A Typed Logic for Stacks and Jumps
DRAFT

Nick Benton
Microsoft Research

March 2, 2004

Abstract

This note shows how one may define a program logic in the style of Floyd and Hoare for a simple, typed, stack-based, imperative language with unstructured control flow and local variables.

1 Introduction

Recent interest in security and certification has led to Floyd-Hoare and VDM-style programming logics becoming surprisingly fashionable, and to much work on type systems and specification 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[1] with control flow expressed using labelled jumps.

Most work on formalizing the type systems of these intermediate languages [10, 4] has treated the reality of stacks and jumps, though some authors have chosen to work with structured imperative control flow [5] or functional-style applicative expressions [11].

Even recent work on more general specification logics [1, 7, 6] has, however, mostly been done in the context of high-level languages. Borgström [2] 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 [8, 9] 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.

Here we present and prove the correctness of a simple logic for directly proving partial correctness assertions on a minimal stack-based language with

[1] Actually a rather poor design decision, but it's done now...

jumps. This is slightly more complex than traditional Hoare logic for while programs, as we have to deal with

- Unstructured control flow
- A stack which varies in size
- Globals and stack locations which vary in type
- Errors (underflow and type errors) arising from the previous factors

The first is not a significant problem (Floyd dealt with unstructured flowcharts in 1967 [3]). Here we address the other difficulties by imposing a simple type system to track the types of locations and the size of the stack and then defining assertions relative to those types.

2 The Language

2.1 Syntax

The syntax of basic values, instructions and programs is as follows. We assume the existence of a set $\mathbb{V}$ of variables (this is naturally taken to be finite if one thinks of them as local variables for a method or registers).

$$
\begin{array}{rcll}
n & \in & \mathbb{Z} & \\
b & \in & \mathbb{B} = \{true, false\} & \\
v & := & b \mid n & \\
l & \in & \mathbb{N} & \text{labels} \\
x & \in & \mathbb{V} & \text{variables} \\
op & \in & \{+, *, -, <, >, =, \ldots\} & \\
I & := & \texttt{pushc}\ \underline{v} \mid \texttt{pushvar}\ x \mid \texttt{pop}\ x \mid \texttt{binop}_{op} \mid \texttt{brtrue}\ l \mid \texttt{halt} & \\
p & := & [1 : I, 2 : I, \ldots, l : I] & \text{programs}
\end{array}
$$

The `pushc` $\underline{v}$ instruction pushes a constant boolean or integer value v onto the stack. The `pushvar` 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. 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.

Note that programs can be viewed as functions from an initial segment of the strictly positive naturals to instructions. Thus we may use either functional or unordered list notation for them.

2.2 Operational Semantics

States are functions from variables to values (i.e. to the union of the booleans and the integers, which in this presentation we assume to be disjoint). Stacks are finite sequences of values:

$$S \ni \mathit{States} = \mathbb{V} \to (\mathbb{Z} \cup \mathbb{B})$$

$$\sigma \ni \mathit{Stacks} = (\mathbb{Z} \cup \mathbb{B})^*$$

A configuration consists of a program p and triple of a state, a stack and program counter (the current label). We present the operational semantics using a small-step transition relation:

$$[p]; \langle S, \sigma, l\rangle \to \langle S', \sigma', l'\rangle$$

defined by

$$\begin{array}{l}
[p, l : \texttt{pushc } v]; \langle S, \sigma, l\rangle \to \langle S, (\sigma, v), l+1\rangle \\
[p, l : \texttt{pushvar } x]; \langle S, \sigma, l\rangle \to \langle S, (\sigma, S(x)), l+1\rangle \\
[p, l : \texttt{pop } x]; \langle S, (\sigma, v), l\rangle \to \langle S[v/x], \sigma, l+1\rangle \\
[p, l : \texttt{binop}_{op}]; \langle S, (\sigma, v_1, v_2), l\rangle \to \langle S, (\sigma, v_1 op v_2), l+1\rangle \quad \text{if } op \text{ defd} \\
[p, l : \texttt{brtrue } l']; \langle S, (\sigma, true), l\rangle \to \langle S, \sigma, l'\rangle \\
[p, l : \texttt{brtrue } l']; \langle S, (\sigma, false), l\rangle \to \langle S, \sigma, l+1\rangle
\end{array}$$

and we define the multistep transition relation $[p]; \langle S, \sigma, l\rangle \to^* \langle S', \sigma', l'\rangle$ in the obvious way:

$$[p]; \langle S, \sigma, l\rangle \to^* \langle S, \sigma, l\rangle$$

$$\frac{[p]; \langle S, \sigma, l\rangle \to \langle S', \sigma', l'\rangle \quad [p]; \langle S', \sigma', l'\rangle \to^* \langle S'', \sigma'', l''\rangle}{[p]; \langle S, \sigma, l\rangle \to^* \langle S'', \sigma'', l''\rangle}$$

We sometimes write $[p]; \langle S, \sigma, l\rangle \Downarrow \langle S', \sigma'\rangle$ to mean $[p]; \langle S, \sigma, l\rangle \to^* \langle S', \sigma', l'\rangle$ for some l' such that $p(l') = \texttt{halt}$.

The following is obvious, assuming each primitive operation is a function, since each configuration has at most one successor.

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

It is also worth noting that execution preserves, and is independent of, extensions to the stack at the bottom:

Lemma 2. *If* $[p]; \langle S, \sigma, l\rangle \to^* \langle S', \sigma', l'\rangle$ *then for any* σ'', $[p]; \langle S, (\sigma'', \sigma), l\rangle \to^* \langle S', (\sigma'', \sigma'), l'\rangle$. □

3 Types and Assertions

As well as terminating executions that hit a `halt` instruction, programs can exhibit four other kinds of behaviour:

1. Divergence
2. Type errors
3. Stack underflow
4. Wild jumps

In the small step semantics, divergence is corresponds to an infinite sequence of transitions, whilst stack underflow, type errors and wild jumps show up as configurations in which the current instruction is not `halt`, yet no transition is possible (stuck configurations).

Our goal is to define a Floyd-Hoare style logic. Clearly we will have predicates P of some sort on stacks and states, but there are various possible semantics for judgements[2] {P} p,l {Q}:

1. Total correctness. If $\langle S, \sigma\rangle \in [\![P]\!]$ then $[p]; \langle S, \sigma, l\rangle \Downarrow \langle S', \sigma'\rangle$ with $\langle S', \sigma'\rangle \in [\![Q]\!]$.
2. Very partial correctness. If $\langle S, \sigma\rangle \in [\![P]\!]$ and $[p]; \langle S, \sigma, l\rangle \Downarrow \langle S', \sigma'\rangle$, then $\langle S', \sigma'\rangle \in [\![Q]\!]$. Such specifications are satisfied by erroneous as well as diverging programs.
3. Error-free partial correctness. If $\langle S, \sigma\rangle \in [\![P]\!]$ then EITHER $[p]; \langle S, \sigma, l\rangle \Downarrow \langle S', \sigma'\rangle$ with $\langle S', \sigma'\rangle \in [\![Q]\!]$, OR $[p]; \langle S, \sigma, l\rangle$ diverges.
4. Error-tracking partial correctness. We enrich the language of predicates to include explicit treatment of erroneous states (as in the work of Jacobs and Huisman [6]).
5. Underspecified partial correctness. We modify the semantics to be defined but non-deterministic in some error cases (e.g. simple type errors like adding a boolean to an integer, maybe reading below the bottom of the stack). Then do normal partial correctness wrt the modified semantics.
6. Default values. Modify the semantics to return default values for accesses below the bottom of the stack.
7. Move to a relational interpretation of predicates.
8. ...

[2] Actually, our judgements are not of quite this form.

There also seems to be some scope for varying exactly how predicates on stacks themselves should be formulated, to deal with, or avoid, references to inaccessible stack locations.

Here we choose to rule out erroneous behaviour by means of a simple type system, and to define our predicates relative to those types. Thus we will only be able to prove properties of programs which are typeable. 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

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

$$\begin{array}{rcll} \tau & \in & \{\texttt{int}, \texttt{bool}\} & \text{base types} \\ \Sigma & := & - \mid \Sigma, \tau & \text{stack types} \\ \Delta & \in & \mathbb{V} \to \{\texttt{int}, \texttt{bool}\} & \text{state types} \end{array}$$

We define $[\![\texttt{int}]\!] = \mathbb{Z}$ and $[\![\texttt{bool}]\!] = \mathbb{B}$. Typing for states is straightforward:

$$\vdash S : \Delta \text{ iff } \forall x \in \mathbb{V}. S(x) \in [\![\Delta(x)]\!]$$

For stacks, we use the following[3] rules:

$$\vdash \sigma : - \quad \frac{\vdash \sigma : \Sigma \quad \vdash v : \tau}{\vdash \sigma, v : \Sigma, \tau}$$

We define substitution on states and state types like this:

$$\begin{array}{rcl} (\Delta[x \mapsto \tau])(y) & = & \begin{cases} \tau & \text{if } x = y \\ \Delta(y) & \text{otherwise} \end{cases} \\ (S[x \mapsto v])(y) & = & \begin{cases} v & \text{if } x = y \\ S(y) & \text{otherwise} \end{cases} \end{array}$$

Lemma 3. *If* $\vdash S : \Delta$ *and* $v : \tau$ *then* $\vdash S[x \mapsto v] : \Delta[x \mapsto \tau]$. □

Similarly for stacks, though here substitution is partial:

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

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

Lemma 4. *If* $\vdash \sigma : \Sigma$ *and* $v : \tau$ *then* $\vdash \sigma[i \mapsto v] : \Sigma[i \mapsto \tau]$. □

[3]Note that these rules interpret stack types as constraints on the top of the stack, not as a exact specification of the whole stack.

3.2 Assertions and Program Types

To write specifications, we will need a syntax for boolean- and integer-valued expressions. This is given by the following grammar:

$$E \;:=\; \underline{n} \mid \underline{b} \mid x \mid s(i) \mid E \mathit{\,op\,} E$$

As before, the metavariable *op* 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.)

Expressions can have type either `int` or `bool` in the context of a given stack and store typing:

$$\Delta, \Sigma \vdash n : \texttt{int} \qquad \Delta, \Sigma \vdash b : \texttt{bool}$$

$$(\Delta, x : \tau), \Sigma \vdash x : \tau \qquad \Delta, (\Sigma, \tau) \vdash s(0) : \tau$$

$$\frac{\Delta, \Sigma \vdash s(i) : \tau}{\Delta, (\Sigma, \tau') \vdash s(i+1) : \tau} \qquad \frac{\Delta, \Sigma \vdash E_1 : \tau_1 \quad \Delta, \Sigma \vdash E_2 : \tau_2 \quad \mathit{op} : \tau_1 \times \tau_2 \to \tau_3}{\Delta, \Sigma \vdash E_1 \mathit{\,op\,} E_2 : \tau_3}$$

We define substitution $E[E'/x]$ and $E[E'/s(i)]$ in the usual way.

Lemma 5.

1. *If* $S[x \mapsto \tau'], \sigma \vdash E : \tau$ *and* $S, \sigma \vdash E' : \tau'$ *then* $S, \sigma \vdash E[E'/x] : \tau$.
2. *If* $S, \sigma[i \mapsto \tau'] \vdash E : \tau$ *and* $S, \sigma \vdash E' : \tau'$ *then* $S, \sigma \vdash E[E'/s(i)] : \tau$.

□

If $\Delta, \Sigma \vdash E : \tau$, $S : \Delta$ and $\sigma : \Sigma$ then we define $[\![E]\!](S, \sigma) \in [\![\tau]\!]$ inductively as follows: (and this *is* well-defined):

$$\begin{aligned}
[\![\underline{n}]\!](S, \sigma) &= n \\
[\![\underline{b}]\!](S, \sigma) &= b \\
[\![\underline{x}]\!](S, \sigma) &= S(x) \\
[\![s(0)]\!](S, (\sigma, v)) &= v \\
[\![s(i+1)]\!](S, (\sigma, v)) &= [\![s(i)]\!](S, \sigma) \\
[\![E_1 \mathit{\,\underline{op}\,} E_2]\!](S, \sigma) &= [\![E_1]\!](S, \sigma) \mathit{\;op\;} [\![E_2]\!](S, \sigma)
\end{aligned}$$

Lemma 6.

1. *If* $S[x \mapsto \tau'], \sigma \vdash E : \tau$ *and* $S, \sigma \vdash E' : \tau'$ *then*

$$[\![E[E'/x]]\!](S, \sigma) = [\![E]\!](S[x \mapsto [\![E']\!](S, \sigma)], \sigma).$$

2. *If* $S, \sigma[i \mapsto \tau'] \vdash E : \tau$ *and* $S, \sigma \vdash E' : \tau'$ *then*

$$[\![E[E'/s(i)]]\!](S, \sigma) = [\![E]\!](S, \sigma[i \mapsto [\![E']\!](S, \sigma)]).$$

□

If $\Delta, \Sigma \vdash E_i : \texttt{bool}$ for $i \in \{1, 2\}$, we write $\Sigma, \Delta \models E_1 \implies E_2$ to mean

$$\forall S. \forall \sigma. \text{if } \vdash S : \Delta \text{ and } \vdash \sigma : \Sigma \text{ then } [\![E_1]\!](S, \sigma) \implies [\![E_2]\!](S, \sigma)$$

If E is an expression, then we define $\mathit{shift}(E)$ by

$$\begin{array}{rcl} \mathit{shift}(s(i)) & = & s(i+1) \\ \mathit{shift}(E_1 \ \mathit{op}\ E_2) & = & \mathit{shift}(E_1) \ \mathit{op}\ \mathit{shift}(E_2) \\ \mathit{shift}(E) & = & E \text{ otherwise} \end{array}$$

Lemma 7.

1. *If* $\Delta, \Sigma \vdash E : \tau$ *then for any* τ', $\Delta, (\Sigma, \tau') \vdash \mathit{shift}(E) : \tau$.
2. *If* $\Delta, \Sigma \vdash E : \tau$, $\vdash S : \Delta$ *and* $\sigma : \Sigma$ *then for any* v,

$$[\![E]\!](S, \sigma) \ = \ [\![\mathit{shift}(E)]\!](S, (\sigma, v)).$$

□

One can present the type system for programs and then the assertion checking rules on typed programs, but here we choose to combine them into one system. A program type Γ thus maps labels to triples of state type, stack type and assertion:

$$\Gamma \ := \ [1 : \Delta_1, \Sigma_1, E_1; \ldots; l : \Delta_l, \Sigma_l, E_l]$$

with the well-typedness condition

$$\frac{\forall 1 \leq l' \leq l.\ \Delta_{l'}, \Sigma_{l'} \vdash E_{l'} : \texttt{bool}}{\vdash [1 : \Delta_1, \Sigma_1, E_1; \ldots; l : \Delta_l, \Sigma_l, E_l]}$$

The rules for checking a particular instruction in a program in the context of a program typing Γ, assuming that $\vdash \Gamma$ holds, are shown in Figure 1. We then say a program is well-typed and specified, and write $\Gamma \vdash p$, if all its instructions are:

$$\frac{\vdash \Gamma \qquad \forall 1 \leq l' \leq l.\ \Gamma \vdash I_{l'}}{\Gamma \vdash [1 : I_1, \ldots, l : I_l]}$$

$$\Gamma \vdash l : \texttt{halt}$$

$$\frac{\Gamma(l) = \Delta_l, \Sigma_l, E_l \qquad \vdash v : \tau \qquad \Gamma(l+1) = \Delta_l, (\Sigma_l, \tau), E_{l+1} \\ \Delta_l, (\Sigma_l, \tau) \models \mathit{shift}(E_l) \implies E_{l+1}[v/s(0)]}{\Gamma \vdash l : \texttt{pushc}\ v}$$

$$\frac{\Gamma(l) = \Delta_l, \Sigma_l, E_l \qquad \Delta_l(x) = \tau \qquad \Gamma(l+1) = \Delta_l, (\Sigma_l, \tau), E_{l+1} \\ \Delta_l, (\Sigma_l, \tau) \models \mathit{shift}(E_l) \implies E_{l+1}[x/s(0)]}{\Gamma \vdash l : \texttt{pushvar}\ x}$$

$$\frac{\Gamma(l) = \Delta_l, (\Sigma_{l+1}, \tau), E_l \qquad \Gamma(l+1) = \Delta_l[x \mapsto \tau], \Sigma_{l+1}, E_{l+1} \\ \Delta_l, (\Sigma_{l+1}, \tau) \vdash E_l \implies (\mathit{shift}(E_{l+1}))[s(0)/x]}{\Gamma \vdash l : \texttt{pop}\ x}$$

$$\frac{\Gamma(l) = \Delta_l, (\Sigma, \tau_1, \tau_2), E_l \qquad \Gamma(l+1) = \Delta_l, (\Sigma, \tau_3), E_{l+1} \qquad \mathit{op} : \tau_1 \times \tau_2 \to \tau_3 \\ \Delta_l, (\Sigma, \tau_2, \tau_1) \models E_l \implies (\mathit{shift}(E_{l+1}))[(s(0)\ \texttt{op}\ s(1))/s(1)]}{\Gamma \vdash l : \texttt{binop}_{\mathit{op}}}$$

$$\frac{\Gamma(l) = \Delta_l, (\Sigma_{l+1}, \texttt{bool}), E_l \qquad \Gamma(l+1) = \Delta_l, \Sigma_{l+1}, E_{l+1} \qquad \Gamma(l') = \Delta_l, \Sigma_{l+1}, E_{l'} \\ \Delta_l, (\Sigma_{l+1}, \texttt{bool}) \models E_l \wedge \neg s(0) \implies \mathit{shift}(E_{l+1}) \\ \Delta_l, (\Sigma_{l+1}, \texttt{bool}) \models E_l \wedge s(0) \implies \mathit{shift}(E_{l'})}{\Gamma \vdash l : \texttt{brtrue}\ l'}$$

Figure 1: Rules for checking types and specifications

3.3 Soundness

We say what it means for a configuration to be well typed and specified with respect to an extended program type:

$$\frac{\Gamma \vdash p \quad \Gamma(l) = \Delta_l, \Sigma_l, E_l \quad \vdash \sigma : \Sigma_l \quad \vdash S : \Delta_l \quad [\![E_l]\!](S, \sigma) = \mathit{true}}{\Gamma \vdash [p]; \langle S, \sigma, l \rangle}$$

Taking the requirements in order, we need that the whole program is well typed and specified (which includes the requirement that the extended program type is well-formed), the state and stack have the type assigned to them at the current label and meet the specification associated with this label.

Soundness of the logic with respect to the operational semantics is then formulated in the usual preservation and progress style:

Theorem 1. *If $\Gamma \vdash [p]; \langle S, \sigma, l \rangle$ then either $p(l) = \texttt{halt}$ or there exist S', σ', l' such that $[p]; \langle S, \sigma, l \rangle \to \langle S', \sigma', l' \rangle$ and $\Gamma \vdash [p]; \langle S', \sigma', l' \rangle$.*

Proof. We consider each case for the instruction I_l in turn.

`halt` Nothing to prove.

`pushc` v If we take $S' = S$, $\sigma' = (\sigma, v)$, $l' = l + 1$ then we have a transition. We know $\vdash S : \Delta_l$, $\vdash \sigma : \Sigma_l$ and $[\![E_l]\!](S, \sigma) = true$ by assumption. We know $\Delta_{l'} = \Delta_l$, $\Sigma_{l'} = (\Sigma_l, \tau)$ and $v : \tau$ from the fact that I_l is well typed and specified. Hence we can deduce $\vdash \sigma' : \Sigma_{l'}$ and $\vdash S' : \Delta_{l'}$. Now

$$\begin{array}{ll} & [\![E_l]\!](S, \sigma) = true \\ \Longrightarrow & [\![shift(E_l)]\!](S, (\sigma, v)) = true \quad \text{Lemma 7} \\ \Longrightarrow & [\![E_{l'}[v/s(0)]]\!](S, (\sigma, v)) = true \quad \text{assmp} \\ \Longrightarrow & [\![E_{l'}]\!](S', \sigma') = true \quad \text{Lemma 6} \end{array}$$

`pushvar` x Let $v = S(x)$ and take $S' = S$, $\sigma' = (\sigma, v)$, $l' = l + 1$. Then we have a transition. We know $\Delta_{l'} = \Delta_l$, $\Sigma_{l'} = \Sigma_l, \tau$, where $\tau = \Delta_l(x)$ by the type rules, so $\vdash S' : \Delta_{l'}$ and $\vdash \sigma' : \Sigma_{l'}$. For the specification, we then reason essentially as in the case above.

`pop` x Take $l' = l + 1$ then we know there's a τ such that $\Sigma_l = (\Sigma_{l'}, \tau)$, $\Delta_{l'} = \Delta_l[x \mapsto \tau]$ from typing, and $\vdash S : \Delta_l$ and $\vdash \sigma : \Sigma_l$ by assumption. So $\sigma = (\sigma', v)$ for some $\sigma' : \Sigma_{l'}$ and $v : \tau$. Let $S' = S[x \mapsto v]$ so we have a transition, and $S' : \Delta_{l'}$ by Lemma *.

$$\begin{array}{ll} & [\![E_l]\!](S, \sigma) = true \\ \Longrightarrow & [\![shift(E_{l'})[s(0)/x]]\!](S, \sigma) = true \quad \text{assmp} \\ \Longrightarrow & [\![shift(E_{l'})[s(0)/x]]\!](S, (\sigma', v)) = true \\ \Longrightarrow & [\![shift(E_{l'})]\!](S[x \mapsto [\![s(0)]\!](S, (\sigma', v))], (\sigma', v)) = true \quad \text{Lemma 6} \\ \Longrightarrow & [\![shift(E_{l'})]\!](S[x \mapsto v], (\sigma', v)) = true \\ \Longrightarrow & [\![E_{l'}]\!](S', \sigma') = true \end{array}$$

as required.

`binop`$_{op}$ Take $l' = l+1$ then we know that $\Sigma_l = (\Sigma, \tau_2, \tau_1)$ and $\Sigma_{l'} = (\Sigma, \tau_3)$ for some Σ, where $op : \tau_1 \times \tau_2 \to \tau_3$. Then as $\vdash \sigma : \Sigma_l$ we must have $\sigma = (\sigma'', v_2, v_1)$ for some $\vdash \sigma'' : \Sigma$, $v_2 : \tau_2$ and $v_1 : \tau_1$. Let $\sigma' = (\sigma'', v_1\, op\, v_2)$ and then $\vdash \sigma' : \Sigma_{l'}$. For the state, we know $\vdash S : \Delta_l$ and $\Delta_{l'} = \Delta_l$ so we take $S' = S$ and we have typing and a transition. Now

$$\begin{array}{ll} & [\![E_l]\!](S, \sigma) = true \\ \Longrightarrow & [\![E_l]\!](S, (\sigma'', v_2, v_1)) = true \\ \Longrightarrow & [\![shift(E_{l'})[(s(0)\, op\, s(1))/s(1)]]\!](S, (\sigma'', v_2, v_1)) = true \\ \Longrightarrow & [\![shift(E_{l'})]\!](S, (\sigma'', [\![s(0)\, op\, s(1)]\!](S, (\sigma'', v_2, v_1)), v_1)) = true \\ \Longrightarrow & [\![shift(E_{l'})]\!](S, (\sigma'', v_1\, op\, v_2, v_1)) = true \\ \Longrightarrow & [\![E_{l'}]\!](S, (\sigma'', v_1\, op\, v_2)) = true \\ \Longrightarrow & [\![E_{l'}]\!](S, \sigma') = true \end{array}$$

as required.

`brtrue` k We know $\Delta_{l+1} = \Delta k = \Delta_l$, and $\Sigma_{l+1} = \Sigma k$ and $\Sigma_l = (\Sigma_k, \texttt{bool})$. So $\sigma = (\sigma', b)$ for some boolean b, and $\vdash \sigma' : \Sigma_k$ by typing. If $b = true$ then

we have a transition with $l' = k$ and $S' = S$, so $\vdash S' : \Delta_{l'}$ is immediate. Then for the spec

$$\begin{array}{ll} & [\![E_l]\!](S, \sigma) = true \\ \Longrightarrow & [\![E_l]\!](S, (\sigma', true)) = true \\ \Longrightarrow & [\![E_l \wedge s(0)]\!](S, (\sigma', true)) = true \\ \Longrightarrow & [\![shift(E_k)]\!](S, (\sigma', true)) = true \\ \Longrightarrow & [\![E_k]\!](S, \sigma') = true \\ \Longrightarrow & [\![E_{l'}]\!](S', \sigma') = true \end{array}$$

as required. The case for $b = false$ and $l' = l + 1$ is similar.

□

3.4 Examples

Consider the source program

```
x := 0;
while(5>x) {
  x := x+1;
}
```

A typical Java or $C^\sharp$ compiler would produce code roughly like the following, where we have annotated each instruction with a type and specification. Note that the while loop is compiled with the test and conditional backwards branch at the end, and is preceded by a header which branches unconditionally into the loop to execute the test the first time.

l	I_l	$\Delta_l; \Sigma_l$	E_l
1	`pushc` 0	x : `int`; $-$	$true$
2	`pop` x	x : `int`; `int`	$s(0) = 0$
3	`pushc` $true$	x : `int`; $-$	$x = 0$
4	`brtrue` 9	x : `int`; `bool`	$x = 0 \wedge s(0) = true$
5	`pushvar` x	x : `int`; $-$	$x < 5$
6	`pushc` 1	x : `int`; `int`	$s(0) < 5$
7	$\texttt{binop}_{add}$	x : `int`; `int`, `int`	$s(1) < 5 \wedge s(0) = 1$
8	`pop` x	x : `int`; `int`	$s(0) \leq 5$
9	`pushvar` x	x : `int`; $-$	$x \leq 5$
10	`pushc` 5	x : `int`; `int`	$x \leq 5 \wedge s(0) = x$
11	$\texttt{binop}_{gt}$	x : `int`; `int`, `int`	$x \leq 5 \wedge s(1) = x \wedge s(0) = 5$
12	`brtrue` 5	x : `int`; `bool`	$x \leq 5 \wedge s(0) = (x < 5)$
13	`halt`	x : `int`; $-$	$x = 5$

Here are the justifications that each instruction is well-typed and specified according to the rules:

1. x : `int`, (`int`) $\models$ $shift(true) \implies (s(0) = 0)[0/s(0)]$ which is valid.

2. $x : \texttt{int}, (\texttt{int}) \models s(0) = 0 \implies \mathit{shift}(x = 0)[s(0)/x]$ which is valid.

3. $x : \texttt{int}, (\texttt{bool}) \models \mathit{shift}(x = 0) \implies (x = 0 \wedge s(0) = \mathit{true})[\mathit{true}/s(0)]$ which is valid.

4. Here there are two things to check. $x : \texttt{int}, (\texttt{bool}) \models x = 0 \wedge s(0) = \mathit{true} \wedge \neg s(0) \implies \mathit{shift}(x < 5)$ which holds because we have false on the left, and $x : \texttt{int}, (\texttt{bool}) \models x = 0 \wedge s(0) = \mathit{true} \wedge s(0) \implies \mathit{shift}(x \leq 5)$ which is valid.

5. $x : \texttt{int}, (\texttt{int}) \models \mathit{shift}(x < 5) \implies (s(0) < 5)[x/s(0)]$ which is valid.

6. $x : \texttt{int}, (\texttt{int}, \texttt{int}) \models \mathit{shift}(s(0) < 5) \implies (s(1) < 5 \wedge s(0) = 1)[1/s(0)]$ which is $x : \texttt{int}, (\texttt{int}, \texttt{int}) \models s(1) < 5 \implies (s(1) < 5 \wedge 1 = 1)$ which is valid.

7. $x : \texttt{int}, (\texttt{int}, \texttt{int}) \models s(1) < 5 \wedge s(0) = 1 \implies \mathit{shift}(s(0) \leq 5)[(s(0) + s(1))/s(1)]$ which is $x : \texttt{int}, (\texttt{int}, \texttt{int}) \models s(1) < 5 \wedge s(0) = 1 \implies s(0) + s(1) \leq 5$ which is valid.

8. $x : \texttt{int}, (\texttt{int}) \models s(0) \leq 5 \implies \mathit{shift}(x \leq 5)[s(0)/x]$ which is valid.

9. $x : \texttt{int}, (\texttt{int}) \models \mathit{shift}(x \leq 5) \implies (x \leq 5 \wedge s(0) = x)[x/s(0)]$ which is valid.

10. $x : \texttt{int}, (\texttt{int}, \texttt{int}) \models \mathit{shift}(x \leq 5 \wedge s(0) = x) \implies (x \leq 5 \wedge s(1) = x \wedge s(0) = 5)[5/s(0)]$ which is $x : \texttt{int}, (\texttt{int}, \texttt{int}) \models x \leq 5 \wedge s(1) = x \implies (x \leq 5 \wedge s(1) = x \wedge 5 = 5)$ which is valid.

11. $$x : \texttt{int}, (\texttt{int}, \texttt{int}) \models (x \leq 5 \wedge s(1) = x \wedge s(0) = 5) \implies \mathit{shift}(x \leq 5 \wedge s(0) = (x < 5))[(s(0) > s(1))/s(1)]$$ which is $x : \texttt{int}, (\texttt{int}, \texttt{int}) \models (x \leq 5 \wedge s(1) = x \wedge s(0) = 5) \implies \mathit{shift}(x \leq 5 \wedge (s(0) > s(1)) = (x < 5))$ which is valid.

12. Two things to prove: $x : \texttt{int}, (\texttt{bool}) \models x \leq 5 \wedge s(0) = (x < 5) \wedge \neg s(0) \implies \mathit{shift}(x = 5)$ which is valid, and $x : \texttt{int}, (\texttt{bool}) \models x \leq 5 \wedge s(0) = (x < 5) \wedge s(0) \implies \mathit{shift}(x < 5)$ which is valid.

13. Nothing to prove.

Thus we have established that the program is well-typed and that the value of x will be 5 on termination.

4 Conclusions

We have presented a typed program logic for a tiny stack-based intermediate language (bearing roughly the same relation to Java bytecode or CIL that the

language of while programs does to Java or $C^\sharp$). The logic is proved to be sound with respect to an operational semantics.

Nothing about the logic is amazingly novel, though the interplay between the type system and assertion language (the size of the stack and the types of stack locations and locals can vary) and the use of the $shift(\cdot)$ operation are interesting. As far as I can ascertain, this simple and natural system has not been presented before, though it certainly seems worth writing down.

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

[3] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of the AMS Symposium in Applied Mathematics*, volume 19, pages 19–32, 1967.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.