A Typed, Compositional Logic for a Stack-Based Abstract Machine

Nick Benton
Microsoft Research, Cambridge
nick@microsoft.com

June 2005

Technical Report
MSR-TR-2005-84

We define a compositional program logic in the style of Floyd and Hoare for a simple, typed, stack-based abstract machine with unstructured control flow, global variables and mutually recursive procedure calls. Notable features of the logic include a careful treatment of auxiliary variables and quantification and the use of substructural typing to permit local, modular reasoning about program fragments. Semantic soundness is established using an interpretation of types and assertions defined by orthogonality with respect to sets of contexts.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Recent research on language-based techniques in security and certification has led to renewed interest in Floyd-Hoare and VDM-style programming logics, and to much work on type systems and logics for low-level code. Two industrially significant typed intermediate languges have received a great deal of attention: the bytecode of the JVM, used as a target for Java, and the Common Intermediate Language of the CLR, used as a target for languages including $C^\sharp$ and Visual Basic. Both of these intermediate languages are stack-based with control flow expressed using labelled jumps and method calls.

Most research on formalizing the type systems of these intermediate languages [34, 11] has treated the reality of stacks and jumps, though some authors have chosen to work with structured imperative control flow [12] or functional-style applicative expressions [37]. Work on more general specification logics [1, 28, 15] has, however, mostly been done in the context of high-level languages.

Here we present and prove the correctness of a simple logic for directly proving partial correctness assertions on a minimal stack-based machine with jumps and first-order procedure calls. This is rather more complex than traditional Hoare logic for while programs. As well as unstructured control flow, we have to deal with a stack that varies in size and locations that vary in type, which means some care has to be taken to ensure assertions are even well-formed. There are also various kinds of error that are, at least a priori, possible in the dynamic semantics: stack underflow, wild jumps and type errors. We deal with these issues by defining a fairly simple type system that rules out erroneous behaviour, and defining assertions relative to typed programs.

There are also complexities associated with (possibly mutually-recursive) procedure calls, which become especially acute if one wishes to be able to reason locally and modularly, rather than re-analysing bodies at every callsite. We solve these problems using three techniques: firstly, we are very explicit about types, contexts and quantifiers (in particular, we have universally quantified assertions on labels in the context, in the style of Reynolds's specification logic [31]); secondly, we adopt a 'tight' interpretation of store typings, which allows us to use substructural reasoning to adapt assumptions on procedures to their calling context; thirdly, we use a very general rely/guarantee-style rule for linking arbitrary program fragments.

The other novelty of the present work is the semantics with respect to which we prove soundness. Assertions on a program p are interpreted extensionally, using a form of orthogonality (perping) with respect to contexts extending p. A further technical twist in the proofs is the use of step-indexed approximations to the semantics of assertions and their orthogonals.

2 The Machine

We assume the existence of a set $\mathbb{V}$ of names for global variables. The metavariables bop and uop range over typed (binary and unary, respectively) arithmetic and logical operations. Programs are finite partial functions from labels (natural numbers) to instructions:

$$
\begin{array}{rcl}
n & \in & \mathbb{Z} \\
b & \in & \mathbb{B} = \{true, false\} \\
v & \in & \mathit{Values} = \mathbb{Z} \cup \mathbb{B} \\
l & \in & \mathbb{N} \qquad \text{labels} \\
x & \in & \mathbb{V} \qquad \text{variables} \\
\mathit{bop} & \in & \{+, *, -, <, >, =, \wedge, \vee, \ldots\} \\
\mathit{uop} & \in & \{\neg, \ldots\} \\
I & := & \texttt{pushc}\ \underline{v} \mid \texttt{pushv x} \mid \texttt{pop x} \mid \texttt{dup} \mid \texttt{binop}_{\mathit{bop}} \mid \\
 & & \texttt{unop}_{\mathit{uop}} \mid \texttt{brtrue}\ l \mid \texttt{call}\ l \mid \texttt{ret} \mid \texttt{halt} \\
\mathit{Programs} \ni p & := & [l_1 : I_1, \ldots, l_n : I_n]
\end{array}
$$

Stores are finite functions from variable names to values (i.e. to the union of the booleans and the integers, which in this presentation we assume to be disjoint). Our machine will have two kinds of stack: evaluation stacks, σ, used for intermediate results, passing arguments and returning results, are finite sequence of values, whilst control stacks, C, are finite sequence of labels (return addresses):

$$
\begin{array}{rcl}
\mathit{Stores} \ni G & := & x_1 = v_1, \ldots, x_n = v_n \\
\mathit{Stacks} \ni \sigma & := & v_1, \ldots, v_n \\
\mathit{Callstacks} \ni C & := & l_1, \ldots, l_n
\end{array}
$$

We use a comma ',' for both the noncommutative, total concatenation of sequences and for the commutative, partial union of finite maps with disjoint domains. We write a dash '$-$' for both the empty sequence and the empty finite map, and use $|\cdot|$ for the length operation on finite sequences. We define an update operation on stacks by

$$
\begin{array}{rcl}
(\sigma, v')[0 \mapsto v] & = & \sigma, v \\
(\sigma, v')[i+1 \mapsto v] & = & \sigma[i \mapsto v], v'
\end{array}
$$

A configuration is quintuple of a program, a callstack, a global store, an evaluation stack and a label (the program counter):

$$
\mathit{Configs} = \mathit{Programs} \times \mathit{Callstacks} \times \mathit{Stores} \times \mathit{Stacks} \times \mathbb{N}
$$

The operational semantics of our abstract[1] machine is defined by the small-step transition relation $\rightarrow$ on configurations shown in Figure 1.

The `pushc` $\underline{v}$ instruction pushes a constant boolean or integer value v onto the evaluation stack. The `pushv x` instruction pushes the value of the variable x onto the stack. The `pop x` instruction pops the top element off the stack and stores it in the variable x. The $\texttt{binop}_{op}$ instruction pops the top two elements off the stack and pushes the result of applying the binary operator *op* to them, provided their sorts match the signature of the operation. The `brtrue` l instruction pops the top element of the stack and transfers control either to label l if the value was $true$, or to the next instruction if it was $false$. The `halt` instruction halts the execution. The `call` l

[1]According to the terminology of Ager et al.[32], this is should really be called a *virtual* machine, rather than an *abstract* one.

$$
\begin{array}{rcl}
\langle p, l : \texttt{pushc}\ v | C | G | \sigma | l\rangle & \to & \langle p, l : \texttt{pushc}\ v | C | G | \sigma, v | l+1\rangle \\
\langle p, l : \texttt{pushv x} | C | G, x = v | \sigma | l\rangle & \to & \langle p, l : \texttt{pushv x} | C | G, x = v | \sigma, v | l+1\rangle \\
\langle p, l : \texttt{dup} | C | G | \sigma, v | l\rangle & \to & \langle p, l : \texttt{dup} | C | G | \sigma, v, v | l+1\rangle \\
\langle p, l : \texttt{pop x} | C | G, x = v' | \sigma, v | l\rangle & \to & \langle p, l : \texttt{pop x} | C | G, x = v | \sigma | l+1\rangle \\
\langle p, l : \texttt{binop}_{bop} | C | G | \sigma, v_1, v_2 | l\rangle & \to & \langle p, l : \texttt{binop}_{bop} | C | G | \sigma, v_3 | l+1\rangle \\
\text{if } v_3 = (v_1\ bop\ v_2). & & \\
\langle p, l : \texttt{unop}_{uop} | C | G | \sigma, v | l\rangle & \to & \langle p, l : \texttt{unop}_{uop} | C | G | \sigma, v' | l+1\rangle \\
\text{if } v' = uop\ v. & & \\
\langle p, l : \texttt{brtrue}\ l' | C | G | \sigma, true | l\rangle & \to & \langle p, l : \texttt{brtrue}\ l' | C | G | \sigma | l'\rangle \\
\langle p, l : \texttt{brtrue}\ l' | C | G | \sigma, false | l\rangle & \to & \langle p, l : \texttt{brtrue}\ l' | C | G | \sigma | l+1\rangle \\
\langle p, l : \texttt{call}\ l' | C | G | \sigma | l\rangle & \to & \langle p, l : \texttt{call}\ l' | C, l+1 | G | \sigma | l'\rangle \\
\langle p, l : \texttt{ret} | C, l' | G | \sigma | l\rangle & \to & \langle p, l : \texttt{ret} | C | G | \sigma | l'\rangle
\end{array}
$$

Figure 1: Operational Semantics of the Abstract Machine

instruction pushes a return address onto the call stack before transferring control to label l. The `ret` instruction transfers control to a return address popped from the control stack.

For $k \in \mathbb{N}$, we define the k-step transition relation $\to^k$ and the infinite transition predicate $\to^\omega$ in the usual way. We say a configuration is 'safe for k steps' if it either halts within k steps or takes k transitions without error. Formally:

$$
\mathit{Safe}_0\langle p|C|G|\sigma|l\rangle \qquad\qquad \mathit{Safe}_k\langle p, l : \texttt{halt}|C|G|\sigma|l\rangle
$$

$$
\frac{\langle p|C|G|\sigma|l\rangle \to \langle p|C'|G'|\sigma'|l'\rangle \quad \mathit{Safe}_k\langle p|C'|G'|\sigma'|l'\rangle}{\mathit{Safe}_{k+1}\langle p|C|G|\sigma|l\rangle}
$$

and we write $\mathit{Safe}_\omega\langle p|C|G|\sigma|l\rangle$ to mean $\forall k \in \mathbb{N}.\mathit{Safe}_k\langle p|C|G|\sigma|l\rangle$.

Although this semantics appears entirely standard, note that the choice to work with partial stores is significant: execution can get stuck accessing an undefined variable, so, for example, there are contexts which can distinguish the sequence `pushv x;pop x` from a no-op. The following is obvious, assuming each primitive operation is a function:

Lemma 1. *The operational semantics is deterministic.* □

Execution and safety are independent of extensions to the program, to the store, and to either of the stacks at the bottom:

Lemma 2.

1. *If* $\langle p|C|G|\sigma|l\rangle \to^k \langle p|C'|G'|\sigma'|l'\rangle$ *then for any* p', G'', C'' *and* σ''

$$\langle p, p'|C'', C|G'', G|\sigma'', \sigma|l\rangle \to^k \langle p', p|C'', C'|G'', G'|\sigma'', \sigma'|l'\rangle.$$

2. *If* $\mathit{Safe}_k\langle p|C|G|\sigma|l\rangle$ *then for any* p', G', C' *and* σ', $\mathit{Safe}_k\langle p', p|C', C|G', G|\sigma', \sigma|l\rangle$.

□

3 Types and Assertions

As well as divergence and normal termination (hitting a `halt`), programs can get stuck in various ways: type errors applying basic operations, reading or writing undefined variables, underflowing either of the stacks, or jumping or calling outside the program. We will rule out such behaviour using a simple type system, and define assertions relative to those types. This seems natural, but it is worth stressing that it is not the only reasonable way to proceed – although both the JVM and CLR have type systems and type checkers ('verifiers'), the CLR does give a semantics to unverifiable code; such code can be executed if it has been granted sufficient permissions. Our type-based approach prevents one from proving any properties at all of unverifiable code.

3.1 Basic Types and Expressions

We start by defining types for values, stores and stacks:

$$\begin{array}{rcll} \tau & := & \mathtt{int} \mid \mathtt{bool} & \text{base types} \\ \Sigma & := & \tau_1, \ldots, \tau_n & \text{stack types} \\ \Delta & := & x_1 : \tau_1, \ldots, x_n : \tau_n & \text{store types} \end{array}$$

The stack type update $\Sigma[i \mapsto \tau]$ is defined just as for stacks.

We assume a set of *auxiliary variables* ranged over by a. An *auxiliary variable context* Θ is a finite map from auxiliary variables to basic types. A *valuation*, ρ is a function from auxiliary variables to values. We write $\rho : a_1 : \tau_1, \ldots, a_n : \tau_n$ for $\forall 1 \leq i \leq n.\ \rho(a_i) : \tau_i$.

Although our low-level machine does not deal directly with complex expressions, we will use them in forming assertions. The grammar of *expressions* is as follows:

$$E \ := \ \underline{n} \mid \underline{b} \mid x \mid a \mid s(i) \mid E \ \mathit{bop} \ E \mid \mathit{uop} \ E \mid \forall a \in \tau . E$$

As before, the metavariable *bop* ranges over all our binary operators, including arithmetic and logical ones. n and b are integer and boolean constants and x ranges over $\mathbb{V}$. We also have an expression form $s(i)$ for i a natural number, which represents the ith element down the stack. (We could equally well use list notation, with $hd(s)$ and $tl(s)$, instead of indexing stack locations by natural numbers.)

Note that universal quantification over integers and booleans is an expression form. For simplicity, we assume that at least equality and a classical basis set of propositional connectives (e.g. negation and conjunction) are already boolean operators in the language; otherwise we would simply add them to expressions. In any case, we will feel free to use fairly arbitrary first-order arithmetic formulae (including existential quantification and inductively defined predicates) in assertions, regarding them as syntactic sugar for, or standard extensions of, the above grammar.

Expressions are assigned base types in the context of a given stack typing, Σ, store typing, Δ and auxiliary variable context Θ by the following rules:

$$\Theta;\Delta;\Sigma \vdash n : \mathtt{int} \qquad \Theta;\Delta;\Sigma \vdash b : \mathtt{bool}$$

$$\Theta;\Delta, x:\tau;\Sigma \vdash x : \tau \qquad \Theta, a:\tau;\Delta;\Sigma \vdash a : \tau$$

$$\Theta;\Delta;\Sigma,\tau \vdash s(0) : \tau \qquad \frac{\Theta;\Delta;\Sigma \vdash s(i) : \tau}{\Theta;\Delta;\Sigma,\tau' \vdash s(i+1) : \tau}$$

$$\frac{\Theta, a:\tau;\Delta;\Sigma \vdash E : \mathtt{bool}}{\Theta;\Delta;\Sigma \vdash \forall a \in \tau.E : \mathtt{bool}} \qquad \frac{\Theta;\Delta;\Sigma \vdash E : \tau_1 \quad \mathit{uop} : \tau_1 \to \tau_2}{\Theta;\Delta;\Sigma \vdash \mathit{uop}\, E : \tau_2}$$

$$\frac{\Theta;\Delta;\Sigma \vdash E_1 : \tau_1 \quad \Theta;\Delta;\Sigma \vdash E_2 : \tau_2 \quad \mathit{bop} : \tau_1 \times \tau_2 \to \tau_3}{\Theta;\Delta;\Sigma \vdash E_1\, \mathit{bop}\, E_2 : \tau_3}$$

Expression typing satisfies the usual weakening properties, and the definitions of, and typing lemmas concerning, substitutions $E[E'/x]$, $E[E'/a]$ and $E[E'/s(i)]$ are also as one would expect.

Lemma 3.

1. *If* $\Theta, a:\tau';\Delta;\Sigma \vdash E : \tau$ *and* $\Theta;\Delta;\Sigma \vdash E' : \tau'$ *then* $\Theta;\Delta;\Sigma \vdash E[E'/a] : \tau$.
2. *If* $\Theta;\Delta[x \mapsto \tau'];\Sigma \vdash E : \tau$ *and* $\Theta;\Delta;\Sigma \vdash E' : \tau'$ *then* $\Theta;\Delta;\Sigma \vdash E[E'/x] : \tau$.
3. *If* $\Theta;\Delta;\Sigma[i \mapsto \tau'] \vdash E : \tau$ *and* $\Theta;\Delta;\Sigma \vdash E' : \tau'$ *then* $\Theta;\Delta;\Sigma \vdash E[E'/s(i)] : \tau$.

$\square$

There is, however, a mild subtlety in our use of updating/extending for stack and local types compared with the more usual extension used for auxiliary variable contexts. This is because auxiliary variables have explicit binders and are subject to alpha-conversion, whereas global variables and stack locations are not bound in the same way and we therefore have to allow substitution of expressions E', that may contain uses of a variable x at a type τ'', for x of a different type τ' in E.

If $\Theta;\Delta;\Sigma \vdash E : \tau$, $\rho : \Theta$, $G : \Delta$ and $\sigma : \Sigma$ then we define $[\![E]\!]\,\rho\, G\, \sigma \in [\![\tau]\!]$ by:

$$\begin{array}{rcl}
[\![\underline{v}]\!]\,\rho\, G\, \sigma & = & v \\
[\![a]\!]\rho\, G\, \sigma & = & \rho(a) \\
[\![x]\!]\,\rho\, G\, \sigma & = & G(x) \\
[\![s(0)]\!]\,\rho\, G\,(\sigma, v) & = & v \\
[\![s(i+1)]\!]\,\rho\, G\,(\sigma, v) & = & [\![s(i)]\!]\,\rho\, G\, \sigma \\
[\![\underline{uop}\, E]\!]\,\rho\, G\, \sigma & = & \mathit{uop}\,([\![E]\!]\,\rho\, G\, \sigma) \\
[\![E_1\, \underline{bop}\, E_2]\!]\,\rho\, G\, \sigma & = & ([\![E_1]\!]\,\rho\, G\, \sigma)\; \mathit{bop}\; ([\![E_2]\!]\,\rho\, G\, \sigma) \\
[\![\forall a \in \tau.E]\!]\,\rho\, G\, \sigma & = & \bigwedge_{v \in [\![\tau]\!]} [\![E]\!]\,\rho[a \mapsto v]\, G\, \sigma
\end{array}$$

As usual, the semantics is well defined and commutes with each of our three forms of substitution.

Lemma 4. *If $\rho : \Theta$, $G : \Delta$, $\sigma : \Sigma$, and $\Theta; \Delta; \Sigma \vdash E' : \tau'$ then*

1. *If $\Theta, a : \tau'; \Delta; \Sigma \vdash E : \tau$ then*
$$[\![E[E'/a]]\!]\, \rho\, G\, \sigma = [\![E]\!]\, \rho[a \mapsto [\![E']\!]\, \rho\, G\, \sigma]\, G\, \sigma$$

2. *If $\Theta; \Delta[x \mapsto \tau']; \Sigma \vdash E : \tau$ then*
$$[\![E[E'/x]]\!]\, \rho\, G\, \sigma = [\![E]\!]\, \rho\, G[x \mapsto [\![E']\!]\, \rho\, G\, \sigma]\, \sigma).$$

3. *If $\Theta; \Delta, \Sigma[i \mapsto \tau'] \vdash E : \tau$ then*
$$[\![E[E'/s(i)]]\!]\, \rho\, G\, \sigma = [\![E]\!]\, \rho\, G\, \sigma[i \mapsto [\![E']\!]\, \rho\, G\, \sigma].$$

□

If $\Theta; \Delta; \Sigma \vdash E_i : \texttt{bool}$ for $i \in \{1, 2\}$, we write $\Theta; \Sigma; \Delta \models E_1 \implies E_2$ to mean
$$\forall \rho : \Theta. \forall G : \Delta. \forall \sigma : \Sigma.\ [\![E_1]\!]\, \rho\, G\, \sigma \implies [\![E_2]\!]\, \rho\, G\, \sigma$$

We need some syntactic operations which 'reindex' expressions when the stack is pushed or popped. Define $\mathit{shift}(E)$ by

$$\begin{array}{rcl}
\mathit{shift}(s(i)) & = & s(i+1) \\
\mathit{shift}(E_1\ \mathit{bop}\ E_2) & = & \mathit{shift}(E_1)\ \mathit{bop}\ \mathit{shift}(E_2) \\
\mathit{shift}(\mathit{uop}\ E) & = & \mathit{uop}\ (\mathit{shift}(E)) \\
\mathit{shift}(\forall a \in \tau.E) & = & \forall a \in \tau.\mathit{shift}(E) \\
\mathit{shift}(E) & = & E \text{ otherwise}
\end{array}$$

Lemma 5.

1. *If $\Theta; \Delta; \Sigma \vdash E : \tau$ then for any τ', $\Theta; \Delta; (\Sigma, \tau') \vdash \mathit{shift}(E) : \tau$.*

2. *If $\Theta; \Delta; \Sigma \vdash E : \tau$, $\rho : \Theta$, $G : \Delta$ and $\sigma : \Sigma$ then for any v,*
$$[\![E]\!]\, \rho\, G\, \sigma \;=\; [\![\mathit{shift}(E)]\!]\, \rho\, G\, (\sigma, v).$$

□

We also define an operation combining substitution for $s(0)$ with 'unshifting'. If $\Theta; \Delta; \Sigma, \tau' \vdash E : \tau$ and $\Theta; \Delta; \Sigma \vdash E' : \tau'$ then define $E \setminus\!\setminus E'$ as follows:

$$\begin{array}{rcll}
s(0) \setminus\!\setminus E' & = & E' & \\
s(i+1) \setminus\!\setminus E' & = & s(i) & \\
(\forall a \in \tau.E) \setminus\!\setminus E' & = & \forall a \in \tau.(E \setminus\!\setminus E') & \text{capture-avoiding} \\
(E_1\ \mathit{bop}\ E_2) \setminus\!\setminus E' & = & (E_1 \setminus\!\setminus E')\ \mathit{bop}\ (E_2 \setminus\!\setminus E') & \\
E \setminus\!\setminus E' & = & E & \text{otherwise}
\end{array}$$

Lemma 6. *If $\Theta; \Delta; \Sigma, \tau' \vdash E : \tau$ and $\Theta; \Delta; \Sigma \vdash E' : \tau'$ and $\rho \in [\![\Theta]\!]$, $G \in [\![\Delta]\!]$ and $\sigma \in [\![\Sigma]\!]$ then*

1. $\Theta; \Delta; \Sigma \vdash E \setminus\!\setminus E' : \tau$.

2. $[\![E \setminus\!\setminus E']\!]\, \rho\, G\, \sigma = [\![E]\!]\, \rho\, G\, (\sigma, ([\![E']\!]\, \rho\, G\, \sigma))$.

□

3.2 Types and Assertions for Programs

One could first present a type system for programs and then a second inference system for assertion checking defined only over typed programs. However, since the structure of the two inference systems would be very similar, and we require types to be present in defining the assertions, we will combine them into one system.

The type system presented here is simple, monomorphic and somewhat restrictive, being similar in style to that of the JVM. Over this we layer a richer assertion language, including explicit universal quantification. We will define the structure of, and axiomatise an entailment relation on, this assertion language explicitly (rather than delegating both to some ambient higher-order logic).

An *extended label type*, χ, is a universally-quantified pair of a precondition and a postcondition, where the pre- and postconditions each comprise a store type, a stack type and a boolean-valued expression:

$$\chi \;:=\; \Delta; \Sigma; E \to \Delta'; \Sigma'; E' \mid \forall a : \tau . \chi$$

A *label environment* is a finite mapping from labels to extended label types

$$\Gamma \;:=\; l_1 : \chi_1, \ldots, l_n : \chi_n$$

These are subject to the following well-formedness conditions:

$$\frac{\Theta \vdash \chi_1 \; \texttt{ok} \quad \cdots \quad \Theta \vdash \chi_n \; \texttt{ok}}{\Theta \vdash l_1 : \chi_1, \ldots, l_n : \chi_n \; \texttt{ok}} \qquad \frac{\Theta, a : \tau \vdash \chi \; \texttt{ok}}{\Theta \vdash \forall a : \tau . \chi \; \texttt{ok}}$$

$$\frac{\Theta; \Delta; \Sigma \vdash E : \texttt{bool} \quad \Theta; \Delta'; \Sigma' \vdash E' : \texttt{bool}}{\Theta \vdash \Delta; \Sigma; E \to \Delta'; \Sigma'; E' \; \texttt{ok}}$$

The intuitive meaning of

$$l : \Delta; \Sigma; E \to \Delta'; \Sigma'; E'$$

is that if one jumps to l with a store of type Δ and a stack of type Σ, such that E is true, then the program will, without getting stuck, either diverge, `halt`, or reach a `ret` with the callstack unchanged and a store of type Δ' and a stack of type Σ' such that E' is true. We will formalise (a more extensional version of) this intuition in Section 4.

We define $\chi[E/a]$ in the obvious capture-avoiding way and axiomatise an entailment relation on well-formed extended label types as shown in Figure 2. The basic entailment judgement has the form

$$\Theta; \hat{E} \vdash \chi \leq \chi'$$

where $\Theta \vdash \hat{E} : \texttt{bool}$, $\Theta \vdash \chi \; \texttt{ok}$ and $\Theta \vdash \chi' \; \texttt{ok}$. The purpose of $\hat{E}$, which will also show up in the rules of the program logic proper, is to constrain the values taken by the variables in Θ. Including $\hat{E}$ in judgements does not seem necessary for proving properties of closed programs, but we shall see later how it helps us to reason in a modular fashion about program fragments.

The [refl] and [trans] rules just ensure that $\leq$ is a preorder, whislt the quantifier rules express that universal quantification is a meet.

The [$\to$] rule is basically the usual one for subtyping function types, here playing the role of Hoare logic's rule of consequence. The [$\forall\exists \to$] rule is a kind of internalization of the usual left rule for existential quantification. Note how in these two rules,

Order:

$$\frac{\Theta \vdash \chi \ \texttt{ok} \quad \Theta \vdash \hat{E} : \texttt{bool}}{\Theta; \hat{E} \vdash \chi \leq \chi}\ \text{refl} \qquad \frac{\Theta; \hat{E} \vdash \chi \leq \chi' \quad \Theta; \hat{E} \vdash \chi' \leq \chi''}{\Theta; \hat{E} \vdash \chi \leq \chi''}\ \text{trans}$$

Quantifier:

$$\frac{\Theta \vdash \forall a : \tau.\chi \ \texttt{ok} \quad \Theta \vdash E : \tau \quad \Theta \vdash \hat{E} : \texttt{bool}}{\Theta; \hat{E} \vdash \forall a : \tau.\chi \leq \chi[E/a]}\ \forall\text{-subs}$$

$$\frac{\Theta \vdash \chi' \ \texttt{ok} \quad \Theta \vdash \hat{E} : \texttt{bool} \quad \Theta, a : \tau; \hat{E} \vdash \chi' \leq \chi}{\Theta; \hat{E} \vdash \chi' \leq \forall a : \tau.\chi}\ \forall\text{-glb}$$

Arrow:

$$\frac{\Theta; \Delta; \Sigma \vdash F \wedge \hat{E} \implies E \qquad \Theta; \Delta'; \Sigma' \vdash E' \wedge \hat{E} \implies F'}{\Theta; \hat{E} \vdash (\Delta; \Sigma; E \rightarrow \Delta'; \Sigma'; E') \leq (\Delta; \Sigma; F \rightarrow \Delta'; \Sigma'; F')}\ \rightarrow$$

$$\frac{\Theta \vdash \hat{E} : \texttt{bool} \quad \Theta, a : \tau; \Delta; \Sigma \vdash E : \texttt{bool} \quad \Theta; \Delta'; \Sigma' \vdash E' : \texttt{bool}}{\Theta; \hat{E} \vdash \forall a : \tau.(\Delta; \Sigma; E \rightarrow \Delta'; \Sigma', E') \leq (\Delta; \Sigma; \exists a \in \tau.E \rightarrow \Delta'; \Sigma'; E')}\ \forall\exists \rightarrow$$

Frame:

$$\frac{\Theta \vdash \hat{E} : \texttt{bool} \quad \Theta; \overline{\Delta}; \overline{\Sigma} \vdash I : \texttt{bool} \quad \Theta \vdash \Delta; \Sigma; E \rightarrow \Delta'; \Sigma'; E' \ \texttt{ok}}{\begin{array}{ll}\Theta; \hat{E} \vdash & (\Delta; \Sigma; E \rightarrow \Delta'; \Sigma'; E') \\ & \leq (\overline{\Delta}, \Delta; \overline{\Sigma}, \Sigma; \mathit{shift}^{|\Sigma|}(I) \wedge E \rightarrow \overline{\Delta}, \Delta'; \overline{\Sigma}, \Sigma'; \mathit{shift}^{|\Sigma'|}(I) \wedge E')\end{array}}$$

Figure 2: Subtyping/Entailment for Extended Label Types

classical first-order logic, which we do not analyse further, interacts with our more explicit (and inherently intuitionisitic) program logic.

The most complex and interesting case is the frame rule, which is closely related to the rule of the same name in separation logic [25].[2] This allows an invariant I to be added to the pre and postconditions of an extended label type χ, provided that invariant depends only on store and stack locations that are guaranteed to be disjoint from the footprint of the program up to a return to the current top of the callstack. Note how references to stack locations in the invariant are adapted by shifting. The frame rule allows assumptions about procedures to be locally adapted to each call site, which is necessary for modular reasoning. Rather than a single separating conjunction $*$ on assertions, we use our 'tight' (multiplicative) interpretation of state types to ensure separation and use ordinary (additive) conjunction on the assertions themselves.

In use, of course, one needs to adapt extended types to contexts in which there is some relationship between the variables and stack locations mentioned in Δ and Σ and those added in $\overline{\Delta}$ and $\overline{\Sigma}$. This is achieved by using (possibly new) auxiliary variables to split the state dependency before applying the frame rule: see Example 4 in Section 5

[2]Since our rule concerns both 'frame properties' [19] and 'frames' in the sense of activation records, it arguably has even more claim on the name :-)

for a simple example.

3.3 Assigning Extended Types to Programs

Our basic judgement form is $\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma'$ where Γ and Γ' are label environments with disjoint domains, $\hat{E}$ is a boolean-valued expression and p is a program. Γ expresses assumptions about code that will be linked with p, whilst Γ' says what p will guarantee under those assumptions. Thus none of the labels in Γ, and all of the labels in Γ', will be in the domain of p.

The rules for assigning extended types to programs are shown (eliding some obvious well-formedness conditions in a vain attempt to improve readibility) in Figures 3 and 4.

The key structural rule is [link], which allows proved program fragments to be concatenated. The rule has a suspiciously circular nature: if p_1 guarantees Γ_1 under assumptions Γ_2, and p_2 guarantees Γ_2 under assumptions Γ_1, then p_1 linked with p_2 guarantees both Γ_1 and Γ_2 unconditionally. The rule *is*, however, sound for our partial correctness (safety) interpretation, as we shall show in the next section.

The [$\forall$-r] rule is a mild variant of the usual introduction/right rule for universal quantification. The auxiliary variable a does not appear free in Γ, so we may universally quantify it in each (hence the vector notation) of the implictly conjoined conclusions. The vector notation also appears in the [ctxr] rule, allowing global conditions on auxiliary variables to be transferred to the preconditions of each of the conclusions. An equivalent, more conventional, approach would be to state these two rules with a single conclusion and then have a right rule for conjunction:

$$\frac{\Theta; \hat{E}; \Gamma \vdash p : \rhd\Gamma_1 \quad \Theta; \hat{E}; \Gamma \vdash p : \rhd\Gamma_2}{\Theta; \hat{E}; \Gamma \vdash p : \rhd\Gamma_1, \Gamma_2}$$

The only reason for the presentation chosen here is that it threads the subject program fragment p linearly through the derivation, making it clear that we only analyse its internal structure once.

The reader will notice that the axioms fail to cope with branch or call instructions whose target is the instruction itself, as we have said that judgements in which the same label appears on the left and right are ill-formed. This is easily rectified by adding special case rules, but we refrain from doing so here. We also remark that if we are willing to make aggressive use of the frame rule, subtyping and auxiliary variable manipulations, the axioms can be presented in a more stripped-down form. For example, the rule for `ret` can be presented as just

$$-; \mathit{true}; - \vdash [l : \texttt{ret}] \rhd l : -; -; \mathit{true} \to -; -; \mathit{true}$$

$$\frac{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma', l : \chi}{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma'}\ \text{widthr}$$

$$\frac{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma' \quad \Theta \vdash \chi\ \texttt{ok} \quad l \notin dom(p)}{\Theta; \hat{E}; \Gamma, l : \chi \vdash p \rhd \Gamma'}\ \text{widthl}$$

$$\frac{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma', l : \chi \quad \Theta; \hat{E} \vdash \chi \leq \chi'}{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma', l : \chi'}\ \text{subr}$$

$$\frac{\Theta; \hat{E}; \Gamma, l : \chi \vdash p \rhd \Gamma' \quad \Theta \vdash \chi' \leq \chi}{\Theta; \hat{E}; \Gamma, l : \chi' \vdash p \rhd \Gamma'}\ \text{subl}$$

$$\frac{\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma' \quad \Theta, \Theta' \vdash \hat{E}' \implies \hat{E}}{\Theta, \Theta'; \hat{E}' \vdash p \rhd \Gamma'}\ \text{ctxl}$$

$$\frac{\Theta; \hat{E}; \Gamma \vdash p \rhd \vec{l_i} : \vec{\Delta}_i; \vec{\Sigma}_i; \vec{E}_i \rightarrow \vec{\Delta}'_i; \vec{\Sigma}'_i; \vec{E}'_i}{\Theta; true; \Gamma \vdash p \rhd \vec{l_i} : \vec{\Delta}_i; \vec{\Sigma}_i; \hat{E} \wedge \vec{E}_i \rightarrow \vec{\Delta}'_i; \vec{\Sigma}'_i; \vec{E}'_i}\ \text{ctxr}$$

$$\frac{\Theta \vdash \Gamma\ \texttt{ok} \quad \Theta \vdash \hat{E} : \texttt{bool} \quad \Theta, a : \tau; \hat{E}; \Gamma \vdash p \rhd \vec{l_i} : \vec{\chi_i}}{\Theta; \hat{E}; \Gamma \vdash p \rhd \vec{l_i} : \forall a : \tau . \vec{\chi_i}}\ \forall\text{-r}$$

$$\frac{\Theta; \hat{E}; \Gamma, \Gamma_2 \vdash p_1 \rhd \Gamma_1 \quad \Theta; \hat{E}; \Gamma, \Gamma_1 \vdash p_2 \rhd \Gamma_2}{\Theta; \hat{E}; \Gamma \vdash p_1, p_2 \rhd \Gamma_1, \Gamma_2}\ \text{link}$$

Figure 3: Program Logic: Structural and Logical Rules

$$\Theta; \hat{E}; \Gamma \vdash [l : \mathtt{halt}] \rhd l : \chi$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta; \Sigma, \tau; E \to \Delta'; \Sigma'; E' \\ \vdash [l : \mathtt{pushc}\ v] \rhd l : \Delta; \Sigma; E \setminus\setminus v \to \Delta'; \Sigma'; E'\end{array} \quad \text{(where } v : \tau\text{)}$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta, x : \tau; \Sigma, \tau; E \to \Delta'; \Sigma'; E' \\ \vdash [l : \mathtt{pushv\ x}] \rhd l : \Delta, x : \tau; \Sigma; E \setminus\setminus x \to \Delta'; \Sigma'; E'\end{array}$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta, x : \tau; \Sigma; E \to \Delta'; \Sigma'; E' \\ \vdash [l : \mathtt{pop\ x}] \rhd l : \Delta, x : \tau'; \Sigma, \tau; \mathit{shift}(E)[s(0)/x] \to \Delta'; \Sigma'; E'\end{array}$$

$$\Theta; \hat{E}; \Gamma, l+1 : \Delta; \Sigma, \tau, \tau; E \to \Delta'; \Sigma'; E' \vdash [l : \mathtt{dup}] \rhd l : \Delta; \Sigma, \tau; E \setminus\setminus s(0) \to \Delta'; \Sigma'; E'$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta; \Sigma, \tau_3; E \to \Delta'; \Sigma'; E' \\ \vdash [l : \mathtt{binop}_{bop}] \rhd l : \Delta; \Sigma, \tau_1, \tau_2; \mathit{shift}(E)[(s(1)\, bop\, s(0))/s(1)] \to \Delta'; \Sigma'; E' \\ \qquad \text{(where } bop : \tau_1 \times \tau_2 \to \tau_3\text{)}\end{array}$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta; \Sigma, \tau_2; E \to \Delta'; \Sigma'; E' \\ \vdash [l : \mathtt{unop}_{uop}] \rhd l : \Delta; \Sigma, \tau_1; E[(uop\, s(0))/s(0)] \to \Delta'; \Sigma'; E'\end{array} \quad (uop : \tau_1 \to \tau_2)$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta; \Sigma; E \setminus\setminus false \to \Delta'; \Sigma'; E',\ l' : \Delta; \Sigma; E \setminus\setminus true \to \Delta'; \Sigma'; E' \\ \vdash [l : \mathtt{brtrue}\ l'] \rhd l : \Delta; \Sigma, \mathtt{bool}; E \to \Delta'; \Sigma'; E'\end{array}$$

$$\begin{array}{l}\Theta; \hat{E}; \Gamma, l+1 : \Delta''; \Sigma''; E'' \to \Delta'; \Sigma'; E',\ l' : \Delta; \Sigma; E \to \Delta''; \Sigma''; E'' \\ \vdash [l : \mathtt{call}\ l'] \rhd l : \Delta; \Sigma; E \to \Delta'; \Sigma'; E'\end{array}$$

$$\Theta; \hat{E}; \Gamma \vdash [l : \mathtt{ret}] \rhd l : \Delta; \Sigma; E \to \Delta; \Sigma; E$$

Figure 4: Program Logic: Instruction-Specific Axioms

4 Semantics of Types and Assertions

One might formulate and prove a correctness theorem for a logic such as this is syntactically, using a 'preservation and progress' argument. Technically, such an approach has the disadvantage that it would require the extension of our proof rules for extended label types to whole configurations, rather than just programs. The syntactic approach also fails to capture the *meaning* of types and assertions, which we believe is more than a philosophical objection. In practice, we would like to be able safely to link low-level components which have been verified using different proof systems and would arguably also like to have a formal statement of the invariants that *should* be satisfied by trusted-but-unverified components. These goals require a notion of semantics for types and assertions that is independent of the inference system used to assign them to programs.[3]

We will formulate the meaning of types and assertions using a notion of orthogonality with respect to contexts ('perping'). This is a general pattern, related to CPS and linear negation, that has been applied in a number of different operational settings in recent years, including structuring semantics, defining operational versions of admissible predicates, logical relations [27] and ideal models for types [36], and proving strong normalization [18]. To establish the soundness of our link rule we also find it convenient[4] to index our semantic definitions by step-counts, a technical device that Appel and his collaborators have also used extensively in defining semantic interpretations of types over low-level languages [3, 4, 2].

Assume $\Theta; \Delta; \Sigma \vdash E : \texttt{bool}$ and $\rho : \Theta$. We define

$$\mathbb{E}_\rho(\Delta; \Sigma; E) \;\subseteq\; \mathit{Stores} \times \mathit{Stacks} \;\overset{def}{=}\; \{(G, \sigma) \mid G : \Delta \wedge \sigma : \Sigma \wedge [\![E]\!]\,\rho\, G\, \sigma = \mathit{true}\}$$

If $S \subseteq \mathit{Stores} \times \mathit{Stacks}$ and $k \in \mathbb{N}$, we define

$$S_k^\top \subseteq \mathit{Configs} = \{\langle p|C|G'|\sigma'|l\rangle \mid \forall (G,\sigma) \in S. \mathit{Safe}_k \langle p|C|G', G|\sigma', \sigma|l\rangle\}$$

So $S_k^\top$ is the set of configurations that, when extended with any state in S, are safe for k steps: think of these as (k-approximate) test contexts for S.

Lemma 7. *If* $\Theta; \Delta; \Sigma \vdash E \implies E'$ *and* $k \in \mathbb{N}$ *then*

1. $\mathbb{E}_\rho(\Delta; \Sigma; E) \subseteq \mathbb{E}_\rho(\Delta; \Sigma; E')$.
2. $\mathbb{E}_\rho(\Delta; \Sigma; E')_k^\top \subseteq \mathbb{E}_\rho(\Delta; \Sigma; E)_k^\top$.
3. $\mathbb{E}_\rho(\Delta; \Sigma; E)_k^\top \supseteq \mathbb{E}_\rho(\Delta; \Sigma; E)_{k+1}^\top$. □

Now for $\Theta \vdash \Gamma$ ok, $\rho : \Theta$ and $k \in \mathbb{N}$, we define $\models_\rho^k p \rhd \Gamma$ inductively as follows:

$$\models_\rho^k p \rhd l_1 : \chi_1, ..., l_n : \chi_n \iff \textstyle\bigwedge_{i=1}^n . \models_\rho^k p \rhd l_i : \chi_i$$

$$\models_\rho^k p \rhd l : \forall a \in \tau.\chi \iff \forall x \in [\![\tau]\!]. \models_{\rho[a \mapsto x]}^k p \rhd l : \chi$$

$$\begin{aligned}&\models_\rho^k p \rhd l : \Delta; \Sigma; E \to \Delta'; \Sigma'; E' \iff \forall (G, \sigma) \in \mathbb{E}_\rho(\Delta; \Sigma; E). \\ &\quad \forall \langle p, p'|C|G'|\sigma'|l'\rangle \in \mathbb{E}_\rho(\Delta'; \Sigma'; E')_k^\top . \mathit{Safe}_k\langle p, p'|C, l'|G', G|\sigma', \sigma|l\rangle\end{aligned}$$

[3]A syntactic approach to the semantics of program logics can also be excessively intensional, distinguishing observationally equivalent programs in a way that may weaken the logic for applications such as program transformation. Since we are making no claims here about completeness of our logic, we refrain from pushing this argument more strongly.

[4]One can see this as coinduction, or (at least morally) as making explicit the chain of approximations in the construction of a recursive domain that is implicit in the definition of our machine.

The important case is the last one: a program p satisfies $l : \Delta; \Sigma; E \to \Delta'; \Sigma'; E'$ to a k-th approximation if for any k-test context for E' that extends p and has entry point l', if one pushes l' onto the call stack, extends the state with one satisfying E, and commences execution at l, then the overall result is safe for k steps.[5]

We then define the semantics of contextual judgements by

$$\begin{array}{ll} & \Theta; \hat{E}; \Gamma \models p \rhd \Gamma' \\ \iff & \forall \rho : \Theta . \forall k \in \mathbb{N} . \forall p'.\ [\![\hat{E}]\!]\rho = \mathit{true} \wedge \ \models^k_\rho p', p \rhd \Gamma \implies \models^{k+1}_\rho p', p \rhd \Gamma' \end{array}$$

So p satisfies Γ' under assumptions Γ if, for all k, any extension of p that satisfies Γ for k steps satisfies Γ' for $k+1$ steps.

Lemma 8.

1. *If $\Theta \vdash \chi\ ok$ then for any $\rho : \Theta$, p, $l \in dom(p)$, we have $\models^0_\rho p \rhd l : \chi$.*
2. *If $\Theta; \hat{E}; \Gamma \models p \rhd \Gamma'$ then $\Theta; \hat{E}; \Gamma \models p', p \rhd \Gamma'$ for any p' with $dom(p') \cap dom(\Gamma) = \emptyset$.*
3. *$\Theta; \hat{E}; - \models p : \Gamma$ iff $\forall k \in \mathbb{N}. \forall \rho : \Theta. [\![\hat{E}]\!]\rho = \mathit{true} \implies \models^k_\rho p \rhd \Gamma$.*
4. *If $\Theta, a : \tau \vdash \chi\ ok$ and $\Theta \vdash E : \tau$ and $\rho : \Theta$ then*
$$\models^k_{\rho[a \mapsto [\![E]\!]\rho]} p \rhd l : \chi \iff \models^k_\rho p \rhd l : \chi[E/a]$$
5. *If $\Theta \vdash \chi\ ok$ and $\rho : \Theta$ then for any p, l, k, $a \notin \Theta$ and v*
$$\models^k_\rho p \rhd l : \chi \iff \models^k_{\rho[a \mapsto v]} p \rhd l : \chi$$

Proof.

1. All configurations are safe for zero steps.
2. Any extension of (p', p) is an extension of p.
3. Left to right follows from the definition, taking p' to be empty. Right to left follows from the previous part.
4. A simple induction on the structure of χ. The quantifier case requires commuting extensions to ρ, whilst the arrow case uses Lemma 4 to deduce $\mathbb{E}_\rho(\Delta; \Sigma; F[E/a]) = \mathbb{E}_{\rho[a \mapsto [\![E]\!]\rho]}(\Delta; \Sigma; F)$ and the result follows from the semantics of assertions.
5. Another simple induction on χ, relying on weakening for expressions in the arrow case.

□

The following theorem establishes the semantic soundness of the entailment relation on extended label types:

Theorem 1. *If $\Theta; \hat{E} \vdash \chi \leq \chi'$ then for all p, l, $\rho : \Theta$, $k \in \mathbb{N}$*

$$[\![\hat{E}]\!]\rho = \mathit{true} \wedge \models^k_\rho p \rhd l : \chi \implies \models^k_\rho p \rhd l : \chi'$$

[5]It would actually suffice only to ask the context to be safe for $k-1$ steps, rather than k.

Proof. Induction on the rules in Figure 2. The cases [refl] and [trans] are obvious. For [$\forall$-subs] we use Lemma 8 (substitution). The [$\forall$-glb] and [$\forall\exists \rightarrow$] cases are immediate from the definitions and Lemma 8 (weakening). The [$\rightarrow$] case follows from Lemma 7.

For the frame rule, assume $[\![\hat{E}]\!]\rho = true$,

$$\models^k_\rho p \rhd l : \Delta; \Sigma; E \rightarrow \Delta'; \Sigma'; E'$$

and $(G, \sigma) \in \mathbb{E}_\rho(\overline{\Delta}, \Delta; \overline{\Sigma}, \Sigma; \mathit{shift}^{|\Sigma|}(I) \wedge E)$. By the typing assumptions, we can split the stack and the globals into disjoint pieces, $G = G_1, G_2$ and $\sigma = \sigma_1, \sigma_2$ such that $(G_1, \sigma_1) \in \mathbb{E}_\rho(\overline{\Delta}; \overline{\Sigma}; I)$ and $(G_2, \sigma_2) \in \mathbb{E}_\rho(\Delta; \Sigma; E)$. Now assume

$$\langle p, p' | C | G' | \sigma' | l' \rangle \in \mathbb{E}_\rho(\overline{\Delta}, \Delta'; \overline{\Sigma}, \Sigma'; \mathit{shift}^{|\Sigma'|}(I) \wedge E')^\top_k$$

Another splitting argument shows that this implies

$$\langle p, p' | C | G' | G_1 | \sigma', \sigma_1 | l' \rangle \in \mathbb{E}_\rho(\Delta'; \Sigma'; E')^\top_k$$

so by the assumption on p and l

$$\mathit{Safe}_k \langle p, p' | C, l' | G', G_1, G_2 | \sigma', \sigma_1, \sigma_2 | l \rangle$$

as required. □

We can now state and prove the soundness of the program logic rules that were presented in Figures 3 and 4:

Theorem 2. *If* $\Theta; \hat{E}; \Gamma \vdash p \rhd \Gamma'$ *then* $\Theta; \hat{E}; \Gamma \models p \rhd \Gamma'$.

Proof. This follows by rule induction, the previous lemma and the definition of the operational semantics. The soundness of the [width-], [sub-] and [ctxl] rules follows immediately from the interpretation of types and assertions and Theorem 1. The soundness of [$\forall$-r] and [ctxr] is similarly straightforward.

Soundness of [link]. Assume $\Theta; \hat{E}; \Gamma, \Gamma_2 \models p_1 \rhd \Gamma_1$ and $\Theta; \hat{E}; \Gamma, \Gamma_2 \models p_1 \rhd \Gamma_1$. Assume further that for some ρ such that $[\![\hat{E}]\!]\rho = true$ and for some k and p, $\models^k_\rho p, p_1, p_2 \rhd \Gamma$. We want to show

$$\models^{k+1}_\rho p, p_1, p_2 \rhd \Gamma_1, \Gamma_2$$

A simple mathematical induction shows

$$\forall k' \leq k + 1 . (\models^{k'}_\rho p, p_1, p_2 \rhd \Gamma_1) \wedge (\models^{k'}_\rho p, p_1, p_2 \rhd \Gamma_2)$$

from which the desired result is immediate. (The base case is the first part of Lemma 8 and clearly $\forall k' \leq k. \models^{k'}_\rho p, p_1, p_2 \rhd \Gamma$.)

We note that the reason for indexing our safety predicates and contexts by natural numbers k was precisely to make this case of the soundness proof go through.

Soundness of the instruction-specific axioms. We consider each of these in turn. The arguments typically involve using lemmas about the preservation of expression semantics by our various forms of substitution to show a configuration can take one step and reach one assumed to be safe for k steps, establishing that the original configuration is safe for $k + 1$ steps.

halt Nothing to prove by definition of safety.

pushc v Assume $v : \tau$, $\rho : \Theta$ and

$$\models^k_\rho p, l : \mathtt{pushc}\ v \rhd \Gamma, l+1 : \Delta; \Sigma, \tau; E \to \Delta'; \Sigma'; E'$$

and we wish to show

$$\models^{k+1}_\rho p, l : \mathtt{pushc}\ v \rhd l : \Delta; \Sigma; , E \setminus\!\setminus v \to \Delta'; \Sigma'; E'$$

So let $(G, \sigma) \in \mathbb{E}_\rho(\Delta; \Sigma; E \setminus\!\setminus v)$ and

$$\langle p', p, l : \mathtt{pushc}\ v | C | G' | \sigma' | l' \rangle \in \mathbb{E}_\rho(\Delta'; \Sigma'; E')^{\top}_{k+1} \subseteq \mathbb{E}_\rho(\Delta'; \Sigma'; E')^{\top}_{k}$$

Then by the operational semantics

$$\langle p', p, l : \mathtt{pushc}\ v | C, l' | G', G | \sigma', \sigma | l \rangle \to \langle p', p, l : \mathtt{pushc}\ v | C, l' | G', G | \sigma', \sigma, v | l+1 \rangle$$

and since, by Lemma 6, $(G, (\sigma, v)) \in \mathbb{E}_\rho(\Delta; \Sigma, \tau; E)$ we have by assumption on $l+1$

$$\mathit{Safe}_k \langle p', p, l : \mathtt{pushc}\ v | C, l' | G', G | \sigma', \sigma, v | l+1 \rangle$$

so

$$\mathit{Safe}_{k+1} \langle p', p, l : \mathtt{pushc}\ v | C, l' | G', G | \sigma', \sigma | l \rangle$$

and we're done.

pushv x Similar to above.

pop x Assume

$$\models^k_\rho p, l : \mathtt{pop\ x} \rhd \Gamma, l+1 : \Delta, x : \tau; \Sigma; E \to \Delta'; \Sigma'; E'$$

then we want to show

$$\models^{k+1}_\rho p, l : \mathtt{pop\ x} \rhd l : \Delta, x : \tau'; \Sigma, \tau; \mathit{shift}(E)[s(0)/x] \to \Delta'; \Sigma'; E'$$

so let

$$(G, \sigma) \in \mathbb{E}_\rho(\Delta, x : \tau'; \Sigma, \tau; \mathit{shift}(E)[s(0)/x])$$

and

$$(p', p, l : \mathtt{pop\ x}; C; G'; \sigma'; l') \in \mathbb{E}_\rho(\Sigma'; \Delta'; E')^{\top}_{k+1}$$

Then $G = G_1, x = v'$ for some $G_1 : \Delta$ and $v' : \tau'$, and $\sigma = \sigma_1, v$ for some $\sigma_1 : \Sigma$ and $v : \tau$. Furthermore,

$$\begin{aligned} \mathit{true} &= [\![\mathit{shift}(E)[s(0)/x]]\!]\, \rho\, G\, \sigma \\ &= [\![\mathit{shift}(E)[s(0)/x]]\!]\, \rho\, (G_1, x = v')\, (\sigma_1, v) \\ &= [\![\mathit{shift}(E)]\!]\, \rho\, (G_1, x = v)\, (\sigma_1, v) \\ &= [\![E]\!]\, \rho\, (G_1, x = v)\, \sigma_1 \end{aligned}$$

so $((G_1, x = v), \sigma_1) \in \mathbb{E}_\rho(\Delta, x : \tau; \Sigma; E)$ which means

$$\mathit{Safe}_k \langle p', p, l : \mathtt{pop\ x} | C, l' | G', G_1, x = v | \sigma', \sigma_1 | l+1 \rangle$$

and since the operational semantics also gives

$$\begin{aligned} & \langle p', p, l : \mathtt{pop\ x} | C, l' | G', G_1, x = v' | \sigma', \sigma_1, v | l \rangle \\ \to\ & \langle p', p, l : \mathtt{pop\ x} | C, l' | G', G_1, x = v | \sigma', \sigma_1 | l+1 \rangle \end{aligned}$$

we're done.

dup Just like $\texttt{pushc}\ v$, noting by Lemma 6 that

$$[\![E \setminus\!\setminus s(0)]\!]\,\rho\, G\,(\sigma, v) = [\![E]\!]\,\rho\, G\,(\sigma, v, v)$$

binop$_{bop}$ Assume

$$\models^k_\rho p, l : \texttt{binop}_{bop} \rhd \Gamma, l+1 : \Delta; \Sigma, \tau_3; E \to \Delta'; \Sigma'; E'$$

and

$$(G, \sigma) \in \mathbb{E}_\rho(\Delta; \Sigma, \tau_1, \tau_2; \mathit{shift}(E)[(s(0)\, bop\, s(1))/s(1)])$$

so $\sigma = \sigma_1, v_1, v_2$ for some $\sigma : \Sigma$, $v_1 : \tau_1$ and $v_2 : \tau_2$ with

$$\begin{array}{rcl} true & = & [\![\mathit{shift}(E)[(s(0)\, bop\, s(1))/s(1)]]\!]\,\rho\, G\,\sigma \\ & = & [\![\mathit{shift}(E)[(s(0)\, bop\, s(1))/s(1)]]\!]\,\rho\, G\,(\sigma_1, v_1, v_2) \\ & = & [\![\mathit{shift}(E)]\!]\,\rho\, G\,(\sigma_1, v_1\, bop\, v_2, v_2) \\ & = & [\![E]\!]\,\rho\, G\,(\sigma_1, v_1\, bop\, v_2) \end{array}$$

so $(G, (\sigma_1, v_1\, bop\, v_2)) \in \mathbb{E}_\rho(\Delta; \Sigma, \tau_3; E)$ which together with our assumption means that if

$$\langle p', p, l : \texttt{binop}_{bop} | C | G' | \sigma' | l' \rangle \in \mathbb{E}_\rho(\Delta'; \Sigma'; E')^\top_{k+1}$$

then

$$\mathit{Safe}_k \langle p', p, l : \texttt{binop}_{bop} | C, l' | G', G | \sigma', \sigma_1, v_1\, bop\, v_2 | l+1 \rangle$$

and since the operational semantics gives

$$\begin{array}{rl} & \langle p', p, l : \texttt{binop}_{bop} | C, l' | G', G | \sigma', \sigma_1, v_1, v_2 | l \rangle \\ \to & \langle p', p, l : \texttt{binop}_{bop} | C, l' | G', G | \sigma', \sigma_1, v_1\, bop\, v_2 | l+1 \rangle \end{array}$$

we're done.

unop$_{uop}$ Similar to above.

brtrue l' Assume

$$\begin{array}{rl} \models^k_\rho p, l : \texttt{brtrue}\ l' \rhd \Gamma, & l+1 : \Delta; \Sigma; E \setminus\!\setminus \mathit{false} \to \Delta'; \Sigma'; E', \\ & l' : \Delta; \Sigma; E \setminus\!\setminus \mathit{true} \to \Delta'; \Sigma'; E' \end{array}$$

and $(G, \sigma) \in \mathbb{E}_\rho(\Delta; \Sigma, \texttt{bool}; E)$ and

$$\langle p', p, l : \texttt{brtrue}\ l' | C | G' | \sigma' | l'' \rangle \in \mathbb{E}_\rho(\Delta'; \Sigma'; E')^\top_{k+1}$$

Then $\sigma = \sigma_1, b$ for some $\sigma_1 : \Sigma$ and $b : \texttt{bool}$. Without loss of generality, take $b = \mathit{true}$ so the operational semantics gives

$$\begin{array}{rl} & \langle p', p, l : \texttt{brtrue}\ l' | C, l'' | G', G | \sigma', \sigma | l \rangle \\ \to & \langle p', p, l : \texttt{brtrue}\ l' | C, l'' | G', G | \sigma', \sigma_1 | l' \rangle \end{array}$$

and

$$\begin{array}{rcll} \mathit{true} & = & [\![E]\!]\,\rho\, G\,\sigma & \\ & = & [\![E]\!]\,\rho\, G\,(\sigma_1, \mathit{true}) & \\ & = & [\![E]\!]\,\rho\, G\,(\sigma_1, [\![\mathit{true}]\!]\,\rho\, G\,\sigma_1) & \\ & = & [\![E \setminus\!\setminus \mathit{true}]\!]\,\rho\, G\,\sigma_1 & \text{Lemma 6} \end{array}$$

so $(G, \sigma_1) \in \mathbb{E}_\rho(\Delta; \Sigma; E \setminus\setminus true)$ and hence

$$\mathit{Safe}_k\langle p', p, l : \texttt{brtrue}\ l' | C, l'' | G', G | \sigma', \sigma_1 | l' \rangle$$

and we're done.

call l' Assume

$$\models^k_\rho p, l : \texttt{call}\ l' \rhd \Gamma, l+1 : \Delta''; \Sigma''; E'' \to \Delta'; \Sigma'; E',\ l' : \Delta, \Sigma; E \to \Delta''; \Sigma''; E''$$

We wish to show

$$\models^{k+1}_\rho p, l : \texttt{call}\ l' \rhd l : \Delta; \Sigma; E \to \Delta'; \Sigma'; E'$$

Pick

$$\langle p', p, l : \texttt{call}\ l' | C | G' | \sigma' | l'' \rangle \in \mathbb{E}_\rho(\Delta'; \Sigma'; E')^\top_{k+1}$$

then by down-safety and the assumption on $l+1$ we know that for any $(G'', \sigma'') \in \mathbb{E}_\rho(\Delta''; \Sigma''; E'')$ we have $\mathit{Safe}_k\langle p', p, l : \texttt{call}\ l' | C, l'' | G', G'' | \sigma', \sigma'' | l+1 \rangle$ which means

$$\langle p', p, l : \texttt{call}\ l' | C, l'' | G' | \sigma' | l+1 \rangle \in \mathbb{E}_\rho(\Delta''; \Sigma''; E'')^\top_k$$

Therefore, by assumption on l', for any $(G, \sigma) \in \mathbb{E}_\rho(\Delta; \Sigma; E)$,

$$\mathit{Safe}_k\langle p', p, l : \texttt{call}\ l' | C, l'', l+1 | G', G | \sigma', \sigma | l' \rangle$$

Then the operational semantics yields an initial transition

$$\langle p', p, l : \texttt{call}\ l' | C, l'' | G', G | \sigma', \sigma | l \rangle \to \langle p', p, l : \texttt{call}\ l' | C, l'', l+1 | G', G | \sigma', \sigma | l' \rangle$$

so

$$\mathit{Safe}_{k+1}\langle p', p, l : \texttt{call}\ l' | C, l'' | G', G | \sigma', \sigma | l \rangle$$

as required.

ret Immediate from the definitions. □

5 Examples

Our logic is very fine-grained (and the judgement forms fussily baroque), so proofs of any non-trivial example programs are lengthy and extremely tedious to construct by hand. In this section we just present a few micro-examples, demonstrating particular features of the logic. We hope these convince the reader that, given sufficient patience, one can indeed prove all the program properties one might expect (subject to the limitations of the simple type system, of course), and do so in a fairly arbitrarily modular fashion.

Example 1. Consider the simple program fragment

$$[0:\texttt{pushc}\,1,\ 1:\texttt{binop}_+,\ 2:\texttt{ret}]$$

If we write χ_2 for $-;\texttt{int};s(0)=a+1 \to -;\texttt{int};s(0)=a+1$, an instance of the axiom for `ret` is

$$a:\texttt{int};\mathit{true};-\vdash[2:\texttt{ret}]\rhd 2:\chi_2 \tag{1}$$

and an instance of the axiom for `binop` is

$$\begin{array}{l} a:\texttt{int};\mathit{true};2:\chi_2\vdash[1:\texttt{binop}_+]\rhd \\ \ 1:-;\texttt{int},\texttt{int};\mathit{shift}(s(0)=a+1)[s(1)+s(0)/s(1)]\to -;\texttt{int};s(0)=a+1 \end{array}$$

which, expanding the shifting and substitution, is

$$\begin{array}{l} a:\texttt{int};\mathit{true};2:\chi_2\vdash[1:\texttt{binop}_+]\rhd \\ \ 1:-;\texttt{int},\texttt{int};s(1)+s(0)=a+1\to -;\texttt{int};s(0)=a+1 \end{array} \tag{2}$$

Write χ_1 for $-;\texttt{int},\texttt{int};s(1)+s(0)=a+1\to -;\texttt{int};s(0)=a+1$ and apply the [link] rule to (2) and (1) to obtain

$$a:\texttt{int};\mathit{true};-\vdash[1:\texttt{binop}_+,2:\texttt{ret}]\rhd 1:\chi_1,2:\chi_2$$

and use [widthr] to get

$$a:\texttt{int};\mathit{true};-\vdash[1:\texttt{binop}_+,2:\texttt{ret}]\rhd 1:\chi_1 \tag{3}$$

Now an instance of the rule for `pushc` is

$$\begin{array}{l} a:\texttt{int};\mathit{true};1:\chi_1\vdash[0:\texttt{pushc}\ 1]\rhd \\ \ 0:-;\texttt{int};(s(1)+s(0)=a+1)\setminus\!\setminus 1\to -;\texttt{int};s(0)=a+1 \end{array}$$

which is, expanding the unshifting,

$$\begin{array}{l} a:\texttt{int};\mathit{true};1:\chi_1\vdash[0:\texttt{pushc}\ 1]\rhd \\ \ 0:-;\texttt{int};s(0)+1=a+1\to -;\texttt{int};s(0)=a+1 \end{array} \tag{4}$$

Now, since

$$a:\texttt{int};-;\texttt{int}\vdash \mathit{true}\wedge(s(0)=a+1)\implies(s(0)=a+1)$$

and

$$a:\texttt{int};-;\texttt{int}\vdash \mathit{true}\wedge(s(0)=a)\implies(s(0)+1=a+1)$$

we can apply the $[\to]$ subtyping rule to deduce

$$\begin{array}{lcl} a:\texttt{int};\mathit{true}\vdash & & -;\texttt{int};s(0)+1=a+1\to -;\texttt{int};s(0)=a+1 \\ & \leq & -;\texttt{int};s(0)=a\to -;\texttt{int};s(0)=a+1 \end{array}$$

which combines with (4) by [subr] to give

$$\begin{array}{l} a:\texttt{int};\mathit{true};1:\chi_1\vdash[0:\texttt{pushc}\ 1]\rhd \\ \ 0:-;\texttt{int};s(0)=a\to -;\texttt{int};s(0)=a+1 \end{array} \tag{5}$$

Applying [link] to (5) and (3) gives

$$\begin{array}{l} a:\texttt{int};\mathit{true};-\vdash[0:\texttt{pushc}\ 1,\ 1:\texttt{binop}_+,\ 2:\texttt{ret}]\rhd \\ \ 0:-;\texttt{int};s(0)=a\to -;\texttt{int};s(0)=a+1,1:\chi_1 \end{array}$$

to which we can apply [widthr] and [$\forall$-r] to get

$$-; \mathit{true}; - \vdash [0 : \texttt{pushc}\ 1,\ 1 : \texttt{binop}_{+},\ 2 : \texttt{ret}] \rhd 0 : \chi_0 \tag{6}$$

where

$$\chi_0 \ = \ \forall a : \texttt{int}.(-; \texttt{int}; s(0) = a \rightarrow -; \texttt{int}; s(0) = a + 1)$$

which establishes that for any integer a, if we call label 0 with a on the top of the stack, the fragment will either `halt`, diverge or `ret`urn with $a + 1$ on the top of the stack.

Example 2. We now consider the following simple fragment:

$$[10 : \texttt{call}\ 0, 11 : \texttt{br}\ 0]$$

which one may think of as a tail-call optimized client of the code in the previous example. Write χ_0' for

$$\forall c : \texttt{int}.(-; \texttt{int}; (s(0) = c) \wedge ((c = b) \vee (c = b + 1)) \rightarrow -; \texttt{int}; s(0) = c + 1)$$

which is well-formed in the auxiliary variable context $b : \texttt{int}$. We remark in passing that if we had conjunction at the outer level then χ_0' would be equivalent to something like

$$\begin{array}{ll} & (-; \texttt{int}; s(0) = b \rightarrow -; \texttt{int}; s(0) = b + 1) \\ \wedge & (-; \texttt{int}; s(0) = b + 1 \rightarrow -; \texttt{int}; s(0) = b + 2) \end{array}$$

Now, an instance of the axiom for `br` is

$$\begin{array}{l} b : \texttt{int}; 0 : -; \texttt{int}; s(0) = b + 1 \rightarrow -; \texttt{int}; s(0) = b + 2 \\ \vdash [11 : \texttt{br}\ 0] \rhd 11 : -; \texttt{int}; s(0) = b + 1 \rightarrow -; \texttt{int}; s(0) = b + 2 \end{array}$$

And it is easy to verify the following entailment judgement (substituting $b + 1$ for c, and using elementary logic):

$$b : \texttt{int} \vdash \chi_0' \ \leq \ -; \texttt{int}; s(0) = b + 1 \rightarrow -; \texttt{int}; s(0) = b + 2$$

So by [subl]

$$b : \texttt{int}; 0 : \chi_0' \vdash [11 : \texttt{br}\ 0] \rhd 11 : -; \texttt{int}; s(0) = b{+}1 \rightarrow -; \texttt{int}; s(0) = b{+}2 \tag{7}$$

An instance of the axiom for `call` is

$$\begin{array}{l} b : \texttt{int}; 0 : -; \texttt{int}; s(0) = b \rightarrow -; \texttt{int}; s(0) = b + 1, \\ \quad 11 : -; \texttt{int}; s(0) = b + 1 \rightarrow -; \texttt{int}; s(0) = b + 2 \\ \quad \vdash [10 : \texttt{call}\ 0] \rhd 10 : -; \texttt{int}; s(0) = b \rightarrow -; \texttt{int}; s(0) = b + 2 \end{array}$$

whence another use of entailment and [subl] gives

$$\begin{array}{l} b : \texttt{int}; 0 : \chi_0', 11 : -; \texttt{int}; s(0) = b + 1 \rightarrow -; \texttt{int}; s(0) = b + 2 \\ \quad \vdash [10 : \texttt{call}\ 0] \rhd 10 : -; \texttt{int}; s(0) = b \rightarrow -; \texttt{int}; s(0) = b + 2 \end{array} \tag{8}$$

We can now apply [link] to (7) and (8), followed by [widthr] to get

$$\begin{array}{l} b : \texttt{int}; 0 : \chi_0' \vdash \\ \quad [10 : \texttt{call}\ 0, 11 : \texttt{br}\ 0] \rhd 10 : -; \texttt{int}; s(0) = b \rightarrow -; \texttt{int}; s(0) = b + 2 \end{array} \tag{9}$$

Which establishes (roughly) that for any b, if the code at label 0 can be relied upon to compute the successors of b and of $b+1$, then the code at label 10 guarantees to compute $b+2$.

We now consider linking in the code from the first example. Elementary logic and the [$\rightarrow$] entailment rule gives

$$\begin{array}{l} b : \mathtt{int}, a : \mathtt{int} \\ \vdash -; \mathtt{int}; s(0) = a \rightarrow -; \mathtt{int}; s(0) = a + 1 \\ \ \le\ -; \mathtt{int}; s(0) = a \wedge ((a = b) \vee (a = b + 1)) \rightarrow -; \mathtt{int}; s(0) = a + 1 \end{array}$$

and using [$\forall$-subs], [trans] and [$\forall$-glb] we then get $b : \mathtt{int} \vdash \chi_0 \le \chi_0'$, so by [subl] applied to (9)

$$\begin{array}{l} b : \mathtt{int}; 0 : \chi_0 \vdash \\ \ [10 : \mathtt{call}\ 0, 11 : \mathtt{br}\ 0] \rhd 10 : -; \mathtt{int}; s(0) = b \rightarrow -; \mathtt{int}; s(0) = b + 2 \end{array}$$

which we can [link] with a [weak]ened (6) to get

$$\begin{array}{l} b : \mathtt{int}; - \vdash [0 : \mathtt{pushc}\ 1,\ 1 : \mathtt{binop}_+,\ 2 : \mathtt{ret},\ 10 : \mathtt{call}\ 0, 11 : \mathtt{br}\ 0] \rhd \\ \ 0 : \chi_0,\ 10 : -; \mathtt{int}; s(0) = b \rightarrow -; \mathtt{int}; s(0) = b + 2 \end{array}$$

Now applying [$\forall$-r], (followed by a $\forall$-subs and [subr] to remove gratuitous quantification on χ_0) we can conclude

$$\begin{array}{l} -; - \vdash [0 : \mathtt{pushc}\ 1,\ 1 : \mathtt{binop}_+,\ 2 : \mathtt{ret},\ 10 : \mathtt{call}\ 0, 11 : \mathtt{br}\ 0] \rhd \\ \ 0 : \chi_0,\ 10 : \forall b : \mathtt{int}.(-; \mathtt{int}; s(0) = b \rightarrow -; \mathtt{int}; s(0) = b + 2) \end{array}$$

establishing that for any integer b, calling the code at label 10 with b returns with $b+2$. The point about this example is to demonstrate a certain style of modular reasoning: the proof about the code at 10 and 11 was carried out under a rather weak assumption about the code at 0. *After* linking the two fragments together, we were able to generalize and conclude a stronger result about the code at 10 in the composed program *without* reanalysing either code fragment. To re-emphasize this point, we now consider replacing the code at 0 with something weaker:

Example 3. Given the source program

$$\begin{array}{rrl} p = & [0: & \mathtt{dup}, \\ & 1: & \mathtt{pushc}\ 7, \\ & 2: & \mathtt{binop}_<, \\ & 3: & \mathtt{brtrue}\ 5, \\ & 4: & \mathtt{ret}, \\ & 5: & \mathtt{pushc}\ 1, \\ & 6: & \mathtt{binop}_+, \\ & 7: & \mathtt{ret}] \end{array}$$

we can prove, using the rule for conditional branches, that

$$-; \mathit{true} \vdash p \rhd 0 : \forall a : \mathtt{int}.(-; \mathtt{int}; a < 7 \wedge s(0) = a \rightarrow -; \mathtt{int}; s(0) = a+1) \quad (10)$$

showing that the code at label 0 computes the successor of all integers smaller than 7.

We now consider [link]ing the judgement (10) with that we derived for the client program (9). With a few purely logical manipulations (using $\hat{E} = b < 6$) we can derive

$$\begin{aligned}&-; \mathit{true} \vdash p, [10 : \texttt{call}\ 0, 11 : \texttt{br}\ 0]\\ &\rhd 10 : \forall b : \mathit{int}.(-; \texttt{int}; b < 6 \wedge s(0) = b \rightarrow -; \texttt{int}; s(0) = b + 2)\end{aligned}$$

showing that calling 10 now computes $b + 2$ for all b less than 6. Again, we did not reanalyse the client code, but were able to propogate the information about the range over which our 'partial successor' code at 0 works through the combined program *after* linking.

The inclusion of $\hat{E}$, or some equivalent mechanism, seems necessary for this kind of modular reasoning. There needs to be some way to add constraints on auxiliary variables throughout a judgement, as well as to assumptions or conclusions about individual labels.

Example 4. As a simple example of how our entailment relation allows extended types for labels to be adapted for particular calling contexts, consider the assertion χ_0 we had in our first example:

$$\chi_0 \ = \ \forall a : \texttt{int}.(-; \texttt{int}; s(0) = a \rightarrow -; \texttt{int}; s(0) = a + 1)$$

By $\forall$-subs

$$a : \texttt{int}; \mathit{true} \vdash \chi_0 \leq (-; \texttt{int}; s(0) = a \rightarrow -; \texttt{int}; s(0) = a + 1)$$

then, by the frame rule, we can add an invariant on $s(1)$:

$$\begin{aligned}&a : \texttt{int}; \mathit{true}\\ &\vdash \ (-; \texttt{int}; s(0) = a \rightarrow -; \texttt{int}; s(0) = a + 1)\\ &\leq \ (-; \texttt{int}, \texttt{int}; s(1) < a \wedge s(0) = a \rightarrow -; \texttt{int}, \texttt{int}; s(1) < a \wedge s(0) = a + 1)\end{aligned}$$

and by the $\rightarrow$ rule, weakening the postcondition

$$\begin{aligned}&a : \texttt{int}; \mathit{true} \vdash\\ &(-; \texttt{int}, \texttt{int}; s(1) < a \wedge s(0) = a \rightarrow -; \texttt{int}, \texttt{int}; s(1) < a \wedge s(0) = a + 1)\\ &\leq \ (-; \texttt{int}, \texttt{int}; s(1) < a \wedge s(0) = a \rightarrow -; \texttt{int}, \texttt{int}; s(1) < s(0))\end{aligned}$$

Hence, by transitivity

$$\begin{aligned}&a : \texttt{int}; \mathit{true} \ \vdash\\ &\chi_0 \ \leq \ (-; \texttt{int}, \texttt{int}; s(1) < a \wedge s(0) = a \rightarrow -; \texttt{int}, \texttt{int}; s(1) < s(0))\end{aligned}$$

so we can apply [$\forall$-glb] to get

$$\begin{aligned}&-; \mathit{true} \ \vdash\\ &\chi_0 \ \leq \ (-; \texttt{int}, \texttt{int}; \forall a : \texttt{int}.(s(1) < a \wedge s(0) = a \rightarrow -; \texttt{int}, \texttt{int}; s(1) < s(0)))\end{aligned}$$

and then [$\forall\exists \rightarrow$] and another transitivity gives

$$\begin{aligned}&-; \mathit{true} \vdash\\ &\chi_0 \leq (-; \texttt{int}, \texttt{int}; (\exists a \in \texttt{int}.s(1) < a \wedge s(0) = a) \rightarrow -; \texttt{int}, \texttt{int}; s(1) < s(0)))\end{aligned}$$

and then the $\rightarrow$ rule (equivalent precondition) gives

$$\begin{array}{l} -; \mathit{true} \vdash \\ (-; \mathtt{int}, \mathtt{int}; (\exists a \in \mathtt{int}.s(1) < a \wedge s(0) = a) \rightarrow -; \mathtt{int}, \mathtt{int}; s(1) < s(0)) \\ \leq \\ (-; \mathtt{int}, \mathtt{int}; (s(1) < s(0) \rightarrow -; \mathtt{int}, \mathtt{int}; s(1) < s(0)) \end{array}$$

whence transitivity yields

$$-; \mathit{true} \vdash \chi_0 \leq (-; \mathtt{int}, \mathtt{int}; (s(1) < s(0) \rightarrow -; \mathtt{int}, \mathtt{int}; s(1) < s(0))$$

Hence, although χ_0 only mentions a one-element stack, when one calls a label assumed to satisfy χ_0 one can locally adapt that assumption to the situation where are two things on the stack *and* a non-trivial relationship between them. The pattern used here is typical: we use the frame rule and new auxiliary variables to add a separated invariant and then existentially quantify the new variables away.

Example 5. Consider the source procedure

```
void f() {
  x := 0;
  while(x<5) {
    x := x+1;
  }
}
```

A typical Java or $C^\sharp$ compiler will compile the loop with the test and conditional backwards branch at the end, preceded by a header which branches unconditionally into the loop to execute the test the first time. This yields code something like

$$\begin{array}{rrl} p = & [1: & \mathtt{pushc}\, 0, \\ & 2: & \mathtt{pop}\, \mathtt{x}, \\ & 3: & \mathtt{pushc}\, \mathit{true}, \\ & 4: & \mathtt{brtrue}\, 9, \\ & 5: & \mathtt{pushv}\, \mathtt{x}, \\ & 6: & \mathtt{pushc}\, 1, \\ & 7: & \mathtt{binop}_{+}, \\ & 8: & \mathtt{pop}\, \mathtt{x}, \\ & 9: & \mathtt{pushv}\, \mathtt{x}, \\ & 10: & \mathtt{pushc}\, 5, \\ & 11: & \mathtt{binop}_{<}, \\ & 12: & \mathtt{brtrue}\, 5, \\ & 13: & \mathtt{ret}] \end{array}$$

The presence of such unstructured control-flow makes no difference to reasoning in our low-level logic, and we can easily derive

$$-; \mathit{true}; - \vdash p \rhd 0 : x : \mathtt{int}; -; \mathit{true} \rightarrow x : \mathtt{int}; -; x = 5$$

just as one would expect.

Example 6. Although our machine has call and return instructions, it does not specify any particular calling convention or even delimit entry points of procedures. Both the machine and the logic can deal with differing calling conventions and multiple entry points. For example, given

$$\begin{array}{rcrl} p & = & [1: & \texttt{pushv}\,\texttt{x}, \\ & & 2: & \texttt{pushc}\,1, \\ & & 3: & \texttt{binop}_{+}, \\ & & 4: & \texttt{ret}] \end{array}$$

we can derive

$$\begin{array}{l} -; \mathit{true}; - \vdash p \rhd \\ \quad 1 : \forall a : \texttt{int}.(x : \texttt{int}; -; x = a \rightarrow x : \texttt{int}; \texttt{int}; x = a \wedge s(0) = a + 1), \\ \quad 2 : \forall a : \texttt{int}.(-; \texttt{int}; s(0) = a \rightarrow -; \texttt{int}; s(0) = a + 1) \end{array}$$

so one can either pass a parameter in the variable x, calling address 1, or on the top of the stack, calling address 2.

Example 7. Here's a simple example of mutual recursion, as might arise from the source program

```
fun even x = if x=0 then true else odd (x-1)
and  odd x = if x=0 then false else even (x-1)
```

If we write

$$\chi_e = \forall a : \texttt{int}.(-; \texttt{int}; s(0) = a \rightarrow -; \texttt{bool}; s(0) = \mathit{even}(a))$$

and

$$\chi_o = \forall a : \texttt{int}.(-; \texttt{int}; s(0) = a \rightarrow -; \texttt{bool}; s(0) = \mathit{odd}(a))$$

then if

$$\begin{array}{rcrl} p_e & = & [1: & \texttt{dup}, \\ & & 2: & \texttt{pushc}\,0, \\ & & 3: & \texttt{binop}_{=}, \\ & & 4: & \texttt{brtrue}\,9, \\ & & 5: & \texttt{pushc}\,1, \\ & & 6: & \texttt{binop}_{-}, \\ & & 7: & \texttt{call}\,11, \\ & & 8: & \texttt{ret}, \\ & & 9: & \texttt{pushc}\,\mathit{true}, \\ & & 10: & \texttt{ret}] \end{array}$$

we can derive

$$-; \mathit{true}; 11 : \chi_o \vdash p_e \rhd 1 : \chi_e$$

and similarly, given

$$\begin{array}{rrl} p_o & = & [11: \ \texttt{dup}, \\ & & \ 12: \ \texttt{pushc}\, 0, \\ & & \ 13: \ \texttt{binop}_{=}, \\ & & \ 14: \ \texttt{brtrue}\, 19, \\ & & \ 15: \ \texttt{pushc}\, 1, \\ & & \ 16: \ \texttt{binop}_{-}, \\ & & \ 17: \ \texttt{call}\, 11, \\ & & \ 18: \ \texttt{ret}, \\ & & \ 19: \ \texttt{pushc}\, false, \\ & & \ 20: \ \texttt{ret}] \end{array}$$

we can derive

$$-; true; 1 : \chi_e \vdash p_o \rhd 11 : \chi_o$$

and then we can [link] the two functions together to deduce

$$-; true; - \vdash p_e, p_o \rhd 1 : \chi_e, 11 : \chi_o$$

These derivations also work if we optimise the tail call at 7 and/or 11.

6 Discussion

We have presented a typed program logic for a simple stack-based intermediate language, bearing roughly the same relationship to Java bytecode or CIL that a language of while-programs with procedures does to Java or $C^\sharp$.

The contributions of this work include the modular treatment of program fragments and linking (similar to, for example, [10]); the explicit treatment of different kinds of contexts and quantification; the interplay between the prescriptive, tight interpretation of types and the descriptive interpretation of expressions, leading to a separation-logic style treatment of adaptation; the use of shifting to reindex assertions; dealing with non-trivially unstructured control flow (including multiple entry points to mutually-recursive procedures) and an indexed semantic model based on perping.

There is some related work on logics for bytecode programs. Borgström [9] has approached the problem of proving bytecode programs meet specifications by first decompiling them into higher-level structured code and then reasoning in standard Floyd-Hoare logic. Quigley [29, 30] has formalized rules for Hoare-like reasoning about a small subset of Java bytecode within Isabelle, but her treatment is based on trying to rediscover high-level control structures (such as while loops); this leads to rules which are both complex and rather weak. More recently, Bannwart and Müller have combined [6] the simple logic of an early draft of the present paper [7] with a higher-level, more traditional Hoare logic for Java to obtain a rather different logic for bytecodes than that we present here. We should also mention the work of Aspinall et al on a VDM-like logic for resource verification of a JVM-like language [5].

Even for high-level languages, satisfactory accounts of auxiliary variables and rules for adaptation in Hoare logics for languages with procedures seem to be surprisingly recent, see for example the work of Kleymann [17] and Oheimb & Nipkow [35, 26]. Our fussiness about contexts and use of substructural ideas is rather different from those works, however, leading to a rather elegant account of invariants of procedure calls and a complete absence of side-conditions.

Another line of very closely related research is that on proof-carrying code [24, 23] and typed assembly languages [22]. Much of that work has a similar 'logical' flavour to this, with substructural logics often being used to reason about stacks [16], heaps [20] and aliasing [33]. Our RISC-like approach to a stack-based low-level machine, with no built-in notion of procedure entry points or calling conventions, is similar to that of STAL [21]. Compared with most of these projects, we have considered a much simpler machine (no pointer manipulation, dynamic allocation or code pointers), but we go beyond simple syntactic type soundness to give a rather richer program logic with a semantic interpretation. As we have already mentioned, the technique of step-indexed approximations comes from the work of the Appel and his collaborators on foundational proof-carrying code (FPCC). Shao and Hamid [13] have considered interfacing Hoare logic with a syntactic type system for low-level code as a way of verifying linkage between typed assembly language programs and modules verified using a different system – this is similar to the motivation we gave earlier for using a semantic interpretation of assertions.

Clearly, this kind of logic cries out for (at least partial) automation, and this is something we hope to investigate in the near future. A related technical, rather than engineering, point is that it is arguable that one should not even bother to define the structural rules for handling auxiliary variables, quantification, etc. the way we have done here, but rather simply inherit them from a shallow embedding of the semantics in an ambient (machine-checked) higher order logic. Much of the recent related work we have cited takes this approach, which certainly has a great deal to recommend it. We prefer a more explicit approach, at least for preliminary investigations with toy calculi, firstly because it avoids potential subtleties being missed (punting entailment in first-order arithmetic off to an auxiliary system seems qualitatively different from doing the same with contextual equivalence for an interesting language), secondly because it aids understanding, and finally because it may eventually help in designing logic-specific inference algorithms.

There are many variations and improvements one can make, such as treating halting states differently, adding subtyping and polymorphism and experimenting with other connectives at the program logic level. At present we have good power and modularity at the level of assertions, but the type system is rather weak and inflexible. But it must be admitted that in some ways this system represents a rather odd point in the design space, as we have tried to keep the 'spirit' of traditional Hoare logic: pre and post conditions, first-order procedures and the use of classical predicate calculus to form assertions on a flat state. Whilst a generalisation to a higher-order machine, in which code pointers are first class, would bring some complexity (entailment between state assertions becomes more complex), it also seems to offer some significant simplifications: everything is in continuation-passing style, so one only needs preconditions, for example. The other natural extension is to treat more general dynamic heap allocation. Both first-class code pointers and heap data structures have been the objects of a great amount of closely related work on semantics and types of both low-level and high-level languages (e.g. [8] and the references therein); transferring some of those ideas to the world of general assertions on low-level code looks eminently doable. Another direction for generalization is to move from predicates to binary relations on states. Our ultimate goal is a relational logic for a low-level language into which one can translate a variety of high-level typed languages whilst preserving equational reasoning. We regard this system as a step towards that goal, rather than an endpoint in its own right.

We have not yet fully explored the properties and ramifications of our semantic interpretation. One of its effects is to close the interpretation of extended label types with respect to a contextual equivalence, which is a pleasant feature, but our inference system then seems unlikely to be complete. The links with recent work of Honda, Yoshida and Berger on observational equivalence and program logics for state and higher-order functions (e.g. [14]) need further investigation. Note that the extent to which our extensional semantics entails a more naive intensional one depends on what test contexts one can write, and that these test contexts are not merely allowed to be untypable, but interesting ones all *are* untypeable: they 'go wrong' when a predicate fails to hold. There is an adjoint 'perping' operation that maps sets of configurations to subsets of $\mathit{Stores} \times \mathit{Stacks}$ and more inference rules (for example, involving conjunction) seem to be valid for state assertions that are closed, in the sense that $[\![E]\!] = [\![E]\!]^{\top\top}$. We could impose this closure by definition, or by moving away from classical logic for defining the basic assertions over states.

References

[1] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Proceedings of 7th International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, 1997.

[2] A. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[3] A. Appel and A. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th POPL*, 2000.

[4] A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5), September 2001.

[5] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and Alberto Momigliano. A program logic for resource verification. In *Proceedings of 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs2004)*, Lecture Notes in Computer Science. Springer-Verlag, September 2004.

[6] F. Bannwart and P. Muller. A program logic for bytecode. In *Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, April 2005.

[7] N. Benton. A typed logic for stacks and jumps. Draft Note, March 2004.

[8] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA*, 2005.

[9] J. Borgström. Translation of smart card applications for formal verification. Masters Thesis, SICS, Sweden, 2002.

[10] L. Cardelli. Program fragments, linking, and modularization. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997.

[11] S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 1998.

[12] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL)*, 2001.

[13] N. A. Hamid and Z. Shao. Interfacing hoare logic and type systems for foundational proof-carrying code. In *Proc. 17th International Conference on the Applications of Higher Order Logic Theorem Proving (TPHOLs'04)*, volume 3223 of *Lecture Notes in Computer Science*, September 2004.

[14] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *LICS*, 2005.

[15] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *3rd International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 284–303, 2000.

[16] L. Jia, F. Spalding, D. Walker, and N. Glew. Certifying compilation for a language with stack allocation. In *IEEE Symposium on Logic in Computer Science*, June 2005.

[17] T. Kleymann. Hoare logic and auxiliary variables. Technical Report ECS-LFCS-98-399, LFCS, University of Edinburgh, 1998.

[18] S. Lindley and I. Stark. Reducibility and $\top\top$ lifting for computation types. In *Proceedings of TLCA'05*, April 2005.

[19] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In D. Michie and B. Meltzer, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[20] G. Morrisett, A. Amal, and M. Fluet. L3: A linear language with locations. In *Seventh International Conference on Typed Lambda Calculi and Applications*, April 2005.

[21] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal on Functional Programming*, 12(1), January 2002.

[22] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21, 1999.

[23] G. Necula. Proof-carrying code. In *POPL*, 1997.

[24] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[25] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL)*, 2001.

[26] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In *Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *LNCS*. Springer, 2002.

[27] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.

[28] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *European Symposium on Programming (ESOP)*, 1999.

[29] C. Quigley. A programming logic for Java bytecode programs. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2003.

[30] C. L. Quigley. *A Programming Logic for Java Bytecode Programs*. PhD thesis, University of Glasgow, Department of Computing Science, January 2004.

[31] J. C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.

[32] M. Sig Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, 2003.

[33] F Smith, D Walker, and G Morrisett. Alias types. In *European Symposium on Programming (ESOP)*, 2000.

[34] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[35] D. von Oheimb. Hoare logic for mutual recursion and local variables. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *LNCS*. Springer, 1999.

[36] J. Vouillon and P.-A. Mellies. Semantic types: A fresh look at the ideal model for types. In *Proceedings of the 31st POPL*, 2004.

[37] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET common language runtime. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*, 2004.