Pre- and postcondition; termination measure

$$\mathsf{addP}(n, m) \equiv \exists v_1\, v_2.\ r_1 \Downarrow v_1 \wedge r_2 \Downarrow v_2 \wedge \mathsf{listnat}(n, v_1) \wedge \mathsf{listnat}(m, v_2)$$

$$\mathsf{addQ}(n, m) \equiv \exists v_2.\ r_2 \Downarrow v_2 \wedge \mathsf{listnat}(n + m, v_2)$$

$$\mathsf{addt}(\rho) \equiv \text{the unique } n \text{ s.t. } \exists v_1.\ \rho(r_1) = v_1 \wedge \mathsf{listnat}(n, v_1)$$

Fig. 6c: Code c_{add}, loaded at label 1

```
1  assert ($\exists n\, m.\ \mathsf{addP}(n, m)$) ;
2  ifnil $r_1$ then return;
3  else
4    $r_3$ := $r_1$.1 ;
5    $r_1$ := $r_1$.2 ;
6    $r_2$ := ($r_3$, $r_2$) ;
7    $r_0$ := 1 ;  // address of $c_{\mathrm{add}}$
8    call $r_0$ ;
9    return ;
```

Fig. 6d: Verification obligation for unary addition (using Vsimple)

$$\forall n_1,\ m_1,\ n.$$
$$\Big(\Gamma,\ (\exists v_2.\ r_2 \Downarrow v_2 \wedge \mathsf{listnat}(n_1 + m_1, v_2)) \vdash_n \{\exists v_1\, v_2.\ r_1 \Downarrow v_1 \wedge r_2 \Downarrow v_2 \wedge \mathsf{listnat}(n, v_1) \wedge \mathsf{listnat}(m, v_2) \wedge \langle\!|\mathsf{addt}|\!\rangle = n\}\ c_{\mathrm{add}}\ \{\bot\}\Big)$$

$$\Gamma \equiv \mathsf{funptr}\ 1\ \mathsf{addt}\ \big[\lambda n_2\, m_2.\ \mathsf{addP}(n_2, m_2) \wedge \langle\!|\mathsf{addt}|\!\rangle < n\big]\ \big[\mathsf{addQ}\big]$$

Figure 6. Example 1: unary addition.

rule. The proof obligation that is generated by Vsimple (after some minor simplifications) is shown in Figure 6. It is straightforward to use the rules of the logic to fulfill this proof obligation. We use the precondition generator from §3.1 to ease the burden somewhat.

Here we sketch the verification proof. The base case occurs when r_1 contains the nil value. In this case $n_1 = 0$ and the postcondition follows immediately. In the recursive case, lines 4–6 do the interesting work of shifting a single cons cell, and moves from a state satisfying $\mathsf{addP}(n_1 + 1, m_1)$ to a state satisfying $\mathsf{addP}(n_1, m_1 + 1)$. The recursive call (lines 7–8) thus has its precondition satisfied, and the termination measure (which tracks n_1) has decreased. The final proof obligation is then to show that $\mathsf{addQ}(n_1, m_1 + 1)$ implies $\mathsf{addQ}(n_1 + 1, m_1)$.

Example 2: apply. While the code for the "apply" function is dead simple, the specification is rather subtle. The "apply" function makes essential use of function pointers and thus has a higher-order specification. The basic idea is that one packages together a function label with some additional arguments using a cons cell in register r_0. Apply unpacks the cons cell and calls the contained function using with the enclosed arguments. We toss in a higher-order `assert` just before the call for fun.

In order to give a reasonable specification for this function and other higher-order operations, we identify a convenient calling convention. We call functions that adhere to our calling convention "standard" functions. Register r_0 is used for passing function arguments and results. Registers r_1–r_4 are callee-saves registers (whose values must be preserved over the call) and all other registers are caller-saves. In addition, we require the precondition, postcondition, and termination measure, for standard functions, to be defined only on the argument/return value (the value in r_0). We say a function satisfies $\mathsf{stdfun}(\ell, t, P, Q)$ (where t, P and Q are defined over a single value rather than an entire store) if ℓ is a standard function in the sense just defined. The stdfun predicate can be defined in terms of funptr in a straightforward way.

In the specification for apply (Figure 7) t, P and Q are the *parameters of the specification*; they describe the function that will be called. When it comes time to verify the apply function, we will use the Vfull rule and b will range over tuples (t, P, Q). This way we can specify and prove correct the apply function in complete isolation, without requiring any static assumptions about the functions it will be passed. In some later verification, the specification of apply can be instantiated with any function specification that the verifier knows about. In particular, apply can be applied to itself! This would not be a recursive call, in the traditional sense, but is rather a *dynamic* higher-order call.

Termination remains assured because the way the specifications get "stacked" on top of each other. This stacking of function specifications, in general, creates a tree-like structure where the leaves of the tree must be first-order functions (whose specifications do not depend on the specifications of other functions). The whole thing hangs together because there is no way to create a cycle in the tree of function specifications, and thus no way to introduce new, potentially nonterminating, recursion patterns. See the formal development for an example of such "stacked" function applications.

Example 3: map. Our final example is the map function, which applies some operation to every element in a list. "Map" exhibits both the higher-order nature of "apply" and also has an interesting recursive structure of its own. In HAL, even this simple function requires more than 20 lines of code and its specification is tedious (due mostly to the details of the calling convention), so here we will merely give the program text and sketch the main ideas of its verification. So that the map function will be a "standard" function, we break it into two pieces: a worker and a wrapper. The worker is the interesting recursive function and the wrapper merely takes care of the details of saving and restoring registers.

For the map worker, the list to be mapped over is stored in r_3, and the label for the "standard" function to be applied to each list element is in r_2. r_1 plays the role of a data stack, and r_0 is used to exchange arguments and results with the function being called. Since the function in r_2 is standard, we can rely on the values of r_1–r_4 being unchanged across function calls. The termination measure for this function is essentially a sum, ranging over all the elements in the list, of the termination measure of the mapped function.

The overall pre/post specification for map is built using the specification of the mapped function. Basically, the precondition for map asserts that the values in the list to be mapped associate pairwise with data values a_i according to the mapped function's precondition, and that the given function pointer is actually a standard function with the given specification. Likewise, the postcondition asserts that the returned list associates pairwise with the same data a_i according to the mapped function's postcondition.

Because we carefully arranged the map wrapper function to be a "standard" function, and because we verified map with a higher-order specification it is possible to use map, itself, as a function to map over a list! Imagine that we have set up an initial state consisting of a list in register r_0 such that each element of the list is a cons cell containing the label constant 5 (the map wrapper function!) in the first component and the second, another cons cell containing a pair of some *other* function to map over the list in the next component. Then, if we call map with the apply function, we

Fig. 7a: "standard" functions

$\mathsf{csregs}(v_1, v_2, v_3, v_4) \;\equiv\; (r_1 \Downarrow v_1) \wedge (r_2 \Downarrow v_2) \;\wedge\; (r_3 \Downarrow v_3) \;\wedge\; (r_4 \Downarrow v_4)$

$\mathsf{stdfun}(\ell, t_\ell, P_\ell, Q_\ell) \;\equiv\; \mathsf{funptr}\;\ell\;\big(\lambda\rho.\, t_\ell(\rho(r_0))\big)$
$\big[\lambda(v_1, v_2, v_3, v_4, a).\,(r_0 \Downarrow v_0) \;\wedge\; P_\ell(a)(v_0) \wedge \mathsf{csregs}(v_1, v_2, v_3, v_4)\big]$
$\big[\lambda(v_1, v_2, v_3, v_4, a).\,(r_0 \Downarrow v_0) \;\wedge\; Q_\ell(a)(v_0) \wedge \mathsf{csregs}(v_1, v_2, v_3, v_4)\big]$

$\mathsf{applyP}(a)(v) \;\equiv\; \exists \ell\; v_2.\; v = (\ell, v_2) \wedge \mathsf{stdfun}(\ell, t, P, Q) \wedge P(a)(v_2)$

$\mathsf{applyQ}(a)(v) \;\equiv\; Q(a)(v)$

$\mathsf{applyt}(v) \;\equiv\; t(v) + 1$

Fig. 7b: code c_{app} for apply with abbreviated Hoare triples

$$R \;\equiv\; \{r_0 \Downarrow v' \wedge Q(a)(v')\}$$

```
    ⊢₀ {r₀ ⇓ (ℓ, v) ∧ P(a)(v) ∧ t(v)+1 = n ∧ stdfun(ℓ, t, P, Q)}
1   r₅ := r₀.0 ;
    ⊢₀ {r₀ ⇓ (ℓ, v) ∧ r₅ ⇓ ℓ∧
        P(a)(v) ∧ t(v) + 1 = n ∧ stdfun(ℓ, t, P, Q)}
2   r₀ := r₀.1 ;
    ⊢₀ {(r₀ ⇓ v) ∧ (r₅ ⇓ ℓ) ∧ P(a)(v) ∧ (t(v) + 1 = n) ∧ stdfun(ℓ, t, P, Q)}
3   assert (∃ℓ′, t′, P′, Q′. (r₅ ⇓ ℓ′) ∧ stdfun(ℓ′, t′, P′, Q′)) ;
    ⊢₀ {(r₀ ⇓ v) ∧ (r₅ ⇓ ℓ) ∧ P(a)(v) ∧ (t(v) + 1 = n) ∧ stdfun(ℓ, t, P, Q)}
4   call r₅ ;
    ⊢ₙ {r₀ ⇓ v′ ∧ Q(a)(v′)}
5   return ;
    ⊢ₙ {⊥}
```

Fig. 7c: Verification obligation for apply (using Vfull)

$$\forall (t, P, Q)\;(v_1, v_2, v_3, v_4, a)\; n.\; \Big(\top,\; (\exists v_0'.\; Q(a)(v_0') \wedge r_0 \Downarrow v_0' \wedge \mathsf{csregs}(v_1, v_2, v_3, v_4)) \;\vdash_n \{\exists \ell\; v_0.\; \mathsf{stdfun}(\ell, t, P, Q) \wedge P(a)(v_0) \wedge r_0 \Downarrow (\ell, v_0) \wedge \mathsf{csregs}(v_1, v_2, v_3, v_4) \wedge t(v_0) + 1 = n\}\;\; c_{\mathrm{app}}\;\; \{\bot\}\Big)$$

Figure 7. Example 2: apply

```
ifnil r₃
then
  r₀ := 0 ; return ;    // base case, return "nil"
else
  r₀ := r₃.1 ;          // get the head of the list
  call r₂ ;             // call the mapped function
  r₁ := (r₀, r₁) ;      // push the mapped value
  r₃ := r₃.2 ;          // pop the head of the list
  r₅ := 4 ;             // recursive "map" call
  call r₅ ;
  r₅ := r₁.1 ;          // add the new list head
  r₀ := (r₅, r₀) ;
  r₁ := r₁.2 ;          // pop the stack
  return ;
```

Figure 8. Listing for map worker (at label 4)

```
r₁ := (r₂, r₁) ;    // push registers r₂ and r₃
r₁ := (r₃, r₁) ;
r₂ := v₀.0 ;        // load the function pointers into r₂
r₃ := v₀.1 ;        // load list argument into r₃
r₅ := 4 ;           // call map worker
call r₅ ;
r₃ := r₁.0 ;        // restore r₃
r₁ := r₁.1 ;
r₂ := r₁.0 ;        // restore r₂
r₁ := r₁.1 ;
return ;
```

Figure 9. Listing for map wrapper (at label 5)

get the effect of mapping map over each component sublist. We can use the apply function to build a kind of poor man's closure system which allows us to perform arbitrarily nested map operations.

With a little effort, one could build a real closure system along similar lines by defining a function call convention specifically for the purpose. The logic has the necessary ingredients of higher-order specifications and impredicative quantification.

5. Erased Operational Semantics for HAL

We are ready to present the operational semantics for HAL. In this section we give HAL an *erased* semantics—that is, a semantics that does not depend on the complicated semantic model for predicates. Our erased semantics demonstrates that HAL has a reasonable, conventional, and fully-computable operational semantics. We state our end-to-end theorem in terms of our erased semantics to avoid ambiguity over the meaning of what we have proved.

In §7.1 we will give HAL an *unerased* semantics that will depend on the model for predicates and will be considerably less standard. In fact, it will not be computable! We will prove an *erasure* theorem giving the relationship between the two semantics.

The erased step relation for HAL $\overset{E}{\mapsto}$ takes five arguments:

$$\Psi \vdash (\rho,\, s) \overset{E}{\mapsto} (\rho',\, s')$$

As in §2, we have a program Ψ; stores ρ, ρ'; and stacks s, s'. The erased step relation is defined as the least relation satisfying the rules given in Figure 10. The machine steps by executing the command c at the head of the stack; the tail contains the function return points. The command c must be a sequence c_1 ; c_2. The step rule applicable depend primarily on the command c_1. If the head of the stack s is not a sequence then the machine is stuck. This helps ensure that we do not "fall off the bottom" of a function—we must execute a `return` instruction **before** we reach the end. In the rule Ecall, we append a special instruction, `assert` $\bot$, to the end of the called function's code to ensure that if the last instruction is a `return` then it will execute as expected.[2]

The individual rules in the erased step relation are entirely conventional—indeed, that is the point. We say that a command c is *straightline* if the step relation always continues with the next instruction in the sequence. There are four rules that handle the simple straightline commands: Elabel, Econs, Efetch1, and Efetch2. All four modify the store ρ in the expected ways: loading a constant (*i.e.*, the code label ℓ) into a variable x (Elabel); storing

[2] One alternative is to add a second rule (Ereturn2) for `return` to handle the special case when there is no following instruction in the sequence.

$$\frac{}{\Psi \vdash (\rho,\ (x\ :=\ \ell\,;\ c) :: s) \overset{E}{\mapsto} ([x \leftarrow \ell]\rho,\ c :: s)}\ \text{Elabel} \qquad \frac{\rho(x_1) = v_1 \qquad \rho(x_2) = v_2}{\Psi \vdash (\rho,\ (x_3\ :=\ (x_1,\ x_2)\,;\ c) :: s) \overset{E}{\mapsto} ([x_3 \leftarrow (v_1, v_2)]\rho,\ c :: s)}\ \text{Econs}$$

$$\frac{\rho(x_1) = (v_1, v_2)}{\Psi \vdash (\rho,\ (x_2\ :=\ x_1.1\,;\ c) :: s) \overset{E}{\mapsto} ([x_2 \leftarrow v_1]\rho,\ c :: s)}\ \text{Efetch1} \qquad \frac{\rho(x_1) = (v_1, v_2)}{\Psi \vdash (\rho,\ (x_2\ :=\ x_1.2\,;\ c) :: s) \overset{E}{\mapsto} ([x_2 \leftarrow v_2]\rho,\ c :: s)}\ \text{Efetch2}$$

$$\frac{}{\Psi \vdash (\rho,\ (\texttt{assert}\ P\,;\ c)\ ::\ s) \overset{E}{\mapsto} (\rho,\ c\ ::\ s)}\ \text{Eassert} \qquad \frac{\rho(x) = 0}{\Psi \vdash (\rho,\ (\texttt{ifnil}\ x\ \texttt{then}\ c_1\ \texttt{else}\ c_2\,;\ c) :: s) \overset{E}{\mapsto} (\rho,\ (c_1\,;\ c) :: s)}\ \text{Eif1}$$

$$\frac{}{\Psi \vdash (\rho,\ ((c_1\,; c_2)\,; c_3) :: s) \overset{E}{\mapsto} (\rho,\ (c_1\,; (c_2\,; c_3)) :: s)}\ \text{Eseq} \qquad \frac{\rho(x) = (v_1, v_2)}{\Psi \vdash (\rho,\ (\texttt{ifnil}\ x\ \texttt{then}\ c_1\ \texttt{else}\ c_2\,;\ c) :: s) \overset{E}{\mapsto} (\rho,\ (c_2\,;\ c) :: s)}\ \text{Eif2}$$

$$\frac{}{\Psi \vdash (\rho,\ (\texttt{return}\,;\ c)\ ::\ s) \overset{E}{\mapsto} (\rho,\ s)}\ \text{Ereturn} \qquad \frac{\rho(x) = \ell \qquad \Psi(\ell) = c'}{\Psi \vdash (\rho,\ (\texttt{call}\ x\,;\ c) :: s) \overset{E}{\mapsto} (\rho,\ (c'\,;\ \texttt{assert}\ \bot) :: c :: s)}\ \text{Ecall}$$

Figure 10. Erased Operational Semantics

the new pair (v_1, v_2) into variable x_3 (Econs); and projecting the first and second components, respectively, of the pair (v_1, v_2) into variable x_2 (Efetch1 and Efetch2). We write $\rho(x)$ to reference the value ρ associated with x, and $[x \leftarrow v]\rho$ to mean the fresh store derived from ρ by updating variable x to contain the value v.

The "complicated" straightline command assert is a nop in the erased semantics; it does nothing at all. This a feature, not a bug. It means our erased semantics is entirely standard and computable. The *unerased* semantics in §7.1 "checks" assertions as it steps.

There are three control-flow rules not involving `call`/`return`. The sequence rule (Eseq) is a bookkeeping rule that simply reassociates ";". The two branch rules (Eif1 and Eif2) test whether the variable x is equal to 0 or is equal to some pair (v_1, v_2). If it is 0 (Eif1), the machine steps to c_1; otherwise (Eif2), it steps to c_2. If the value of x is some label other than 0, the machine is stuck.[3]

The final two rules both involve functions. The rule for returning from a function (Ereturn) simply pops the head of the stack s. The call rule (Ecall) is slightly more complicated. Here, we look up the label ℓ stored in variable x, and use the program Ψ to extract the function body c'. We suspend the remainder of the current function c by pushing it onto the stack, and put the new function body c' on top, with an appended assert $\bot$ as explained above.

We write the reflexive, transitive closure of $\overset{E}{\mapsto}$ as $\overset{E}{\mapsto}^*$. We say that a *configuration* (Ψ, ρ, s) halts in the erased semantics when:

$$\mathsf{halts}^E(\Psi, \rho, s) \ \equiv\ \exists \rho'.\ \Psi \vdash (\rho, s)\ \overset{E}{\mapsto}^*\ (\rho', \texttt{nil}) \qquad (1)$$

That is, a configuration will halt in the erased semantics if it will eventually `return` from the top-level function ($s' = \texttt{nil}$). Programs verified in our logic will halt by this definition on our erased semantics. We give the formal statement and proof of this soundness theorem in §7. Informally, the soundness theorem says if:

1. We have used our H- and V-rules to verify $\Psi : \Gamma$
2. Γ includes a function at location ℓ with function body c, termination measure t, precondition $\mathbb{P}$, and postcondition $\mathbb{Q}$
3. Termination measure t is defined on some initial state
4. Precondition $\mathbb{P}(a)$ holds on that state for some a

Then the configuration whose code is simply $c :: \texttt{nil}$ will eventually halt in a configuration satisfying the postcondition $\mathbb{Q}(a)$.

Until we have given the semantic model for predicates in §6.1 we are unable to state this soundness theorem in full generality since we have not formally explained what "a state satisfying P" means. However, we can state a weaker version now:

Theorem 1 (Total Correctness, Full Erasure)**.** *Suppose* $\Psi : \Gamma$ *and* $\Gamma \vdash \mathsf{funptr}\ \ell\ t\ \big[\lambda a.\ \top\big]\ \big[\lambda a.\ \top\big]$*. Let* ρ *be any* store *such that* $t(\rho)$ *is defined. Then* $\mathsf{halts}^E(\Psi, \rho, \Psi(\ell))$*.* *Proved in §7.3.*

Observation. Since the rules in Figure 10 do not do anything with the argument to the `assert` statement, the erased semantics does not care at which (nonempty) type τ the commands $\chi(\tau)$ and program $\phi(\tau)$ are instantiated. A reader concerned about our semantic model is encouraged to substitute predicate with unit and $\bot$ with the unit value. One obtains a program and a proof that the program halts according to an operational semantics which contains none of the complexities of our semantic model for predicates.

6. Resolving the Circularity in predicates

The remaining task for this paper is to show how our axiomatic semantics for HAL from §3 connects to our erased operational semantics for HAL from §5. That is, we need to provide a semantic model for our logic and then prove that model sound with respect to our operational semantics. We divide the modeling task into two parts. In this section we resolve the circularity in the definition of predicate from Figure 2—that is, for the assertions of our Hoare tuples. In §7 we will build a model for the program logic itself.

6.1 Using Indirection Theory to Stratify Through Syntax

The pseudomodel of predicates in Figure 2 fits into a pattern:

$$K \qquad \approx \qquad F((K \times O) \to \mathcal{T}) \qquad (2)$$

In this pattern, F is a covariant functor, O is some kind of "flat data", and K is an object one wishes to model. Unfortunately, a cardinality argument shows that there are no solutions in set theory to pseudoequation (2), so we will content ourselves with building an approximate model using indirection theory [HDA10]. In our case, F is the parameterized program ϕ from Figure 1 and O is just store. Indirection theory "ties the knot" and defines K such that:

$$\begin{array}{lcll} \mathsf{K} & & \equiv & [\text{ADH10}, \texttt{knot\_hered.v}] \\ \mathsf{sq\_program} & \check{\Psi} & \equiv & K \\ \mathsf{state} & \sigma & \equiv & \mathsf{sq\_program} \times \mathsf{store} \\ \mathsf{predicate} & P & \equiv & \{P : \mathsf{state} \to \mathcal{T} \mid \mathsf{hereditary}(P)\} \end{array} \qquad (3)$$

The construction of the *knot* K is similar to the one given in [HDA10, §8] but we have enhanced it so that all predicates inside the knot are *hereditary*, a technical property detailed in §6.2. A *squashed program* sq_program is simply a knot; a state is a pair

[3] The motivation is that it is easy to distinguish 0 from a valid data pointer, but not easy to distinguish a data pointer from an arbitrary function pointer.

of a $\mathsf{sq\_program}$ and a store. A $\mathsf{predicate}$ is a hereditary function from states to truth values $\mathcal{T}$. We write $\sigma \models P$ instead of $P(\sigma)$ when we wish to emphasize that we are thinking of P as an assertion as opposed to a function. What is missing is the relationship between squashed and unsquashed programs. We cannot have an isomorphism or we run into cardinality problems; instead, we get:

$$\mathsf{sq\_program} \quad \preceq \quad \mathbb{N} \times \mathsf{program}.$$

Here $X \preceq Y$ means that the "small" type X and the "big" type Y are related by a section/retraction pair witnessed by two functions:

$$\begin{array}{rcl} \mathsf{squash} & : & (\mathbb{N} \times \mathsf{program}) \to \mathsf{sq\_program} \\ \mathsf{unsquash} & : & \mathsf{sq\_program} \to (\mathbb{N} \times \mathsf{program}) \end{array} \qquad (4)$$

The power of indirection theory is that there is a very simple (indeed, categorical) pair of axioms relating squash and $\mathsf{unsquash}$:

$$\begin{array}{rcl} \mathsf{squash}(\mathsf{unsquash}(\check{\Psi})) & = & \check{\Psi} \\ \mathsf{unsquash}(\mathsf{squash}(n, \Psi)) & = & (n, \mathsf{prog\_approx}_n(\Psi)) \end{array} \qquad (5)$$

That is, $\mathsf{squash} \circ \mathsf{unsquash}$ is the identity function, and $\mathsf{unsquash} \circ \mathsf{squash}$ is a kind of approximation function. The $\mathsf{prog\_approx}_n(\Psi)$ function transforms Ψ by locating all of the $\mathtt{assert}(P)$ statements and replacing them with $\mathtt{assert}(\mathsf{approx}_n(P))$. The core of the approximation is handled by the $\mathsf{approx}_n(P)$ function:

$$\begin{array}{rcl} |\check{\Psi}| & \equiv & (\mathsf{unsquash}(\check{\Psi})).1 \\ \mathsf{approx}_n(P) & \equiv & \lambda(\check{\Psi}, \rho).\begin{cases} P(\check{\Psi}, \rho) & |\check{\Psi}| < n \\ \bot & |\check{\Psi}| \geq n \end{cases} \end{array} \qquad (6)$$

First we define the *level* of a squashed program $\check{\Psi}$, written $|\check{\Psi}|$, as the first projection of $\check{\Psi}$'s $\mathsf{unsquashing}$. When a $\mathsf{predicate}$ is approximated to level n, its behavior on programs of level **strictly** less than n is unchanged; on programs of level greater than or equal to n it now returns the constant $\bot$. The approx function is exactly where step-indexed models get both their power (a sound construction) and weakness (a loss of information during approximation).

6.2 Consequences of Approximation

What is the cost of throwing away information during approximation? Ten years after step-indexed models were introduced, the answer is still unclear. Considerable experience has led to an ad-hoc understanding among practitioners of step-indexing of what might be termed *microcosts*—that is, small modifications to the "intuitive" definitions to accommodate the approximation. A large body of previous work has focused on managing and minimizing these microcosts, *e.g.*, via a Gödel-Löb logic of approximation [Ric10].

The fundamental microcost comes from the fact that the approx_n function throws away all behavior on squashed programs of greater than **or equal to** level n. Let P be a $\mathsf{predicate}$ contained in (the $\mathsf{unsquashing}$ of) a squashed program $\check{\Psi}$ of level n. A consequence of equations (5) is that P has been approximated to level n—*i.e.*,

$$P \quad = \quad \mathsf{approx}_n(P)$$

Given this equality, what happens if we apply P to a state containing $\check{\Psi}$? A quick examination of equation (6) demonstrates that the result must be $\bot$. **A predicate cannot say anything meaningful about the squashed program whence it came.** Instead, we will do the next best thing: make $\check{\Psi}$ a little simpler. We say that $\check{\Psi}$ (or σ) is *approximated* to $\check{\Psi}'$ (or σ'), written $\check{\Psi} \rightsquigarrow \check{\Psi}'$, when

$$\begin{array}{rcl} \check{\Psi} \rightsquigarrow \check{\Psi}' & \equiv & \mathsf{let}\ (n, \Psi) = \mathsf{unsquash}(\check{\Psi})\ \mathsf{in} \\ & & (n > 1)\ \wedge\ (\check{\Psi}' = \mathsf{squash}(n-1, \Psi)) \\ (\check{\Psi}, \rho) \rightsquigarrow (\check{\Psi}', \rho') & \equiv & (\rho = \rho')\ \wedge\ (\check{\Psi} \rightsquigarrow \check{\Psi}') \end{array} \qquad (7)$$

That is, we $\mathsf{unsquash}$ $\check{\Psi}$ and then re-squash it to one level lower. Of course, we can only do this when we are not at level 0 to begin with! Since $|\check{\Psi}'| = n - 1 < n$, P will be able to judge states containing $\check{\Psi}'$. Every time we pull a $\mathsf{predicate}$ out of a squashed program $\check{\Psi}$, we will approximate $\check{\Psi}$ to $\check{\Psi}'$ before we use P.

This repeated approximation leads to a second microcost. Suppose $\sigma \models P$ and $\sigma \rightsquigarrow \sigma'$. We want P to be *hereditary*—*i.e.*, stable (or monotonic) as σ is approximated—so that $\sigma' \models P$:

$$\mathsf{hereditary}(P) \ \equiv\ \forall \sigma.\ (\sigma \models P) \to (\sigma \rightsquigarrow^* \sigma') \to (\sigma' \models P) \qquad (8)$$

We write $\rightsquigarrow^*$ and $\rightsquigarrow^+$ to mean the reflexive and irreflexive transitive closures, respectively, of $\rightsquigarrow$. Unfortunately, not all functions from state to $\mathcal{T}$ are hereditary, such as $P_{\mathrm{bad}}(\check{\Psi}, \rho) \ \equiv\ |\check{\Psi}| > 5$. The P_{bad} function will be true only while the level of the program is greater than 5; since approximating the state decreases that level, eventually this function will produce only the constant $\bot$.

In our setting, we only consider $\mathsf{predicates}$ that are hereditary. Whenever we give the semantics of a predicate (except for P_{bad}!), we have proved (in Coq) that the definition is hereditary. For most definitions this is not a big deal; however on occasion there can be a significant amount of work that needs to be done. The benefit of only allowing hereditary $\mathsf{predicates}$ is that once the definitions are done they are easier to use in the core soundness proofs.

A central question is how these kinds of microcosts become macrocosts—that is, what are the fundamental limitations of step indexing techniques? For some time, it was thought that step-indexed models could not be applied to problems of liveness; however, the present work proves otherwise. The practical limitations of step-indexed models remain unknown.

6.3 Semantic Models for Key Assertions

In Figure 11 we give the semantic models for the $\mathsf{predicates}$ first presented in Figure 3. The first nine are the basic logical operators, largely defined by lifting to the metalevel as in [HDA10, §6]. With the notable exception of implication we use the same symbols for both the object level and the metalevel, trusting in the context to disambiguate. Importantly, these operators preserve hereditariness—if P and Q are hereditary, then so are $P \wedge Q$, $P \Rightarrow Q$, etc.

The first four operators ($\top$, $\bot$, $\wedge$, and $\vee$) are entirely as expected. The story is more complicated with implication since

$$\begin{array}{rclcl}
\sigma & \models & \top & \equiv & \top \\
\sigma & \models & \bot & \equiv & \bot \\
\sigma & \models & P \wedge Q & \equiv & (\sigma \models P) \wedge (\sigma \models Q) \\
\sigma & \models & P \vee Q & \equiv & (\sigma \models P) \vee (\sigma \models Q) \\
\sigma & \models & P \Rightarrow Q & \equiv & \forall \sigma'.\ (\sigma \rightsquigarrow^* \sigma') \to \\
 & & & & \quad (\sigma' \models P) \to (\sigma' \models Q) \\
 & & \neg P & \equiv & P \Rightarrow \bot \\
\sigma & \models & \forall x : \tau.\ P(x) & \equiv & \forall x : \tau.\ \sigma \models P(x) \\
\sigma & \models & \exists x : \tau.\ P(x) & \equiv & \exists x : \tau.\ \sigma \models P(x) \\
\sigma & \models & \mu X.\ P & \equiv & [\text{ADH10}, \mathtt{predicates\_rec.v}] \\
 & & & & \\
(\check{\Psi}, \rho) & \models & x \Downarrow v & \equiv & \rho(x) = v \\
(\check{\Psi}, \rho) & \models & [x \leftarrow v]P & \equiv & (\check{\Psi}, [x \leftarrow v]\rho) \models P \\
(\check{\Psi}, \rho) & \models & \mathsf{closed}\ P & \equiv & \forall \rho'.\ (\check{\Psi}, \rho') \models P \\
 & & & & \\
 & & P \vdash Q & \equiv & \forall \sigma.\ (\sigma \models P) \to (\sigma \models Q) \\
(\check{\Psi}, \rho) & \models & \langle\!|t|\!\rangle < n & \equiv & t(\rho) < n \quad (\text{etc. for } \langle\!|t|\!\rangle = n) \\
 & & & & \\
\multicolumn{3}{r}{\sigma \models \mathsf{funptr}\ \ell\ t\ [\mathbb{P}]\ [\mathbb{Q}]} & \equiv & \text{model given in §7} \\
 & & & & \\
\sigma & \models & \triangleright P & \equiv & \forall \sigma'.\ (\sigma \rightsquigarrow^+ \sigma') \to (\sigma' \models P)
\end{array}$$

Figure 11. Semantics of Assertions

object-level implication $\Rightarrow$ is **not** a straightforward lift of meta-level implication $\rightarrow$. Instead, we define object-level implication $P \Rightarrow Q$ in an intuitionistic style over the relation $\leadsto^*$; this ensures that the result is hereditary. Although the context always dictates which level is intended, we distinguish the two symbols to emphasize the additional semantics. Those verifying programs with our logic need not be concerned with this detail: all of our object-level operators have the standard introduction/elimination rules and so behave as expected. Logical negation $\neg P$ is equivalent to $P \Rightarrow \bot$.

We define our quantifiers via a straightforward lift to the metalevel quantifiers. The ability to define natural, impredicative quantifiers is a major strength of step-indexed models. The definition of our (contravariant-supporting) equirecursive operator μ is somewhat complicated. Since we do not use μ in our examples or proofs we elide it here; interested readers should consult the mechanization or the similar construction in Appel *et al.* [AMRV07, §5].

The second group of three predicates relate to the store. Evaluating the variable x to a value v is done exactly as might be expected using the store ρ. The store update modality $[x \leftarrow v]P$ holds on some state $(\check{\Psi}, \rho)$ if the underlying predicate P would hold on the related state where the store ρ has been updated so that x now maps to v. Finally, the modality of closure $\mathsf{closed}(P)$ holds on some state $(\check{\Psi}, \rho)$ when P is true given any ρ'.

Entailment $P \vdash Q$ is simply universally quantified metaimplication. In Figure 11 we model a clause that uses the notation $\langle\!|t|\!\rangle$; other clauses are similar (*e.g.*, $(\check{\Psi}, \rho) \models \langle\!|t|\!\rangle{=}n \ \equiv\ t(\rho){=}n$).

We define the modality of approximation $\triangleright$ using the irreflexive closure $\leadsto^+$. If $\triangleright P$ holds on some state σ, then P holds on all **strictly** more approximate σ'; P need not hold on σ itself. Recall from §3 that the approximation modality is not used in the statement of the H- or V-rules. It is only used within the soundness proof itself; those verifying programs using our logic never see it.

7. A Step-indexed Model for Total Correctness

We have now defined our model for predicates and most the operators of our assertion logic. We have three remaining modeling tasks: the terminating function pointer assertion, the Hoare judgment, and the program verification judgment.

7.1 Operational Semantics

Here we give HAL a second, *unerased*, semantics used as the basis for our soundness proof. An erasure theorem will show that the new semantics is a conservative approximation to the old one from §5.

The unerased step relation $\overset{U}{\mapsto}$ takes six arguments:

$$(\check{\Psi},\, \rho,\, s) \overset{U}{\mapsto} (\check{\Psi}',\, \rho',\, s')$$

As before, ρ and s stand for local variables and stacks. Unlike the erased semantics, the unerased semantics takes its program in squashed form, and the program is actually modified as it is running. This modification occurs only in a very controlled way.

The unerased semantics is nearly identical to the erased semantics already presented in Figure 10. Only two rules require significant modification; these appear in Figure 12. All the other rules are identical to their erased counterparts, except that the program $\check{\Psi}$ is passed through unchanged, *e.g.*, the rule for loading a label is:

$$\frac{}{(\check{\Psi},\, \rho,\, (x \ := \ \ell\, ;\, c) :: s) \overset{U}{\mapsto} (\check{\Psi},\, [x \leftarrow \ell]\rho,\, c :: s)}\ \text{Slabel}$$

All the rules except Ecall and Eassert are modified in the same way.

$$\begin{array}{lcl} \mathsf{halts}(\check{\Psi}, \rho, s) & \equiv & \exists \check{\Psi}', \rho', s'.\, (\check{\Psi}, \rho, s) \overset{U}{\mapsto}{}^* (\check{\Psi}', \rho', s') \ \wedge \\ & & \quad s' = \mathtt{nil} \\ (\check{\Psi}, \rho) \models \mathsf{halts}_n\ s & \equiv & \boxed{|\check{\Psi}| \geq n \rightarrow \mathsf{halts}(\check{\Psi}, \rho, s)} \\ \mathsf{guards}_n\ P\ s & \equiv & P \Rightarrow \mathsf{halts}_n\ s \end{array}$$

Figure 13. Halting and guarding

In the unerased rule Sassert, `assert` statements are actually "checked" in the precondition of the step rule. Thus, an unerased program will get stuck if a false assertion is encountered. This simple change renders the semantics *uncomputable*, in the sense that no algorithm can determine if an arbitrary statement in higher-order logic holds. A program verified with our Hassert rule will know that the assertion holds and so will not get stuck.

The other major change involves the rule for function calls. In the unerased rule, we must unsquash the program to get out the instruction sequence for the called function. At function calls, we also take the opportunity to further approximate the program; this has the effect of decreasing the level of the program by 1 and approximates all the assertions appearing in it.

We must do this approximation so that assertions in the text of the function body will be able to judge the program. Recall from §6.2 that an approximated predicate (such as one in the function we are about to jump to) can only judge worlds of strictly smaller level. If we did not approximate the program at this point, any assertions in the function body would fail, foiling our desired soundness result. This means that the level of the program $\check{\Psi}$ is an upper bound on the number of calls the program can make before getting stuck. The oft-maligned indexes of step-indexed models have a practical utility as witnesses to the number of allowed function calls.

7.2 Function Pointers, Hoare tuples, and Verification

Soundness of a logic of total correctness (w.r.t. its operational semantics) means that whenever a function in a verified program is run in an initial state satisfying its precondition, it will halt in a state satisfying its postcondition. Our soundness proof follows the program outlined by Appel and Blazy [AB07] which involves: building a semantic model for assertions; defining the meaning of judgments; proving the inference rules of the logic as lemmas; and showing that the judgment semantics implies the desired theorem.

The first step we have already discussed in §6; it involves using indirection theory to build a model of program syntax capable of containing semantic assertions.

Judgment Definitions. Appel and Blazy build their semantic Hoare triple using the more basic notion of guarding. They say that a predicate "guards" a program stack if, whenever a memory state satisfies the predicate, that stack is safe to run (i.e., will not go wrong). We follow a similar pattern, but use a guards predicate which enforces termination rather than safety. We say that a predicate P guards a stack s *at level n* if, whenever the memory state satisfies P *and* provided that the program level is at least n, running the stack will eventually terminate. See Figure 13. Notice that there is a clever trick being played here with the definition of halts_n. Halting is not normally a predicate which can be hereditary. As one ages a program, it is able to run for fewer steps and thus might not terminate before it exhausts its level. We work around this issue by saying that a program must only terminate if it has at least level n. As one ages a program, it will eventually cause halts_n to be true vacuously (when its level falls below n).

Figure 12. Unerased Operational Semantics

$$\frac{(\check{\Psi}, \rho) \models P}{(\check{\Psi},\, \rho,\, (\mathtt{assert}\ P\, ;\, c)\ ::\ s) \overset{U}{\mapsto} (\check{\Psi},\, \rho,\, c\ ::\ s)}\ \text{Sassert} \qquad \frac{\rho(x) = \ell \quad \mathtt{unsquash}(\check{\Psi}) = (n, \Psi) \quad \Psi(\ell) = c' \quad \check{\Psi} \leadsto \check{\Psi}'}{(\check{\Psi},\, \rho,\, (\mathtt{call}\ x\, ;\, c) :: s) \overset{U}{\mapsto} (\check{\Psi}',\, \rho,\, (c'\, ;\, \mathtt{assert}\ \bot) :: c :: s)}\ \text{Scall}$$

We use guards in the terminating function pointer assertion:

$$\begin{aligned}
&\check{\Psi} \models \mathsf{funptr}\ \ell\ t\ \llbracket\mathbb{P}\rrbracket\ \llbracket\mathbb{Q}\rrbracket \equiv \\
&\quad \exists c.\ \mathtt{let}\ (n_\Psi, \Psi) = \mathsf{unsquash}(\check{\Psi})\ \mathtt{in}\ \Psi(\ell) = c \wedge \\
&\quad \forall s\ \check{\Psi}'\ n'\ a.\ \check{\Psi} \leadsto^{+} \check{\Psi}' \rightarrow \\
&\qquad (\forall\, \rho.\ (\check{\Psi}', \rho) \models \mathsf{guards}_{n'}\ \mathbb{Q}(a)\ s) \rightarrow \\
&\qquad (\forall\, \rho\, n.\ \ t(\rho) = n \rightarrow \\
&\qquad\quad (\check{\Psi}', \rho) \models \mathsf{guards}_{n+n'}\ \mathbb{P}(a)\ ((c\,;\ \mathtt{assert}\ \bot)\ ::\ s))
\end{aligned} \tag{9}$$

Here, ℓ is a program label, t is a *termination measure* (a partial function from stores to $\mathbb{N}$), and $\mathbb{P}$ and $\mathbb{Q}$ are functions from some type A to assertions. This definition is given in a continuation-oriented style. Whenever we have a stack s which terminates in n steps when $\mathbb{Q}(x)$ is satisfied, then we know that running the function body of ℓ will terminate in $n + n'$ steps whenever $\mathbb{P}(x)$ is satisfied, and where n' is determined by the measure. Thus, funptr captures the specification of a terminating function. Note the premise $\check{\Psi} \leadsto^{+} \check{\Psi}'$; this is one of the microcosts discussed in §6.2. $\check{\Psi}'$ must be strictly more approximate than $\check{\Psi}$ because stepping over a `call` instruction ages the program.

Next we give semantic meaning to the Hoare judgement.

$$\begin{aligned}
&\Gamma,\, R\ \vdash_n\ \{P\}\ c\ \{Q\} \equiv \\
&\quad \forall \check{\Psi}\ n'\ k\ s. \\
&\qquad \check{\Psi} \models \Gamma\ \rightarrow \\
&\qquad (\forall\, \rho.\ (\check{\Psi}, \rho) \models \mathsf{guards}_{n'}\ R\ s) \rightarrow \\
&\qquad (\forall\, \rho.\ (\check{\Psi}, \rho) \models \mathsf{guards}_{n'}\ Q\ (k\ ::\ s)) \rightarrow \\
&\qquad (\forall\, \rho.\ (\check{\Psi}, \rho) \models \mathsf{guards}_{n+n'}\ P\ ((c\,;\ k)\ ::\ s))
\end{aligned} \tag{10}$$

Both the funptr predicate and the Hoare judgment have a similar flavor: assume the postcondition(s) guard the program continuation point(s) and demonstrate that the precondition guards the extended continuation. Naturally, this is no accident. These two definitions interact in interesting ways, as we shall see below. First, funptr appears as a premise of the Hcall rule and directly provides the required guards fact to satisfy the Hoare judgment. Secondly, The Vfull rule shows how we can use the Hoare judgment to establish funptr facts for concrete function definitions.

The final definition we need is that of program verification.

$$\begin{aligned}
&\check{\Psi}' \models \mathsf{approxedof}(\check{\Psi})\ \equiv\ \check{\Psi} \leadsto^{*} \check{\Psi}' \\
&\Psi\ :\ \Gamma\ \equiv\ \forall n.\ \mathsf{approxedof}(\mathsf{squash}\ (n, \Psi)) \wedge \triangleright\Gamma \vdash \Gamma
\end{aligned} \tag{11}$$

What this definition means is that we can prove Γ provided that we assume the program under consideration is some squashed version of Ψ and $\triangleright\Gamma$ (i.e., approximately Γ). The assumption $\triangleright\Gamma$ plays the role of an induction hypothesis and is what allows us to verify recursive functions. The $\mathsf{approxedof}(\check{\Psi})$ predicate means that the current program is approximated from $\check{\Psi}$.

Hoare rules. Now that we have finished our major semantic definitions, we are prepared to prove the rules of the Hoare logic as lemmas. For all of the rules aside from Hcall, the proofs are quite straightforward. One simply uses available premises in an obvious way; only simple manipulations of the definitions are required.

The case for Hcall is more interesting. One of the premises of the Hoare judgment, after unfolding the definition of guards, is that $|\check{\Psi}| \geq n + 1$ (recall that the $n + 1$ comes from the subscript on the Hcall rule). This is sufficient to know that there is some $\check{\Psi}'$ such that $\check{\Psi} \leadsto \check{\Psi}'$. At this point our task is to unpack the definition of funptr (available from the precondition of the Hcall rule) and use it to complete our proof. The first conjunct of funptr tells us that there is some instruction c which implements the function ℓ; this will be the instruction on top of the stack following the call. The second conjunct of funptr will ultimately allow us to discharge our termination obligation. However, we must first fulfill its premises. The first premise $\check{\Psi} \leadsto^{+} \check{\Psi}'$ is easy given $\check{\Psi} \leadsto \check{\Psi}'$. The second, $\forall\rho.\ (\check{\Psi}', \rho) \models \mathsf{guards}_{n'}\ \mathbb{Q}_\ell(a)\ (k\ ::\ s))$, follows from the premise of the Hoare judgment $\forall\rho.\ (\check{\Psi}, \rho) \models \mathsf{guards}_{n'}\ Q\ (k\ ::\ s)$ and the premise from the Hcall rule $\mathsf{closed}(\mathbb{Q}_\ell(a) \Rightarrow Q)$. The final premise, $t(\rho) = n$, is given directly by the corresponding premise of the Hcall rule, $\langle\!|t|\!\rangle = n$. After all this, we have demonstrated that $(\check{\Psi}', \rho) \models \mathsf{guards}_{n+n'}\ \mathbb{P}_\ell(a)\ (c\,;\ \mathtt{assert}\ \bot\ ::\ k\ ::\ s)$, which is sufficient to show the termination of $(\mathtt{call}\ x\,;\ k)\ ::\ s$, the program state just before the call.

Verification Rules. The proofs for all the Hoare rules are actually pretty simple. They come down to little more than manipulations of the definitions. The "magic," such as it is, all happens in the proof of the function verification rule, Vfull. This rule allows one to take Hoare derivations for function bodies and conclude that the corresponding funptr facts hold on a program containing those function bodies. The main idea is that one supplies Φ, a list containing the precondition, postcondition and termination measure for a group of (potentially) mutually-recursive functions. Then, for each function in Φ, one must prove a Hoare derivation of a specific form. The assumptions one is allowed to make are taken from Γ, which contains functions already verified, and from Φ, which allows recursive calls. However, the preconditions in Φ are altered to add a conjunct which strengthens the preconditions by requiring the termination measure to decrease. The return postcondition is, naturally, the postcondition of the function. The precondition is the ordinary function precondition together with the assumption that the termination measure for the initial state is n; this is what connects the strengthened preconditions of the recursive assumptions with the initial state. The linear postcondition is $\bot$, which requires the function body to return rather than "falling of the end." Finally, the Hoare derivation is required to bound the number of function calls by n. This connects the termination measures of the function specifications to their intended semantic meanings.

Whenever one is able to provide such a Hoare derivation for each function in Φ, then one can conclude that each function referenced in Φ actually respects its contract, and the corresponding funptr facts can be conjoined with Γ in the conclusion of the rule.

The soundness proof for the rule is a little delicate, but the main idea is straightforward; the proof goes by (complete) induction on the value of the termination measure. The Hoare derivation is used to complete the argument, using the induction hypothesis to discharge the premises in the definition of the Hoare judgment.

Vsimple, which verifies single recursive functions whose specifications take no parameters, follows as a special case of Vfull.

Total correctness. The final part of the soundness proof worth discussing is a result that connects the results we obtain using our definitions to a more traditional notion of total correctness.

Theorem 2 (Total Correctness)**.** *Suppose* $\Psi\ :\ \Gamma$, *and* $\Gamma\ \vdash\ \mathsf{funptr}\ \ell\ t\ \llbracket\mathbb{P}\rrbracket\ \llbracket\mathbb{Q}\rrbracket$. *Then for all stores* ρ *such that* $t(\rho) = n$, *and* $(\mathtt{squash}(n, \Psi), \rho)$ *satisfies* $\mathbb{P}(a)$ *(for some* a*), executing the function body* $\Psi(\ell)$ *will terminate in a state satisfying* $\mathbb{Q}(a)$*.*

Proof. From $\Psi\ :\ \Gamma$ we obtain $(\mathtt{squash}(n, \Psi), \rho)\ \models\ \Gamma$ via (what amounts to) complete induction on n. We thus obtain $(\mathtt{squash}(n, \Psi), \rho) \models \mathsf{funptr}\ \ell\ t\ \llbracket\mathbb{P}\rrbracket\ \llbracket\mathbb{Q}\rrbracket$.

Now we need to unwind the definition of funptr and use it to establish the eventual goal. We must build some "continuation" stack which will terminate when run with some ρ satisfying $\mathbb{Q}(a)$. This quite easy; we can simply chose the empty stack, i.e. the safely halted state! However, it is more convenient to choose the stack:

$$(\mathtt{assert}\ \mathbb{Q}(a)\,;\ \mathtt{return}\,;\ \mathtt{assert}\ \bot)\ ::\ \mathtt{nil}$$

which asserts the postcondition and immediately returns to the empty stack. This stack is guarded by $\mathbb{Q}(a)$ with 0 function calls.

`Maps.v`	436	finite maps
`FuncListMachine.v`	259	§2, §6, and §7.1
`lemmas.v`	487	utility lemmas
`hoare_total.v`	826	§3 and §7.2
`erase.v`	166	§5 and §7.3
`wp.v`	175	precondition generator
`prgrams.v`	915	§4
total	3,264	

Figure 14. Coq files (including whitespace and comments)

Further, any stack with this sequence at its base represents a computation which, if it halts at all, will halt in a state satisfying $\mathbb{Q}(a)$.

But we know that the program will halt because $\mathbb{P}(a)$ guards the function body with the above stack appended. We must only show that the level of the program is large enough (i.e., $\geq n$). However, we have arranged for this to be the case by squashing Ψ to level n.

Thus the program must halt in a state satisfying $\mathbb{Q}(a)$. □

7.3 Erasure

We have presented two operational semantics, and proved soundness of our logic with respect to the unerased semantics. All that remains to substantiate our claims is to show that the erased semantics and the unerased semantics correspond in the expected way.

What we erase in the translation between the two systems is the embedded assertions and the operational rule which governs them. For simplicity, we retain the assert syntactic form, but make the instruction a nop. Instead of erasing asserts, we replace every predicate in the program with a dummy value (e.g., the unit value).

First we show that whenever it is possible to take an unerased step, then it is possible to take a corresponding erased step.

Lemma 1 (Single-step erasure)**.** *Suppose a program steps in the unserased semantics:* $(\mathsf{squash}(n, \Psi),\ \rho,\ s) \overset{U}{\mapsto} (\check{\Psi}',\ \rho',\ s')$*; then a corresponding step can be taken in the erased semantics* $\mathsf{erase}(\Psi) \vdash (\rho,\ \mathsf{map\ eraseInstr}\ s) \overset{E}{\mapsto} (\rho',\ \mathsf{map\ eraseInstr}\ s')$*, and* $\check{\Psi}' = \mathsf{squash}(n', \Psi)$ *for some* n'*.*

Proof. By cases on the unerased semantics rules. □

Using the single-step lemma, we can show that an unerased halting run has a corresponding erased halting run.

Lemma 2 (Halting erasure)**.** *Suppose* $\mathsf{halts}(\mathsf{squash}(n, \Psi), \rho, s)$*. Then* $\mathsf{halts}^E(\mathsf{erase}(\Psi), \rho, \mathsf{map\ eraseInstr}\ s)$*.*

Proof. By induction on the number of steps taken. □

These two lemmas together with theorem 2 imply theorem 1.

8. Implementation

All the constructions and proofs presented in this paper have been machine checked in Coq. The soundness proof uses the Mechanized Semantic Library, a mechanized semantic toolkit used in a number of different projects [ADH10]. We match our presentation to the proof development in this paper in Figure 14. Mechanized proofs are often extremely long. The core of the development, `hoare_total.v`, is amazingly short at only 826 lines. Here we we provide the central semantic models from §7.2; state and prove all ten H-rules from §3.1 and all three V-rules from §3.2 using those models; and prove the total correctness theorem.

The entire proof development may be found online at:

`http://msl.cs.princeton.edu/termination/`

9. Limitations and Extensions

In order to highlight the interaction of function pointers and semantic assertions, we have pared down HAL to the bare minimum. Naturally, this means we do not consider many other interesting language features such as looping constructs, heap manipulations, lexical variable scopes, or procedure parameters.

Our program logic is somehow simultaneously a bit too weak and a bit too strong. It is too weak in the sense that the upper bound need not be tight, and we make no claims on the lower bound. In addition, it would be very nice if we could relate the precondition to the upper bound so that we could verify, *e.g.*, that a program ran in quadratic time. Our logic is too strong in the sense that the burden of constructing an explicit termination measure may be onerous for someone only concerned with termination. It would be better if one could provide a well-founded relation for each function, hiding the explicit bounds and termination measures under existential binders.

Finally, we have not investigated the completeness of our logic, but have only argued that the logic is *complete enough* to verify interesting programs. Future work may investigate completeness and determine what modifications may be required to achieve it.

10. Related work

Applications of step-indexing and its alternatives. Step-indexing has been used to prove results in type safety [Ahm04], soundness of program logics [HAZ08], and program equivalence [ADR09, DNRB10]. Indirection theory [HDA10] provides clean axioms for step-indexed models. Domain theory is the classic tool for building semantic models [GHK$^+$03]. The price one pays for working in domain theory is the substantial mathematical theory which underlies the discipline. Birkedal *et al.* have a general result which indicates that indirection theory can be constructed using general techniques in certain categories of ultrametric spaces [BRS$^+$].

Predicates in syntax. Semantic assertions often used in program analysis settings such as BoogiePL [BCD$^+$05]. Semantic assertions are one example of a larger class of bookkeeping instructions that embed formulas into program syntax, such as the `makelock` instruction used in concurrent C minor [Hob08].

List machine. HAL is based on the list machine of Appel, Leroy, and Dockins. [ADL10] The list machine was designed to be as simple as possible, while still being minimally interesting from a programming language metatheory and verification standpoint. Compared to the original list machine language, we have added label values (function pointers) and replaced the jump-based control flow system with if/then/else and call/return.

Program logics. Floyd-Hoare logics of total correctness have a venerable lineage going back to the seminal papers by the authors whence we derive the name. [Flo67, Hoa69] Procedures have been long studied in the context of program logics, [Apt81, Old84] but function pointers and higher-order functions have been largely neglected. Schwinghammer et al.'s recent work on "nested" Hoare triples [SBRY09] combines features of separation logic [Rey02] with the ability to reason about "stored code," which behaves quite similar to function pointers. It is a logic of partial correctness.

The work of Honda et al. seems nearest to our own in terms of logical power. [HY04] They provide a logic of total correctness for call-by-value PCF (a λ-calculus with references). The soundness proof goes by a reduction to the π-calculus equipped with a process logic in the rely/guarantee style. [Hon04] Honda et al. do not consider embedded semantic assertions. In later work Honda et al. consider the issue of completeness in this logic. [HBY06]

Aspinall et al. have developed a sound and complete program logic for Grail, a Java subset, which reasons about both correctness and resources. [ABH$^+$07] Although their system includes a form of virtual method invocation, it is not immediately clear if their formalism allows higher-order behaviors.

Termination checkers. Logics of total correctness provide both that a program satisfies its specification and that it terminates. Partial correctness only provides that a program satisfies its specification. Dually, termination checkers focus on showing termination without necessarily demonstrating correctness.

Popular methods for achieving this goal involve discovering termination arguments in the form of ranking functions [CS01] or disjunctive termination arguments [PR04]. Practical tools exists for both user-guided [GTSkF04] and fully automatic [CPR06] synthesis and verification of termination arguments.

11. Conclusion

We have presented a simple language with embedded semantic assertions and functions pointers, together with a logic of total correctness. We have proved our logic sound with respect to a standard operational semantics using step-indexing (in the form of indirection theory), thereby refuting the widely-held belief that step-indexing techniques are not applicable to liveness problems.

Acknowledgments. We thank Andrew W. Appel for his helpful comments and criticisms on earlier versions of this work.

References

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step C minor. In *20th Int'l Conference on Theorem Proving in Higher-Order Logics*, pages 5–21, 2007.

[ABH+07] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. *Theor. Comput. Sci.*, 389(3):411–445, 2007.

[ADH10] Andrew Appel, Robert Dockins, and Aquinas Hobor. Mechanized Semantic Library. Available at http://msl.cs.princeton.edu, 2009–2010.

[ADL10] Andrew W. Appel, Robert Dockins, and Xavier Leroy. A list-machine benchmark for mechanized metatheory. Submitted for publication, 2010.

[ADR09] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *POPL '09: Proc. 36th annual Symposium on Principles of Programming Languages*, pages 340–353, 2009.

[Ahm04] Amal J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, Princeton, NJ, November 2004. Tech Report TR-713-04.

[AMRV07] Andrew W. Appel, Paul-Andre Melliès, Christopher D. Richards, and Jerôme 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, January 2007.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[BCD+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *4th Intl. Symp. on Formal Methods for Components and Objects (FMCO05)*, 2005.

[BRS+] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed kripke models over recursive worlds. Submitted for publication.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI '06: Proc. of the 2006 conference on Programming language design and implementation*, pages 415–426, 2006.

[CS01] Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In *TACAS 2001: Proc. of the 7th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–81. Springer, 2001.

[DNRB10] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful adts. In *POPL '10: Proc. 37th Annual Symp'm on Principles of Programming Languages*, pages 185–198, 2010.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967.

[GHK+03] G. Gierz, K. H. Hofmann, K. Kiemel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*, volume 93 of *Enclyopedia of Mathematics and its Applications*. Cambridge U. Press, Cambridge, UK, 2003.

[GTSkF04] Jürgen Giesl, René Thiemann, Peter Schneider-kamp, and Stephan Falke. Automated termination proofs with aprove. In *In RTA2004: Rewriting Techniques and Applications, volume 3091 of LNCS*, pages 210–220. Springer, 2004.

[HAZ08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. European Symp. on Programming (ESOP 2008) (LNCS 4960)*, pages 353–367. Springer, 2008.

[HBY06] Kohei Honda, Martin Berger, and Nobuko Yoshida. Descriptive and relative completeness of logics for higher-order functions. In *Automata, Languages and Programming*, volume 4052/2006 of *LNCS*, pages 360–371. Springer, 2006.

[HDA10] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In *Proc. 37th Annual ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 171–185, January 2010.

[Hoa69] C A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):578–580, October 1969.

[Hob08] Aquinas Hobor. *Oracle Semanatics*. PhD thesis, Princeton University, Princeton, NJ, November 2008.

[Hon04] Kohei Honda. From process logic to program logic. In *ICFP '04: Proc. of the 9th intl. conference on Functional programming*, pages 163–174, 2004.

[HY04] Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP '04: Proc. of the 6th intl. conference on Principles and practice of declarative programming*, pages 191–202, 2004.

[Old84] Ernst-Rüdiger Olderog. Hoare's logic for programs with procedures — what has been achieved? In *Proc. of the Carnegie Mellon Workshop on Logic of Programs*, pages 383–395. Springer, 1984.

[PR04] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS '04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 32–41, Washington, DC, USA, 2004. IEEE Computer Society.

[Rey02] John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002.

[Ric10] Christopher D. Richards. *The Approximation Modality in Models of Higher-Order Types*. PhD thesis, Princeton University, Princeton, NJ, June 2010.

[SBRY09] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested hoare triples and frame rules for higher-order store. In *CSL'09/EACSL'09: Proc. of the 23rd CSL intl. conference and 18th EACSL Annual conference on Computer science logic*, pages 440–454. Springer, 2009.