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

Nick Benton

Microsoft Research, Cambridge
nick@microsoft.com

Abstract. 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.

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 [33, 12] has treated the reality of stacks and jumps, though some authors have chosen to work with structured imperative control flow [13] or functional-style applicative expressions [36]. Work on more general specification logics [1, 28, 16] 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 rather general rely/guarantee-style rule for linking arbitrary program fragments.

The other novelty 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 twist in the proofs is the use of step-indexed approximations to the semantics of assertions and their orthogonals. Fuller details, including proofs, may be found in the companion technical report [8].

2 The Machine

The metavariables n and b range over the integers, $\mathbb{Z}$, and booleans, $\mathbb{B}$, respectively. We assume a set $\mathbb{V}$ of names for global variables, ranged over by x. The metavariables bop and uop range over typed (binary and unary, respectively) arithmetic and logical operations such as addition and conjunction. Programs, p, are finite partial functions from labels, $l \in \mathbb{N}$, to instructions I:

$$\begin{aligned} I &:= \texttt{pushc}\ \underline{v} \mid \texttt{pushv x} \mid \texttt{pop x} \mid \texttt{dup} \mid \texttt{binop}_{bop} \mid \\ &\quad\ \texttt{unop}_{uop} \mid \texttt{brtrue}\ l \mid \texttt{call}\ l \mid \texttt{ret} \mid \texttt{halt} \\ \mathit{Programs} \ni p &:= [l_1 : I_1, \ldots, l_n : I_n] \end{aligned}$$

Stores are finite functions from $\mathbb{V}$ to values (i.e. to $\mathbb{B} \cup \mathbb{Z}$). Our machine has two stacks: the evaluation stack, σ, used for intermediate values, passing arguments and returning results, is a finite sequence of values, whilst the control stack, C, is a finite sequence of labels (return addresses):

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

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. 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 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

$$
\begin{array}{rl}
\langle p, l : \texttt{pushc}\ v|C|G|\sigma|l\rangle & \rightarrow \langle p, l : \texttt{pushc}\ v|C|G|\sigma, v|l+1\rangle \\
\langle p, l : \texttt{pushv}\ \texttt{x}|C|G, x = v|\sigma|l\rangle & \rightarrow \langle p, l : \texttt{pushv}\ \texttt{x}|C|G, x = v|\sigma, v|l+1\rangle \\
\langle p, l : \texttt{dup}|C|G|\sigma, v|l\rangle & \rightarrow \langle p, l : \texttt{dup}|C|G|\sigma, v, v|l+1\rangle \\
\langle p, l : \texttt{pop}\ \texttt{x}|C|G, x = v'|\sigma, v|l\rangle & \rightarrow \langle p, l : \texttt{pop}\ \texttt{x}|C|G, x = v|\sigma|l+1\rangle \\
\langle p, l : \texttt{binop}_{bop}|C|G|\sigma, v_1, v_2|l\rangle & \rightarrow \langle p, l : \texttt{binop}_{bop}|C|G|\sigma, v_3|l+1\rangle \\
\multicolumn{2}{r}{\text{if } v_3 = (v_1\ bop\ v_2).} \\
\langle p, l : \texttt{unop}_{uop}|C|G|\sigma, v|l\rangle & \rightarrow \langle p, l : \texttt{unop}_{uop}|C|G|\sigma, v'|l+1\rangle \\
\multicolumn{2}{r}{\text{if } v' = uop\ v.} \\
\langle p, l : \texttt{brtrue}\ l'|C|G|\sigma, true|l\rangle & \rightarrow \langle p, l : \texttt{brtrue}\ l'|C|G|\sigma|l'\rangle \\
\langle p, l : \texttt{brtrue}\ l'|C|G|\sigma, false|l\rangle & \rightarrow \langle p, l : \texttt{brtrue}\ l'|C|G|\sigma|l+1\rangle \\
\langle p, l : \texttt{call}\ l'|C|G|\sigma|l\rangle & \rightarrow \langle p, l : \texttt{call}\ l'|C, l+1|G|\sigma|l'\rangle \\
\langle p, l : \texttt{ret}|C, l'|G|\sigma|l\rangle & \rightarrow \langle p, l : \texttt{ret}|C|G|\sigma|l'\rangle
\end{array}
$$

Fig. 1. Operational Semantics of the Abstract Machine

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 $\texttt{brtrue}\ l$ 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 $\texttt{call}\ l$ 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 $\rightarrow^k$ and the infinite transition predicate $\rightarrow^\omega$ 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:

$$Safe_0\langle p|C|G|\sigma|l\rangle \qquad Safe_k\langle p, l : \texttt{halt}|C|G|\sigma|l\rangle$$

$$\frac{\langle p|C|G|\sigma|l\rangle \rightarrow \langle p|C'|G'|\sigma'|l'\rangle \quad Safe_k\langle p|C'|G'|\sigma'|l'\rangle}{Safe_{k+1}\langle p|C|G|\sigma|l\rangle}$$

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

Although this semantics is fairly standard, the choice to work with partial stores is significant: execution can get stuck accessing an undefined variable, so, for example, there are contexts which distinguish the sequence `pushv x; pop x` from a no-op.

3 Types and Assertions

As well as divergence and normal termination, programs can get stuck as a result of type errors applying basic operations, accessing undefined variables, underflowing either of the stacks, or jumping or calling outside the program. We rule out such behaviour using a type system, and define assertions relative to those types. This seems natural, but it is not the only reasonable way to proceed – although both the JVM and CLR have type-checkers ('verifiers'), the CLR does give a semantics to unverifiable code, which

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

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

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

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

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

Fig. 2. Expression Typing

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

A *base type*, τ, is either `int` or `bool`. A *stack type*, Σ, is a finite sequence $\tau_1, \ldots, \tau_n$ of base types. A *store type*, Δ, is a finite map $x_1 : \tau_1, \ldots, x_n : \tau_n$ from program variables to base types. We assume a set of *auxiliary variables*, ranged over by a. An *auxiliary variable context*, Θ, is a finite map from auxiliary variables to base 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 $\forall 1 \leq i \leq n.\ \rho(a_i) : \tau_i$.

Our low-level machine does not deal directly with complex expressions, but we will use them in forming assertions. Their grammar is as follows:

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

The expression form $s(i)$, for i a natural number, represents the ith element down the stack. Note that universal quantification over integers and booleans is an expression form. We assume that at least equality and a classical basis set of propositional connectives (e.g. negation and conjunction) are already 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, store typing and auxiliary variable context by the rules shown in Figure 2. 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 as one would expect. If $\Theta; \Delta; \Sigma \vdash E : \tau$, $\rho : \Theta$, $G : \Delta$ and $\sigma : \Sigma$ then we define $[\![E]\!]\,\rho\,G\,\sigma \in [\![\tau]\!]$ as in Figure 3. The semantics is well defined and commutes with each of our three forms of

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

Fig. 3. Expression Semantics

substitution. 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

$$\forall \rho : \Theta. \forall G : \Delta. \forall \sigma : \Sigma.\; [\![E_1]\!]\,\rho\,G\,\sigma \implies [\![E_2]\!]\,\rho\,G\,\sigma.$$

where $\implies$ is classical first-order implication. We define syntactic operations $shift(E)$ and $E \setminus\setminus E'$ for reindexing expressions when the stack is pushed or popped by

E	$shift(E)$	$E \setminus\setminus E'$
$s(0)$	$s(1)$	E'
$s(i+1)$	$s(i+2)$	$s(i)$
$E_1\ bop\ E_2$	$shift(E_1)\ bop\ shift(E_2)$	$(E_1 \setminus\setminus E')\ bop\ (E_2 \setminus\setminus E')$
$uop\ E_1$	$uop\ (shift(E_1))$	$uop\ (E_1 \setminus\setminus E')$
$\forall a \in \tau.E_1$	$\forall a \in \tau.shift(E_1)$	$\forall a \in \tau.(E_1 \setminus\setminus E')$ capture-avoiding
otherwise	E	E

where the $E \setminus\setminus E'$ operation, combining substitution for $s(0)$ with 'unshifting', is defined when $\Theta; \Delta; \Sigma, \tau' \vdash E : \tau$ and $\Theta; \Delta; \Sigma \vdash E' : \tau'$.

3.2 Types and Assertions for Programs

One could first present a type system and then a second inference system for assertion checking. Since the structure of the two inference systems would be similar, and we need types in defining assertions, we instead combine both into one system. The type part used here is monomorphic and somewhat restrictive, rather like that of the JVM. Over this we layer a richer assertion language, including explicit universal quantification. We define the structure of, and axiomatise entailment 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

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

Order:

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

Quantifier:

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

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

Arrow:

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

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

Frame:

$$\frac{\Theta \vdash \hat{E} : \mathtt{bool} \quad \Theta; \overline{\Delta}; \overline{\Sigma} \vdash I : \mathtt{bool} \quad \Theta \vdash \Delta; \Sigma; E \to \Delta'; \Sigma'; E' \ \mathtt{ok}}{\begin{array}{l}\Theta; \hat{E} \vdash \ (\Delta; \Sigma; E \to \Delta'; \Sigma'; E') \\ \qquad \leq (\overline{\Delta}, \Delta; \overline{\Sigma}, \Sigma; \mathit{shift}^{|\Sigma|}(I) \wedge E \to \overline{\Delta}, \Delta'; \overline{\Sigma}, \Sigma'; \mathit{shift}^{|\Sigma'|}(I) \wedge E')\end{array}}$$

Fig. 4. Subtyping/Entailment for Extended Label Types

These are subject to the following well-formedness conditions:

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

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

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 entailment on well-formed extended label types as shown in Figure 4. The basic entailment judgement has the form $\Theta; \hat{E} \vdash \chi \leq \chi'$ where $\Theta \vdash \hat{E} : \mathtt{bool}$ (we elide store and stack types here), $\Theta \vdash \chi \ \mathtt{ok}$ and $\Theta \vdash \chi' \ \mathtt{ok}$. 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 $[\rightarrow]$ rule is basically the usual one for subtyping function types, here playing the role of Hoare logic's rule of consequence. The $[\forall\exists \rightarrow]$ rule is a kind of internalization of the usual left rule for existential quantification. Note how in these two rules, 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].[1] 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 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 fragment.

The context Γ expresses assumptions about imported code that will be subsequently linked with p, whilst Γ' says what p will guarantee about exported labels, given 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 5 and 6.

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 prove later.

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 conclu-

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

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

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

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

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

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

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

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

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

Fig. 5. Program Logic: Structural and Logical Rules

sions. 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. [2]

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 either by adding special case rules, or[3] by a more general relaxation of our requirement for imports Γ and exports Γ' to be disjoint, but we omit the (uncomplicated) details 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

$$-; true; - \vdash [l : \mathtt{ret}] \rhd l : -; -; true \rightarrow -; -; true$$

[2] Equivalently, one could state these two rules with a single conclusion and add a right rule for conjunction. Our presentation threads the subject program p linearly through the derivation, making it clear that we only analyse its internal structure once.

[3] Thanks to one of the referees for this observation.

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

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

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

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

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

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

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

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

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

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

Fig. 6. Program Logic: Instruction-Specific Axioms

4 Semantics of Types and Assertions

One could certainly formulate and prove a correctness theorem for this logic syntactically, using a 'preservation and progress' argument. Technically, the syntactic approach is probably the simplest way of proving soundness, though it has the mild disadvantage of requiring a somewhat artifical extension of the typing and logical rules to whole configurations, rather than just the programs with which one starts. More fundamentally, the syntactic approach fails to capture the *meaning* of types and assertions, which, although this is partly a question of taste, I believe to be more than a philosophical objection.

In practice, we would like to be able safely to link low-level components that 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 formulated in terms of the observable behaviour of programs, independent of a particular

syntactic inference system. 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, however.

The way in which we choose to formulate an extensional semantics for types and assertions is via the notion of orthogonality with respect to contexts ('perping'). This is a general pattern, related to continuation-passing 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 [35], and proving strong normalization [19].

To establish the soundness of our link rule we also find it convenient to index our semantic definitions by step-counts, a technique that Appel and his collaborators have used extensively in defining semantic interpretations of types over low-level languages [3, 4, 2]. By contrast with our use of orthogonality, which is a deliberate choice of what we regard as the 'right' semantics, the use of indexing is essentially a technical device to make the operational proofs go through.

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

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

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 membership of S.

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

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

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

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

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.[4]

We then define the semantics of contextual judgements by

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

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

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. The following theorem establishes the semantic soundness of the entailment relation on extended label types, and is proved by induction on the rules in Figure 4:

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

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

We then use Theorem 1 and a further induction on the rules in Figures 5 and 6 to establish the semantic soundness of the program logic:

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

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. The technical report contains more details of these examples, as well a simple example of mutual recursion.

Example 1. It takes around twelve detailed steps to derive

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

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. 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 first 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 context $b : \texttt{int}$. It takes seven steps to show

$$\begin{array}{l} b : \texttt{int}; true; 0 : \chi'_0 \vdash \\ \ [10 : \texttt{call}\ 0, 11 : \texttt{br}\ 0] \rhd 10 : -; \texttt{int}; s(0) = b \rightarrow -; \texttt{int}; s(0) = b + 2 \end{array} \tag{2}$$

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. Another eight or so steps of reasoning with entailment and structural rules allows us to combine judgements (1) and (2) to obtain

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

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* re-analysing 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

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

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

$$-; \mathit{true}; - \vdash p \triangleright 0 : \forall a : \mathtt{int}.(-; \mathtt{int}; a < 7 \wedge s(0) = a \rightarrow -; \mathtt{int}; s(0) = a + 1) \tag{3}$$

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

We now consider [link]ing the judgement (3) with that we derived for the client program (2). With a few purely logical manipulations (using $\hat{E} = b < 6$) we can derive

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

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 something equivalent, seems necessary for this kind of reasoning: we need 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. Eight small steps of reasoning with the entailment rules allow one to deduce

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

So, 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. This is a common pattern: 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 p something like

$$[1 : \texttt{pushc}\ 0,\ 2 : \texttt{pop}\ \texttt{x},\ 3 : \texttt{pushc}\ \mathit{true},\ 4 : \texttt{brtrue}\ 9,\ 5 : \texttt{pushv}\ \texttt{x},\\ 6 : \texttt{pushc}\ 1,\ 7 : \texttt{binop}_+,\ 8 : \texttt{pop}\ \texttt{x},\ 9 : \texttt{pushv}\ \texttt{x},\ 10 : \texttt{pushc}\ 5,\ 11 : \texttt{binop}_<,\\ 12 : \texttt{brtrue}\ 5,\ 13 : \texttt{ret}]$$

Such unstructured control-flow makes no difference to reasoning in our low-level logic: as one would hope, we can easily derive

$$-; \mathit{true}; - \vdash p \rhd 0 : x : \texttt{int}; -; \mathit{true} \to x : \texttt{int}; -; x = 5.$$

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

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

we can derive

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

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

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, [11]); 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 [10] 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 [6] have combined 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 [18] and von Oheimb & Nipkow [34, 26]. Our fussiness about contexts and quantification, and use of substructural ideas, differs from most of this other work, leading to a rather elegant account of invariants of procedures and a complete absence of side-conditions. The use of auxiliary variables scoped across an entire judgement, and explicit universal quantification (rather than implicit, closing, quantification on each triple in the context) seems much the best way to reason compositionally, allowing one to relate assumptions on different labels, but has previously been shied away from. As von Oheimb [34] says

> A real solution would be explicit quantification like $\forall Z.\{P\,Z\}\,c\,\{Q\,Z\}$, but this changes the structure of Hoare triples and makes them more difficult to handle. Instead we prefer implicit quantification at the level of triple validity[...]

The other line of closely related research is on proof-carrying code [24, 23] and typed assembly languages [22], much of which has a similar 'logical' flavour to this, with substructural ideas having been applied to stacks [17], heaps [20] and aliasing [32]. Our RISC-like 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 the cited work, we have a much simpler machine (no pointer manipulation, dynamic allocation or code pointers), but go beyond simple syntactic type soundness to give a richer program logic with a semantic interpretation. Especially close is the work of Appel et al on semantic models of types in foundational proof-carrying code, from which we borrowed the step-indexed proof technique, and of Shao and Hamid [14] on interfacing Hoare logic with a syntactic type system for low-level code as a way of verifying linkage between typed assembly language modules verified using different systems.

One might (and all the referees did) reasonably ask why we have not followed most of the recent work in this area by, firstly, formalizing our logic in a theorem prover and, secondly, avoiding all the explicit treatment of quantification, auxiliary variables etc. in favour of inheriting them from a shallow embedding of the semantics in an ambient

higher-order logic. Machine checking is certainly helpful in avoiding unsoundness, is probably essential for managing all the details of logics for realistic-scale languages or machines, and is a necessary component in PCC-like deployment scenarios. We certainly plan to investigate mechanization in the near future, but believe a traditional pencil-and-paper appraoch is quite reasonable for preliminary investigations with toy calculi, despite the history of unsound Hoare logic rules in the literature. We certainly made errors in earlier versions of this paper but, given an independent semantics and formalization of correctness (rather than trying to work purely axiomatically), see no reason why a program logic is more likely to be unsound than any interesting type system or static analysis in the literature (though opinions differ on just how likely that is...). As regards the second point, shallow embeddings are very convenient, especially for mechanization, but they do have the potential to miss the semantic subtleties of non-trivial languages. Whether or not one recognizes it, the embedding constitutes a denotational semantics that, if one is not extremely careful and the language is much more complex than while-programs, will be far from fully-abstract. Delegating the entailment used in the rule of consequence to implication in the metalogic therefore runs the risk of being incomplete for reasoning about behavioural properties of programs in the original language. It is unclear (at least to me) what the semantic import of, say, a relative completeness result factored through such a non-fully-abstract semantics is supposed to be.

There are many variations and improvements one might make to the logic, such as adding subtyping and polymorphism at the type level and adding other connectives to the program logic level. But it must be admitted that 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. A generalisation to higher-order, with first-class code pointers, would bring some complexity, but also seems to offer some simplifications, such as the fact that one only needs preconditions as everything is in CPS. Another extension would be to more general dynamic allocation. Both first-class code pointers and heaps have been the objects of closely related work on semantics and types for both low-level and high-level languages (e.g. [9] and the references therein); transferring those ideas to general assertions on low-level code looks eminently doable. We would also like to generalize 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 ramifications of our semantic interpretation. One of its effects is to close the interpretation of extended label types with respect to an observational equivalence, which is a pleasant feature, but our inference system then seems unlikely to be complete. The links with work of Honda et al. [15] on observationally complete logics for state and higher-order functions deserve 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 con-

figurations to subsets of $Stores \times Stacks$ 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 *Proc. 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 *Proc. 27th ACM Symposium on Principles of Programming Languages (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), 2001.

[5] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Proc. 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3223 of *Lecture Notes in Computer Science*, 2004.

[6] F. Bannwart and P. Muller. A program logic for bytecode. In *Proc. 1st 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. A typed, compositional logic for a stack-based abstract machine. Technical Report MSR-TR-2005-84, Microsoft Research, June 2005.

[9] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, 2005.

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

[11] L. Cardelli. Program fragments, linking, and modularization. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[12] 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 (TOPLAS)*, 21(6), 1999.

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

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

[15] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. 20th IEEE Symposium on Logic in Computer Science (LICS)*, 2005.

[16] 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)*, 2000.

[17] L. Jia, F. Spalding, D. Walker, and N. Glew. Certifying compilation for a language with stack allocation. In *Proc. 20th IEEE Symposium on Logic in Computer Science (LICS)*, 2005.

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

[19] S. Lindley and I. Stark. Reducibility and $\top\top$ lifting for computation types. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, 2005.

[20] G. Morrisett, A. Amal, and M. Fluet. L3: A linear language with locations. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, 2005.

[21] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1), 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 (TOPLAS)*, 21(3), 1999.

[23] G. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (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 *Proc. 10th Annual Conference of the European Association for Computer Science Logic (CSL)*, volume 2142 of *Lecture Notes in Computer Science*, 2001.

[26] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In *Proc. Formal Methods Europe (FME)*, volume 2391 of *Lecture Notes in Computer Science*, 2002.

[27] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. CUP, 1998.

[28] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *Proc. 8th European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, 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 (TPHOLs)*, volume 2758 of *Lecture Notes in Computer Science*, 2003.

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

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

[32] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proc. 9th European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, 2000.

[33] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[34] D. von Oheimb. Hoare logic for mutual recursion and local variables. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, 1999.

[35] J. Vouillon and P.-A. Mellies. Semantic types: A fresh look at the ideal model for types. In *Proc. 31st ACM Symposium on Principles of Programming Languages (POPL)*, 2004.

[36] 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.