Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 286 (2012) 139–155

www.elsevier.com/locate/entcs

Time Bounds for General Function Pointers

Robert Dockins
Princeton University
rdockins@cs.princeton.edu

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

Abstract

We develop a logic of explicit time resource bounds for a language with function pointers and semantic assertions. 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 resource bounding problems, and is the first application of step-indexed models in which the outermost quantifier is existential instead of universal. Our results are machine checked in Coq.

Keywords: Step-indexed models, Termination

1 Introduction

We define a minimal Halting Assert 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 semantic assertions have runtime behavior equivalent to `skip`, they are useful during static analysis (*e.g.*, [5]) and as a mechanism for ensuring that the intermediate states of programs meet set invariants. Semantic assertions may seem benign, but their inclusion in a language with function pointers leads to an unpleasant contravariant circularity. Most domains containing a similar circularity in their semantic models (*e.g.*, concurrency with first-class locks, self-modifying code) are quite complex in ways unrelated to the circularity. We consider HAL to be a test bed for semantic techniques that may be applicable in richer settings in the future.

We design a program logic of explicit time resource bounds for HAL. Programs verified in our logic are guaranteed to satisfy all invariants given in assert statements and are verified against an explicit bound on the number of function calls they make before safely halting. We hope this kind of logic will be applicable to real-time systems, where one is interested in concrete bounds rather than simple termination.

We are unaware of any other logic of resource bounds for languages containing either function pointers or the kind of contravariant circularity present in HAL.

1571-0661 © 2012 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2012.08.010

$\chi(\tau) \equiv x := \ell$	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
$\mid$ `assert` P	semantic assertion, wherein $P : \tau$
$\phi(\tau) \equiv \mathsf{label} \rightharpoonup \chi(\tau)$	parametrized program

Fig. 1. Parameterized commands and programs

We can handle 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`). Our semantic model demonstrates how step indexing can be applied to logics of resource bounds in a compact manner.

Contributions. We design a language containing function pointers and semantic assertions, develop an associated logic of time resource bounds, and apply the logic to example programs. We develop a step-indexed model of the Hoare judgment and prove the logic sound. Our results are checked in Coq.

Associated material. Interested readers can find more details, particularly on the semantic models, in unreviewed previous versions of this work [7]. The Coq development is at `http://msl.cs.princeton.edu/termination/`.

2 An Introduction to HAL

We present HAL commands and programs in Figure 1. Our syntax is parameterized over the type of assertions τ (*i.e.*, the metatype of our parameterized syntax is `Type->Type` instead of `Type`). Most commands are unexciting: load a constant, allocate a fresh pair, project from a pair, sequence, and branch-if-0. 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**—*i.e.*, x is a function pointer. Functions do not take explicit arguments; instead, a programmer must establish an ad-hoc calling convention. The unusual command is the semantic assertion `assert`; here P has the type of the argument τ. A parameterized program $\phi(\tau)$ is a partial function from code labels to parameterized commands.

We give the basic semantic definitions for HAL in Figure 2a. We use natural numbers for program variables (for readability we use r_i instead of i for concrete

variable	$x \equiv \mathbb{N}$
label	$\ell \equiv \mathbb{N}$
value	$v \equiv$ label $+$ (value $\times$ value)
store	$\rho \equiv$ variable $\rightharpoonup$ value
measure	$t \equiv$ store $\rightharpoonup \mathbb{N}$
predicate	$P \approx$ (program $\times$ store) $\to \mathcal{T}$
command	$c \equiv \chi$(predicate)
stack	$s \equiv$ list command
program	$\Psi \equiv$ label $\rightharpoonup$ command

(a) Basic semantic definitions

$\top,\ \bot$	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
$\mu X.\ P$	contravariant equirecursion
$x \Downarrow v$	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
$\langle\!\langle t \rangle\!\rangle$	measure t on the current store
funptr $\ell\ t\ [\mathbb{P}]\ [\mathbb{Q}]$	terminating function pointer

(b) A variety of predicates

Fig. 2. Basic semantic definitions and assertions in our separation logic

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 require 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 parameterized 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 but with the pleasing addition of being sound, is the subject of §5.

We give a variety of predicates in Figure 2b. We have constants ($\top$, $\bot$) and the standard logical connectives ($\wedge$, $\vee$, $\Rightarrow$, $\neg$). 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 predicate itself. We provide a **contravariant-capable** equirecursive μ to describe recursive program invariants as long as the recursion is contractive [9]. 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 simple 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.*,

"$\langle\!|t|\!\rangle < n$"—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 $\langle\!|t|\!\rangle$ are equivalent to $\bot$.

The assertion of particular interest 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. 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 type parameter A is actually part of the function pointer assertion but has been elided for the presentation. When funptr $\ell\ t\ [\mathbb{P}]\ [\mathbb{Q}]$ holds:

(i) The program has code c at address ℓ (recall programs are functions from labels to commands); this is why we need predicates to take programs as arguments.

(ii) When c is called from a context with an initial store ρ, if $t(\rho)$ is defined, then c makes at most $t(\rho)$ function calls before returning to its caller.

(iii) If $t(\rho)$ is defined, then for all a, if $\mathbb{P}(a)$ holds prior to executing c, then $\mathbb{Q}(a)$ will hold when c returns. The parameter a is thus able to relate pre- and postconditions to each other over the function call without auxiliary state.

3 Total Correctness for HAL

Our program logic is divided into two parts. Hoare rules verify commands in HAL; a strength of our approach is that these are natural. Function rules use the verification of a function's body to prove that the function satisfies its specification.

Hoare Rules. Our Hoare judgment, written $\Gamma,\ R \vdash_n \{P\}\ c\ \{Q\}$, where 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 model until §6, but the informal meaning is straightforward. P, c, and Q are the standard precondition, instruction, and postcondition triple 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 funptr assertions in Γ. Finally, starting from precondition P, n is an upper bound on the number of function calls c will execute before it terminates. Our logic is powerful enough that all of these parameters (including the time bound n) can take logical variables instead of concretes; indeed, the second example from §4 demonstrates that we can verify higher-order polymorphic functions independently of their call sites.

We present the Hoare rules for total correctness in Figure 3. 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. For brevity we only show the Hlabel rule; see [7] for Hcons, Hfetch1, and Hfetch2. Since these rules do not make any function calls $n = 0$.

The sequence rule Hseq looks standard; the key point is that the upper bounds on the subcommands c_1 and c_2 are summed for the sequence. For the conditional rule Hif, both c_1 and c_2 must share the same bound n, which is then used for the whole. If the natural bounds differ, one harmonizes them via weakening.

The weakening/consequence rule Hweaken allows covariance in the preconditions

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

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

$$\frac{\Gamma,\, R \;\vdash_n\; \{P_1\}\; c_1\; \{Q\} \qquad \Gamma,\, R \;\vdash_n\; \{P_2\}\; c_2\; \{Q\}}{\Gamma,\, R \;\vdash_n\; \{(x \Downarrow 0 \wedge P_1) \vee ((\neg\,(x \Downarrow 0)) \wedge P_2)\}\; \texttt{ifnil}\; x\; \texttt{then}\; c_1\; \texttt{else}\; c_2\; \{Q\}}\;\text{Hif}$$

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

$$\frac{\Gamma \wedge P \vdash Q}{\Gamma,\, R \;\vdash_0\; \{P\}\; \texttt{assert}\; Q\; \{P\}}\;\text{Hassert} \qquad \frac{}{\Gamma,\, R \;\vdash_0\; \{R\}\; \texttt{return}\; \{\bot\}}\;\text{Hreturn}$$

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

Fig. 3. Hoare rules

$(P,\ P')$ and contravariance in the postconditions $(Q,\ Q')$ and return conditions (R,R'). The function assertions $(\Gamma,\ \Gamma')$ are related covariantly and incorporated in the other entailments in the most general way. We allow the 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 Q. We use $n = 0$ since the `assert` command does not make any function calls. The Hreturn rule 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, the function postcondition $\mathbb{Q}_\ell$, when applied to **the same** a, must imply the postcondition Q **in all stores** (*i.e.*, in particular, in the store after the function call is completed). The metavariable a is chosen to relate the function

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

$$\frac{\begin{array}{c}\Psi \ : \ \Gamma \\ \forall a, n.\ \Big(\big(\Gamma \ \wedge \ \mathsf{funptr}\ \ell\ t\ \big[\lambda a'.\ \mathbb{P}(a')\ \wedge\ \langle\!|t|\!\rangle < n\big]\ \big[\mathbb{Q}\big]\big), \\ \mathbb{Q}(a) \quad \vdash_n \quad \{\mathbb{P}(a)\ \wedge\ \langle\!|t|\!\rangle = n\}\ \ \Psi(\ell)\ \ \{\bot\}\ \Big)\end{array}}{\Psi \ : \ (\Gamma \ \wedge \ \mathsf{funptr}\ \ell\ t\ \big[\mathbb{P}\big]\ \big[\mathbb{Q}\big])}\ \text{Vsimple}$$

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

Fig. 4. Single and mutually recursive function verification

pre- and postconditions to each other over the call. Consider the pair:

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

If we need to know that the invariant $r_{15} \Downarrow (16,(23,42))$ is preserved over the call then we set $a = (r_{15},(16,(23,42)))$. 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 in weakest-precondition style and our predicates include general quantification. Our Coq development defines a precondition generator that computes P from R, n, c, and Q, which we use to cut down on the tedium of mechanically verifying the example programs from §4.

Function Verification. The whole-function rules in Figure 4 form the heart of our program logic. Although the symbol count is daunting, the core idea is natural.

Functions are normally verified one at a time, although mutually recursive function groups are verified as a set. One begins with Vstart, which says that program Ψ has specification $\top$ (*i.e.*, no 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 another Vsimple/Vfull until all of Ψ has been verified.

The Vsimple rule assumes that Ψ already has specification Γ; we wish to add the

specification for the function at ℓ using termination measure t, precondition $\mathbb{P}$, and postcondition $\mathbb{Q}$. The key premise is the second: we must verify, using the H-rules, that for any n and a, the function body $\Psi(\ell)$ meets the specification

$$\ldots,\ \mathbb{Q}(a) \vdash_n \{\mathbb{P}(a)\ \wedge\ \langle\!|t|\!\rangle = n\}\ \Psi(\ell)\ \{\bot\}$$

That is, starting from a state that satisfies $\mathbb{P}(a)$ 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 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:

$$\mathsf{funptr}\ \ell\ t\ \left[\lambda a'.\ \mathbb{P}(a')\ \wedge\ \langle\!|t|\!\rangle < n\right]\ [\mathbb{Q}]$$

That is, the function body $\Psi(\ell)$ can call to other functions specified in Γ as well as recursive calls to itself *as long as the termination measure decreases*.

The Vfull rule generalizes the Vsimple rule in two orthogonal ways. First, Vfull can verify a mutually recursive set of functions. Second, Vfull can verify function specifications where the specifications take *parameters*. The universally-quantified variable b in the Vfull rule represents the specification parameters; b ranges 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 both the recursive assumptions and 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 use this power in the following section.

4 Examples of Verified Programming in HAL

Our logic has three distinctive features: time bounds on recursive function pointers, time bounds on polymorphic function pointers, and semantic assertions. Here we

[1] Even the type of a which connects the pre- and postconditions can depend on the value of b.

$\mathsf{listnat}(0, 0)$

$\mathsf{listnat}(n, v) \to$

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

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

(a) Encoding naturals

```
1  assert (∃n m.  addP(n, m)) ;
2  ifnil r1 then return;
3  else
4    r3 := r1.1 ;
5    r1 := r1.2 ;
6    r2 := (r3, r2) ;
7    r0 := 1 ;  // address of c_add
8    call r0 ;
9    return ;
```

(b) Code c_{add}, loaded at label 1

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

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

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

(c) Pre- and postcondition; termination measure

$$\forall n_1,\ m_1,\ n.$$
$$\Big(\mathsf{funptr}\ 1\ \mathsf{addt}\ \big[\lambda n_2\ m_2.\ \mathsf{addP}(n_2, m_2)\ \wedge\ \langle\!|\mathsf{addt}|\!\rangle < n\big]\ \big[\mathsf{addQ}\big],$$
$$\mathsf{addQ}(n_1, m_1)\ \vdash_n\ \{\mathsf{addP}(n_1, m_1) \wedge \langle\!|\mathsf{addt}|\!\rangle = n\}\ \ \langle\text{code from fig. 5b}\rangle\ \{\bot\}\ \Big)$$

(d) Verification obligation for unary addition (using Vsimple)

Fig. 5. Example 1: unary addition.

cover two examples, the first demonstrating recursive function pointers, and the second demonstrating polymorphic function pointers and the use of a semantic assertion whose truth cannot be checked at run-time. In our Coq development we have examples that combine both simultaneously; also see [7].

Example 1: unary addition. Here we examine a simple recursive function which "adds" two lists representing natural numbers in unary notation (lists terminated by the 0 label). Figure 5a defines the $\mathsf{listnat}$ predicate that relates natural numbers to their unary encoding. The code itself is given in Figure 5b. The idea is that starting from two unary-encoded naturals in registers r_1 and r_2, we strip cons cells from r_1 and add them to r_2 until there are no cells left in r_1, and then return. Line 1 simply asserts the precondition of the function. Line 2 tests if the value in register r_1 is nil; if so, we return. 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. Since the code itself is loaded at label 1, this is a recursive call.

We give the specification in figure 5c. Note that the pre- and postconditions of the addition function are parameterized by the pair of numbers to be added. Recall 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 encodes some natural number.

The addition function is a simple self-recursive function, so we can verify it using the Vsimple rule. The proof obligation that is generated by Vsimple (after some minor simplifications) is shown in Figure 5d, and it is straightforward to use the H-rules of our logic to fulfill this verification obligation proof.

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 a function label with some arguments using a cons cell in r_0. Apply unpacks the cons cell and calls the contained function using the enclosed arguments. We toss in an interesting 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 calling convention. We call functions that adhere to our calling convention "standard". 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.

In the specification for apply (Figure 6) t, P and Q are the *parameters of the specification*; they describe the function that will be called. We need the Vfull rule to verify the apply function, with b ranging 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. **The time bound n in the verification comes from the bound on the input function.** In some later verification, we can instantiate the specification with any function specification already verified. In particular, apply can be applied to itself! This would not be a recursive call, in the traditional sense, but rather a *dynamic* higher-order call.

Termination remains assured due to the way the specifications get "stacked" on top of each other. This stacking of function specifications creates a tree-like structure wherein the leaves 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.

The development also contains an implementation of and verification for the recursive higher-order function map, whose termination argument is the sum of the input function's termination argument applied to each element of the list plus the length of the list (for the recursive calls of map itself); for more details see [7].

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

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

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

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

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

(a) "standard" functions; precondition, postcondition, termination measure for apply

$$R \;\equiv\; \exists v_0'.\ r_0 \Downarrow v_0' \;\wedge\; Q(a)(v_0') \;\wedge\; \mathsf{csregs}(v_1,v_2,v_3,v_4)$$

$\vdash_0 \{r_0 \Downarrow (\ell,v) \wedge P(a)(v) \wedge t(v)+1 = n \wedge \mathsf{stdfun}(\ell,t,P,Q) \wedge \mathsf{csregs}(v_1,v_2,v_3,v_4)\}$

1 $\quad r_5 := r_0.0$;

$\vdash_0 \{r_0 \Downarrow (\ell,v) \;\wedge\; r_5 \Downarrow \ell \;\wedge\; P(a)(v) \;\wedge\; t(v)+1 = n \;\wedge\; \mathsf{stdfun}(\ell,t,P,Q)\}$

2 $\quad r_0 := r_0.1$;

$\vdash_0 \{(r_0 \Downarrow v) \;\wedge\; (r_5 \Downarrow \ell) \;\wedge\; P(a)(v) \;\wedge\; (t(v)+1 = n) \;\wedge\; \mathsf{stdfun}(\ell,t,P,Q)\}$

3 $\quad \mathtt{assert}\ \big(\exists \ell', t', P', Q'.\ (r_5 \Downarrow \ell') \wedge \mathsf{stdfun}(\ell',t',P',Q')\big)$;

$\vdash_0 \{(r_0 \Downarrow v) \;\wedge\; (r_5 \Downarrow \ell) \;\wedge\; P(a)(v) \;\wedge\; (t(v)+1 = n) \;\wedge\; \mathsf{stdfun}(\ell,t,P,Q)\}$

4 $\quad \mathtt{call}\ r_5$;

$\vdash_n \{r_0 \Downarrow v' \wedge Q(a)(v')\}$

5 $\quad \mathtt{return}$;

$\vdash_n \{\bot\}$

(b) code c_{app} for apply with abbreviated Hoare triples, loaded at label 1

$$\forall (t,P,Q)\ (v_1,v_2,v_3,v_4,a)\ n.\quad \Big(\top,\ R\ \vdash_n \{P_{\text{apply}}\}\ \langle\text{code from fig. 6b}\rangle\ \{\bot\}\Big)$$

$$P_{\text{apply}} \;\equiv\; \exists \ell, v_0.\ r_0 \Downarrow (\ell,v_0) \wedge \mathsf{stdfun}(\ell,t,P,Q) \wedge P(a)(v_0)$$
$$\wedge\ \mathsf{csregs}(v_1,v_2,v_3,v_4) \wedge t(v_0)+1 = n$$

(c) Verification obligation for apply (using Vfull, after simplification)

Fig. 6. Example 2: apply

5 Resolving the Circularity in predicates

In this section we resolve the circularity in predicate from Figure 2a. In §6 we build a model for the program logic itself and prove that programs verified in our logic terminate within the correct time bound.

Using Indirection Theory to Stratify Through Syntax. The pseudomodel

of predicates in Figure 2a fits into the pattern $K \approx F((K \times O) \to \mathcal{T})$. In this pattern, F is a covariant functor, O is some kind of "flat data", and K is an object one wishes to model. A cardinality argument shows that there are no solutions in set theory, so we instead build an approximate model using indirection theory [10]. 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:

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

The construction of the *knot* K is similar to the one given in [10, §8] but we have enhanced it so that all predicates inside the knot are *hereditary*, a technical property detailed later. A *squashed program* sq_program is simply a knot; a state is a pair of a sq_program and a store. A 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. The squashed and unsquashed programs are related by two functions $\mathsf{squash} : (\mathbb{N} \times \mathsf{program}) \to \mathsf{sq\_program}$ and $\mathsf{unsquash} : \mathsf{sq\_program} \to (\mathbb{N} \times \mathsf{program})$. The power of indirection theory is that two simple axioms relate squash and unsquash:

$$\begin{array}{rcl} \mathsf{squash}(\mathsf{unsquash}(\check{\Psi})) & = & \check{\Psi} \\ \mathsf{unsquash}(\mathsf{squash}(n, \Psi)) & = & (n, \mathsf{prog\_approx}_n(\Psi)) \end{array} \tag{1}$$

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 $\texttt{assert}(P)$ statements and replacing them with $\texttt{assert}(\mathsf{approx}_n(P))$. The core of the approximation is handled by the $\mathsf{approx}_n(P)$ function:

$$\begin{array}{rcl} |\check{\Psi}| & \equiv & (\mathsf{unsquash}(\check{\Psi})).1 \\ \mathsf{approx}_n(P) & \equiv & \lambda(\check{\Psi}, \rho).\ \begin{cases} P(\check{\Psi}, \rho) & |\check{\Psi}| < n \\ \bot & |\check{\Psi}| \geq n \end{cases} \end{array} \tag{2}$$

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

Consequences of Approximation. What is the cost of losing information during approximation? Ten years after step-indexed models were introduced, the answer is still unclear. Experience has led to an ad-hoc understanding among practitioners of certain *microcosts*—small modifications to the "intuitive" definitions

to accommodate the approximation. Previous work has focused on managing and minimizing these microcosts, *e.g.*, via a Gödel-Löb logic of approximation [14].

The fundamental microcost occurs because approx_n throws away all behavior on squashed programs of greater than or equal to level n. Let P be a predicate contained in (the unsquashing of) a program $\check{\Psi}$ of level n. A consequence of (1) is that P has been approximated to level n—*i.e.*, $P = \mathsf{approx}_n(P)$. What happens if we apply P to a state containing $\check{\Psi}$ itself? A review of (2) proves 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:

$$\check{\Psi} \rightsquigarrow \check{\Psi}' \quad \equiv \quad \mathsf{let}\ (n, \Psi) = \mathsf{unsquash}(\check{\Psi})\ \mathsf{in}\ \big((n > 1)\ \wedge\ (\check{\Psi}' = \mathsf{squash}(n-1, \Psi))\big)$$
$$(\check{\Psi}, \rho) \rightsquigarrow (\check{\Psi}', \rho') \quad \equiv \quad (\rho = \rho')\ \wedge\ (\check{\Psi} \rightsquigarrow \check{\Psi}')$$

That is, we 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 predicate out of a squashed program $\check{\Psi}$, we will approximate $\check{\Psi}$ to $\check{\Psi}'$ before we use P.

Repeated approximation leads to a second microcost. Suppose $\check{\Psi} \models P$ and $\check{\Psi} \rightsquigarrow \check{\Psi}'$. We say P is *hereditary*—stable (or monotonic) as $\check{\Psi}$ is approximated—so that $\check{\Psi}' \models P$; that is, $\mathsf{hereditary}(P) \equiv \forall \check{\Psi}.\, (\check{\Psi} \models P) \to (\check{\Psi} \rightsquigarrow^* \check{\Psi}') \to (\check{\Psi}' \models P)$, where 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; due to approximation, this function will eventually produce only the constant $\bot$. We only consider predicates that are hereditary, and every predicate defined in this paper (except for P_{bad}!) has been proved so.

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 produce the kinds of existential witnesses needed for termination proofs; however, the present work proves otherwise. The practical limitations of step-indexed models remain unknown.

Models for predicates. We define the logical connectives for predicates ($\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $\neg$, $\forall$, $\exists$, μ), and entailment ($P \vdash Q$) by a standard intuitionistic lift over the $\rightsquigarrow^*$ relation as in [9]. We define the modality of approximation ($\triangleright P$), used in the metatheory but not by the end-user, as the boxy operator over $\rightsquigarrow^+$:

$$(\check{\Psi}, \rho) \models \triangleright P \quad \equiv \quad \forall \check{\Psi}'.\, (\check{\Psi} \rightsquigarrow^+ \check{\Psi}') \to \check{\Psi}' \models P$$

The model for the terminating function pointer assertion funptr is complex and is

$$\frac{(\check{\Psi}, \rho) \models P}{(\check{\Psi},\ \rho,\ (\texttt{assert}\ P\,;\ c)\ ::\ s) \mapsto (\check{\Psi},\ \rho,\ c\ ::\ s)}\ \text{Sassert}$$

$$\frac{\rho(x) = \ell \quad \mathsf{unsquash}(\check{\Psi}) = (n, \Psi) \quad \Psi(\ell) = c' \quad \check{\Psi} \rightsquigarrow \check{\Psi}'}{(\check{\Psi},\ \rho,\ (\texttt{call}\ x\,;\ c) :: s) \mapsto (\check{\Psi}',\ \rho,\ (c'\,;\ \texttt{assert}\ \bot) :: c :: s)}\ \text{Scall}$$

Fig. 7. Key rules in operational semantics

developed in §6; the other domain-specific predicates listed in Figure 2b are simply:

$$\begin{array}{llll}
(\check{\Psi}, \rho) & \models & x \Downarrow v & \equiv \quad \rho(x) = v \\
(\check{\Psi}, \rho) & \models & [x \leftarrow v]P & \equiv \quad (\check{\Psi}, [x \leftarrow v]\rho) \models P \\
(\check{\Psi}, \rho) & \models & \mathsf{closed}\ P & \equiv \quad \forall \rho'.\ \ (\check{\Psi}, \rho') \models P \\
(\check{\Psi}, \rho) & \models & \langle\!|t|\!\rangle < n & \equiv \quad t(\rho) < n \quad (\text{etc. } \textit{e.g.}, \text{ for } \langle\!|t|\!\rangle = n)
\end{array}$$

For further discussion on the models for predicates see [7].

6 A Step-indexed Model for Total Correctness

Soundness for our logic means that when a function in a verified program is run in a state satisfying its precondition, it will halt in a state satisfying its postcondition. Our soundness proof follows Appel and Blazy: build a semantic model for assertions; define the meaning of judgments; prove the inference rules of the logic as lemmas; and show that the judgment semantics implies the desired theorem.

Operational semantics. Most of the operational semantics of our language, involving simple data or control-flow, is straightforward. Here, we only highlight the more interesting portions of the operational semantics; for details see [7].

We define a small-step relation $(\check{\Psi},\ \rho,\ s) \mapsto (\check{\Psi}',\ \rho',\ s')$, in which ρ and s stand for local variables and stacks and $\check{\Psi}$ is the squashed program; it is modified as the program runs in a very controlled way. Here is the rule for loading a label:

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

This rule, like most of the instructions in HAL, passes the program $\check{\Psi}$ through without any change. Figure 7 lists the two rules of particular interest. The first is the Sassert rule, which shows how semantic assertions are checked as the program runs. The second is Scall, which shows what happens at function calls. This is the only rule wherein the program is modified: each assertion in the program text is approximated one level down. We must do this approximation so that assertions in the text of the function body will be able to judge the program. If we did not approximate the program at this point, any assertions in the function body would fail, foiling our desired soundness result. Thus, the level of the program $\check{\Psi}$ is an upper bound on the number of calls the program can make before getting stuck.

$$\mathsf{configHalts}_n(\check{\Psi}, \rho, s) \;\equiv\; \exists \check{\Psi}', \rho'.\ (|\check{\Psi}| - |\check{\Psi}'| \leq n) \wedge (\check{\Psi}, \rho, s) \mapsto^* (\check{\Psi}', \rho', \mathtt{nil})$$

$$(\check{\Psi}, \rho) \models \mathsf{halts}_n\ s \;\equiv\; |\check{\Psi}| \geq n \to \mathsf{configHalts}_n(\check{\Psi}, \rho, s)$$

$$\mathsf{guards}_n\ P\ s \;\equiv\; P \Rightarrow \mathsf{halts}_n\ s$$

$$\begin{aligned}\check{\Psi} \models \mathsf{funptr}\ \ell\ t\ [\mathbb{P}]\ [\mathbb{Q}] \;\equiv\; \exists c.\ \ & \mathtt{let}\ (n_\Psi, \Psi) = \mathsf{unsquash}(\check{\Psi})\ \mathtt{in}\ \ \Psi(\ell) = c\ \wedge \\ & \forall\ s, \check{\Psi}', n', a.\ \ (\check{\Psi} \rightsquigarrow^+ \check{\Psi}') \to \\ & (\forall\ \rho.\ (\check{\Psi}', \rho) \models \mathsf{guards}_{n'}\ \mathbb{Q}(a)\ s) \to \\ & (\forall\ \rho\ n.\ \ t(\rho) = n\ \to\ (\check{\Psi}', \rho) \models \mathsf{guards}_{n+n'}\ \mathbb{P}(a)\ ((c;\ \mathtt{assert}\ \bot) :: s))\end{aligned}$$

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

$$\check{\Psi}' \models \mathsf{approxedof}(\check{\Psi}) \;\equiv\; \check{\Psi} \rightsquigarrow^* \check{\Psi}'$$

$$\Psi\ :\ \Gamma \;\equiv\; \forall n.\big((\mathsf{approxedof}(\mathsf{squash}\ (n, \Psi)) \wedge \triangleright\Gamma) \vdash \Gamma\big)$$

Fig. 8. Terminating function pointers; Hoare tuples; whole-program verification

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 (*cf.* Figure 8). 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).

We use guards in the terminating function pointer assertion.Here, ℓ is a program label, t is a measure, and $\mathbb{P}$ and $\mathbb{Q}$ are functions from some type A to assertions. This definition is 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 $\check{\Psi} \rightsquigarrow^+ \check{\Psi}'$; this is one of the microcosts discussed in §5. $\check{\Psi}'$ must be strictly more approximate than $\check{\Psi}$ because stepping over

a `call` instruction ages the program. By design, the funptr predicate and the Hoare judgment are similar: assume the postcondition(s) guard the program continuation point(s) and demonstrate that the precondition guards the extended continuation.

The final definition is program verification $\Psi : \Gamma$. That is, 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 $\check{\Psi}$.

H- and V-rules. Now that we have finished our semantic definitions, we are prepared to prove the rules of the Hoare logic as lemmas. The proofs are straightforward for all of the rules aside from Hcall, which itself is not arduous [7].

The real magic happens in the proof of the function verification rule, Vfull. Vfull converts Hoare derivations for function bodies into the corresponding funptr assertions on programs containing those function bodies. Φ is a list containing the precondition, postcondition and termination measure for a group of mutually recursive functions; for each function in Φ, one proves a particular Hoare derivation. Γ contains the assumptions one is allowed to make and it includes functions already verified and those 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 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 $\bot$; this requires the function body to explicitly return. Finally, the Hoare derivation must bound the number of function calls by n; this connects the termination measures of the function specifications to their semantic meanings. By providing such a Hoare derivation for each function in Φ, one can conclude that each function referenced in Φ respects its contract, and the corresponding funptr facts can be conjoined with Γ in the conclusion of the rule. The proof is by induction on the value of the termination measure; see [7].

Total correctness. The final soundness proof connects our definitions to a more traditional notion of total correctness. Suppose $\Psi : \Gamma$, and $\Gamma \vdash \mathsf{funptr}\ \ell\ t\ \big[\mathbb{P}\big]\ \big[\mathbb{Q}\big]$. 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 $\mathbb{Q}(a)$.

Our core semantic ideas (§6) are compact, requiring only 1,315 lines of Coq.

7 Limitations and Related Work

Limitations. Our program logic is somehow simultaneously too weak and too strong. It is too weak in that the upper bound need not be tight, and we make no claims on the lower bound. Our logic is too strong in 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 existentials.

We can relate the precondition to the upper bound so that we can verify, *e.g.*, that a program runs in polynomial bound (*e.g.*, examine the termination measure `addt` from §4). However, the mechanism to do so is cumbersome; it might be better to allow the user to state termination arguments in the ordinals.

Applications of step-indexing and its alternatives. Step-indexing has been used to prove type safety [2], soundness of program logics [8], and program equivalence [1]. Indirection theory [10] provides clean axioms for step-indexed models. Domain theory is the classic tool for building semantic models. Birkedal *et al.* constructs indirection theory in ultrametric spaces [6].

Predicates in syntax. Semantic assertions are often used in program analysis settings such as BoogiePL [5]. 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 [8].

Program logics with function pointers. Schwinghammer et al.'s recent work on "nested" Hoare triples [15] combines features of separation logic with the ability to reason about "stored code," which is 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 [13]. They provide a logic of total correctness for call-by-value PCF. The soundness proof goes by a reduction to the π-calculus equipped with a process logic in the rely/guarantee style [11]. Honda et al. do not consider embedded semantic assertions or explicit time bounds, but do consider the issue of completeness [12]. Aspinall et al. have developed a sound and complete program logic for Grail, a Java subset, which reasons about both correctness and resources [4]. Their system includes a form of virtual method invocation, but it is not clear if their formalism allows higher-order behaviors.

8 Conclusion

We have presented a simple language with embedded semantic assertions and function pointers, together with a logic of the total correctness of time resource bounds. Our logic is able to reason about terminating function pointers in a very general way, including polymorphic mutually-recursive function groups. We have proved our logic sound with respect to an operational semantics using step-indexing, thus demonstrating that step-indexed models are useful for modeling resource logics.

We thank Andrew W. Appel for his helpful comments; and our support from NSF CNS-0910448, AFOSR FA9550-09-1-0138, and LKY PDF T1 251RES0902.

References

[1] Ahmed, A., D. Dreyer and A. Rossberg, *State-dependent representation independence*, in: *POPL '09: Proc. 36th annual Symposium on Principles of Programming Languages*, 2009, pp. 340–353.

[2] Ahmed, A. J., "Semantics of Types for Mutable State," Ph.D. thesis, Princeton University, Princeton, NJ (2004), tech Report TR-713-04.

[3] Appel, A., R. Dockins and A. Hobor, *Mechanized Semantic Library* (2009–2010), available at http://msl.cs.princeton.edu.

URL http://msl.cs.princeton.edu/

[4] Aspinall, D., L. Beringer, M. Hofmann, H.-W. Loidl and A. Momigliano, *A program logic for resources*, Theor. Comput. Sci. **389** (2007), pp. 411–445.

[5] Barnett, M., B.-Y. E. Chang, R. DeLine, B. Jacobs and K. R. M. Leino, *Boogie: A modular reusable verifier for object-oriented programs*, in: *4th Intl. Symp. on Formal Methods for Components and Objects (FMCO05)*, 2005.

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

[7] Dockins, R. and A. Hobor, *A theory of termination via indirection*, in: A. Ahmed, N. Benton, L. Birkedal and M. Hofmann, editors, *Modelling, Controlling and Reasoning About State*, number 10351 in Dagstuhl Seminar Proceedings (2010).
URL http://drops.dagstuhl.de/opus/volltexte/2010/2805

[8] Hobor, A., "Oracle Semanatics," Ph.D. thesis, Princeton University, Princeton, NJ (2008).

[9] A. Hobor, R. Dockins, and A. Appel. A logical mix of approximation and separation. In *APLAS10*.

[10] Hobor, A., 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)*, 2010, pp. 171–185.

[11] Honda, K., *From process logic to program logic*, in: *ICFP '04: Proc. of the 9th intl. conference on Functional programming*, 2004, pp. 163–174.

[12] Honda, K., M. Berger and N. Yoshida, *Descriptive and relative completeness of logics for higher-order functions*, in: *Automata, Languages and Programming*, LNCS **4052/2006** (2006), pp. 360–371.

[13] Honda, K. and N. 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*, 2004, pp. 191–202.

[14] Richards, C. D., "The Approximation Modality in Models of Higher-Order Types," Ph.D. thesis, Princeton University, Princeton, NJ (2010).

[15] Schwinghammer, J., L. Birkedal, B. Reus and H. 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* (2009), pp. 440–454.