Compositional Type Systems for Stack-Based Low-Level Languages

Ando Saabas **Tarmo Uustalu**

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
Email: {ando|tarmo}@cs.ioc.ee

Abstract

It is widely believed that low-level languages with jumps must be difficult to reason about by being inherently non-modular. We have recently argued that this in untrue and proposed a novel method for developing compositional natural semantics and Hoare logics for low-level languages and demonstrated its viability on the example of a simple low-level language with expressions (Saabas & Uustalu 2005). The central idea is to use the implicit structure of finite disjoint unions present in low-level code as an (ambiguous) phrase structure.

Here we apply our method to a stack-based language and develop it further. We define a compositional natural semantics and Hoare logic for this language and go then on to show that, in addition to Hoare logics, one can also derive compositional type systems as weaker specification languages with the same method. We describe type systems for stack-error freedom and secure information flow.

Keywords: low-level languages, compositionality, Hoare logics, type systems, dataflow analyses, certified code, compilation of proofs, typings from compilation

1 Introduction

The advent of the paradigm of proof-carrying (or, more generally, certified) code has generated significant interest in reasoning about low-level code. This is because software is usually distributed in compiled form for the sake of self-containedness, but also because certification of compiled code instead of source programs eliminates the need for the software consumer to trust a compiler. Low-level languages are widely believed to be difficult to reason about as inherently non-modular. The lack of modularity is attributed to low-level code being flat (a set of labelled instructions with no explicit structure) and to the presence of general jumps. If a language is non-modular, it cannot have a compositional semantics or logic or type system.

We have recently argued that the non-modularity premise in untrue and proposed to exploit a very basic implicit structure present in low-level code as the "phrase structure" for semantic descriptions and logics of low-level languages (Saabas & Uustalu 2005). The structure in question is given by finite unions of pieces of code with non-overlapping support: a piece of code is either a single instruction or a finite union of non-overlapping pieces of code. Despite its banality and ambiguity (any piece of code can be parsed in many ways), this structure is perfectly viable from the point of metatheory and attractive from the point-of-view of practical reasoning about programs: it supports the idea that properties of a large piece of code should be provable from properties of its constituent small pieces (which can be established by different parties). An additional bonus of the method is that it supports compiling high-level programs together with proofs; in the compilation, the structure of a high-level source program hints the optimal way to structure its low-level equivalent.

In (Saabas & Uustalu 2005), we demonstrated this method on the example of a simple low-level language Goto with expressions. In this paper, we develop it further and consider an operand-stack based language Push. This language, although fairly similar on the surface, is more demanding because of the possibility of abnormal terminations due to stack errors (wrong operand types, stack underflow), but it is also richer in that, for Push, it makes sense to study not only logics as calculi for correctness, but also type systems as calculi for attesting weaker properties such as basic safety (stack-error freedom) or properties usually established by dataflow analyses.

The technical contribution of the paper is as follows. We define a structured version SPush of Push and equip it with a compositional natural semantics discriminating between normal and abnormal terminations and agreeing with the non-compositional small-step semantics of Push. We also define an error-free partial-correctness Hoare logic for SPush and prove it to be sound and (relatively) complete wrt. the natural semantics. For a compilation from While to SPush, we show that it preserves While proofs in a constructive sense (so that proof compilation is possible) and reflects SPush proofs. Beyond the logic, we also describe two type systems for SPush. The first system is a weakening of the Hoare logic and attests stack-error freedom, which we show sound and also complete wrt. an appropriate abstracted natural semantics. We also show that our compilation from While can be augmented to accompany the SPush code delivered with a typing derivation attesting that it is stack-error free. The second type system is equivalent to a secure information flow analysis.

The cornerstone technical ideas of the paper are: (i) low-level languages can be handled in a compositional way by exploiting an implicit phrase structure that they do have anyway, (ii) natural semantics can be made sensitive to abnormal terminations by introducing a special abnormal evaluation relation, (iii) Hoare logics and type systems should be derived systematically from natural semantics descriptions, (iv) the abstract interpretations that underlie dataflow

analyses can be described as abstract natural semantics and the analyses themselves as type systems. Not all of these ideas are new, but we believe that the paper combines them in a useful fashion.

The organization of the papers is the following. In Section 2, we introduce the syntax of the language PUSH and its non-compositional small-step semantics. In Section 3, we describe the syntax and the compositional natural (big-step) semantics of the structured version SPUSH. In Section 4, we describe the corresponding Hoare logic. In Sections 5 and 6, we discuss the abstract natural semantics and the type system for safe stack usage. In Section 7, we discuss a compilation of WHILE programs to SPUSH pieces of code and the corresponding compilation of proofs and type derivation generation. Section 8 discusses the abstract natural semantics and type system for secure information flow. Section 9 is a brief overview of the related work while 10 concludes.

2 The language and its small-step semantics

As advertised, our object of study is a simple operand-stack based low-level language, which we call PUSH.

The building blocks of the syntax of PUSH are labels $\ell \in \mathbf{Label}$, which are natural numbers, and instructions $instr \in \mathbf{Instr}$. We also assume having a countable set of program variables (registers) $x \in \mathbf{Var}$. The instructions of the language are defined by the grammar

$$\begin{array}{rcl} instr & ::= & \mathsf{load}\ x \mid \mathsf{store}\ x \mid \mathsf{push}\ n \\ & & \mid \mathsf{add} \mid \mathsf{eq} \mid \ldots \mid \mathsf{goto}\ \ell \mid \mathsf{gotoF}\ \ell \end{array}$$

A piece of code $c \in \mathbf{Code}$ is a finite set of labelled instructions, i.e., a set of pairs of a label and an instruction: $\mathbf{Code} =_{\mathrm{df}} \mathcal{P}_{\mathrm{fin}}(\mathbf{Label} \times \mathbf{Instr})$. A piece of code c is wellformed, if no label in it labels two different instructions, i.e., if $(\ell, instr), (\ell, instr') \in c$ imply $instr = instr'$. The domain of a piece of code is the set of labels in it: $\mathrm{dom}(c) =_{\mathrm{df}} \{\ell \mid (\ell, instr) \in c\}$.

Semantic descriptions of imperative languages are defined in terms of states. A state for PUSH consists of a label ℓ, stack ζ and store σ, which record the pc value and the content of the operand stack and the store at a moment: $\mathbf{State} =_{\mathrm{df}} \mathbf{Label} \times \mathbf{Stack} \times \mathbf{Store}$. A stack is a list whose elements can be both boolean or integer values: $\mathbf{Stack} =_{\mathrm{df}} (\mathbb{Z} \cup \mathbb{B})^*$. (We use the notation X^* for lists over X, $[]$ for the empty list, $x :: xs$ for the list with head x and tail xs and $xs +\!\!+ ys$ for the concatenation of xs and ys.) Variables can only be of integer type and must always be defined: $\mathbf{Store} =_{\mathrm{df}} \mathbf{Var} \to \mathbb{Z}$.

If a language is low-level, its semantics is usually described in an operational form that is small-step (there is no non-trivial notion of big steps one could talk of). The small-step semantics of PUSH is formulated via a single-step reduction relation $- \vdash - \twoheadrightarrow \subseteq \mathbf{State} \times \mathbf{Code} \times \mathbf{State}$ defined in Figure 1. The associated multi-step reduction relation $\twoheadrightarrow^*$ is its reflexive-transitive closure. It is immediate that $\twoheadrightarrow$ is deterministic, there is always at most one step possible. A state can be terminal ($c \vdash (\ell, \sigma) \not\twoheadrightarrow$) for two reasons: (i) we have $\ell \notin \mathrm{dom}(c)$, which signifies normal termination, or (ii) we have $\ell \in \mathrm{dom}(c)$ but the rule for the instruction at ℓ does not apply because of wrong types or shortage of potential operands on the stack, which signifies abnormal termination. (The possibility of abnormal terminations was not present in the language GOTO of (Saabas & Uustalu 2005).) The obvious shortcoming of this semantics is that it is entirely non-compositional (there is no phrase structure to follow) and that all of the code must be known at all times because of the jump instructions.

$$\frac{(\ell, \mathsf{store}\ x) \in c \quad n \in \mathbb{Z}}{c \vdash (\ell, n :: \zeta, \sigma) \twoheadrightarrow (\ell + 1, \zeta, \sigma[x \mapsto n])}\ \text{store}$$

$$\frac{(\ell, \mathsf{load}\ x) \in c}{c \vdash (\ell, \zeta, \sigma) \twoheadrightarrow (\ell + 1, \sigma(x) :: \zeta, \sigma)}\ \text{load}$$

$$\frac{(\ell, \mathsf{push}\ n) \in c}{c \vdash (\ell, \zeta, \sigma) \twoheadrightarrow (\ell + 1, n :: \zeta, \sigma)}\ \text{push}$$

$$\frac{(\ell, \mathsf{add}) \in c \quad n_0, n_1 \in \mathbb{Z}}{c \vdash (\ell, n_0 :: n_1 :: \zeta, \sigma) \twoheadrightarrow (\ell + 1, n_0 + n_1 :: \zeta, \sigma)}\ \text{add}$$

$$\frac{(\ell, \mathsf{eq}) \in c \quad n_0, n_1 \in \mathbb{Z}}{c \vdash (\ell, n_0 :: n_1 :: \zeta, \sigma) \twoheadrightarrow (\ell + 1, n_0 = n_1 :: \zeta, \sigma)}\ \text{eq}$$

$$\ldots$$

$$\frac{(\ell, \mathsf{goto}\ m) \in c}{c \vdash (\ell, \zeta, \sigma) \twoheadrightarrow (m, \zeta, \sigma)}\ \text{goto}$$

$$\frac{(\ell, \mathsf{gotoF}\ m) \in c}{c \vdash (\ell, \mathrm{tt} :: \zeta, \sigma) \twoheadrightarrow (\ell + 1, \zeta, \sigma)}\ \text{gotoF}^{\mathrm{tt}}$$

$$\frac{(\ell, \mathsf{gotoF}\ m) \in c}{c \vdash (\ell, \mathrm{ff} :: \zeta, \sigma) \twoheadrightarrow (m, \zeta, \sigma)}\ \text{gotoF}^{\mathrm{ff}}$$

Figure 1: Single-step reduction rules of PUSH

3 Structured version and natural semantics

To overcome the non-compositionality problem of the semantics described above, some structure needs to be introduced into PUSH code. As was shown in (Saabas & Uustalu 2005), a useful structure to use for defining the semantics of a low-level language compositionally is that of finite unions of non-overlapping pieces of code. This is present in the code anyway, but it is ambiguous (any set is a finite union of sets in many ways) and implicit, so one has to choose and make the choices explicit. Hence we define a corresponding structured version of PUSH, which we call SPUSH. Structured pieces of code $sc \in \mathbf{SCode}$ are defined by the following grammar

$$sc \quad ::= \quad (\ell, instr) \mid \mathbf{0} \mid sc_0 \oplus sc_1$$

which stipulates that a piece of code is either a single labelled instruction or a finite union of pieces of code. We define the domain $\mathrm{dom}(sc)$ of a piece of code sc to be the set of all labels in the code: $\mathrm{dom}(\mathbf{0}) = \emptyset$, $\mathrm{dom}((\ell, instr)) = \{l\}$, $\mathrm{dom}(sc_0 \oplus sc_1) = \mathrm{dom}(sc_0) \cup \mathrm{dom}(sc_1)$.

A piece of code is wellformed iff the labels of all of its instructions are different: a single instruction is always wellformed, $\mathbf{0}$ is wellformed and $sc_0 \oplus sc_1$ is wellformed iff both sc_0 and sc_1 are wellformed and $\mathrm{dom}(sc_0) \cap \mathrm{dom}(sc_1) = \emptyset$. Note that contiguity is not required for wellformedness, the domain of a piece of code does not have to be an interval.

The compositional semantic description we give for SPUSH is a (big-step) natural semantics. Since there is the possibility of abnormal terminations and we want to distinguish between non-terminations and abnormal terminations, we define two evaluation relations, $\succ\!\!\rightarrow, \succ\!\!\rightarrow\!\!\!\!\mid \subseteq \mathbf{State} \times \mathbf{SCode} \times \mathbf{State}$, one for normal, the other for abnormal terminating evaluations. Both relate possible initial states for evaluating a piece of code to the corresponding terminal states. The two relations are defined (mutually inductively) by the rules in Figure 2. Of course, alternatively one could say that we have just one evaluation relation but indexed by a doubleton for distinguishing between the two flavors of termination.

The $\mathrm{load}_{\mathrm{ns}}$ and $\mathrm{push}_{\mathrm{ns}}$ rules should be self-explanatory. Both $\mathsf{store}\ x$ and add can potentially cause an error, therefore there are two rules for them, for normal and abnormal evaluation.

$$\frac{}{(\ell,\zeta,\sigma)\succ(\ell,\mathsf{load}\ x)\rightarrow(\ell+1,\sigma(x)::\zeta,\sigma)}\ \mathrm{load}_{\mathrm{ns}}$$

$$\frac{n\in\mathbb{Z}}{(\ell,n::\zeta,\sigma)\succ(\ell,\mathsf{store}\ x)\rightarrow(\ell+1,\zeta,\sigma[x\mapsto n])}\ \mathrm{store}_{\mathrm{ns}}\quad\frac{\forall n\in\mathbb{Z},\zeta'\in(\mathbb{Z}\cup\mathbb{B})^*.\,\zeta\neq n::\zeta'}{(\ell,\zeta,\sigma)\succ(\ell,\mathsf{store}\ x)\twoheadrightarrow(\ell,\zeta,\sigma)}\ \mathrm{store}^{ab}_{\mathrm{ns}}$$

$$\frac{}{(\ell,\zeta,\sigma)\succ(\ell,\mathsf{push}\ n)\rightarrow(\ell+1,n::\zeta,\sigma)}\ \mathrm{push}_{\mathrm{ns}}$$

$$\frac{n_0,n_1\in\mathbb{Z}}{(\ell,n_0::n_1::\zeta,\sigma)\succ(\ell,\mathsf{add})\rightarrow(\ell+1,n_0+n_1::\zeta,\sigma)}\ \mathrm{add}_{\mathrm{ns}}\quad\frac{\forall n_0,n_1\in\mathbb{Z},\zeta'\in(\mathbb{Z}\cup\mathbb{B})^*.\,\zeta\neq n_0::n_1::\zeta'}{(\ell,\zeta,\sigma)\succ(\ell,\mathsf{add})\twoheadrightarrow(\ell,\zeta,\sigma)}\ \mathrm{add}^{ab}_{\mathrm{ns}}$$

$$\ldots$$

$$\left[\begin{array}{c}\dfrac{(m,\zeta,\sigma)\succ(\ell,\mathsf{goto}\ m)\rightarrow(\ell',\zeta',\sigma')}{(\ell,\zeta,\sigma)\succ(\ell,\mathsf{goto}\ m)\rightarrow(\ell',\zeta',\sigma')}\\[2ex]\dfrac{(m,\zeta,\sigma)\succ(\ell,\mathsf{goto}\ m)\twoheadrightarrow(\ell',\zeta',\sigma')}{(\ell,\zeta,\sigma)\succ(\ell,\mathsf{goto}\ m)\twoheadrightarrow(\ell',\zeta',\sigma')}\end{array}\right]\qquad\frac{m\neq\ell}{(\ell,\zeta,\sigma)\succ(\ell,\mathsf{goto}\ m)\rightarrow(m,\zeta,\sigma)}\ \mathrm{goto}^{\neq}_{\mathrm{ns}}$$

$$\left[\begin{array}{c}\dfrac{}{(\ell,\mathrm{tt}::\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ m)\rightarrow(\ell+1,\zeta,\sigma)}\\[2ex]\dfrac{(m,\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ m)\rightarrow(\ell',\zeta',\sigma')}{(\ell,\mathrm{ff}::\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ m)\rightarrow(\ell',\zeta',\sigma')}\\[2ex]\dfrac{(m,\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ m)\twoheadrightarrow(\ell',\zeta',\sigma')}{(\ell,\mathrm{ff}::\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ m)\twoheadrightarrow(\ell',\zeta',\sigma')}\\[2ex]\dfrac{\forall b\in\mathbb{B},\zeta'\in(\mathbb{Z}\cup\mathbb{B})^*.\,\zeta\neq b::\zeta'}{(\ell,\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ m)\twoheadrightarrow(\ell,\zeta,\sigma)}\end{array}\right]$$

$$\frac{m\neq\ell}{(\ell,\mathrm{tt}::\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ m)\rightarrow(\ell+1,\zeta,\sigma)}\ \mathrm{gotoF}^{\neq\mathrm{tt}}_{\mathrm{ns}}$$

$$\frac{m\neq\ell}{(\ell,\mathrm{ff}::\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ m)\rightarrow(m,\zeta,\sigma)}\ \mathrm{gotoF}^{\neq\mathrm{ff}}_{\mathrm{ns}}$$

$$\frac{m\neq\ell\quad\forall b\in\mathbb{B},\zeta'\in(\mathbb{Z}\cup\mathbb{B})^*.\,\zeta\neq b::\zeta'}{(\ell,\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ m)\twoheadrightarrow(\ell,\zeta,\sigma)}\ \mathrm{gotoF}^{\neq ab}_{\mathrm{ns}}$$

$$\frac{\mathit{ffs}\in\{\mathrm{ff}\}^*}{(\ell,\mathit{ffs}+\!\!+\mathrm{tt}::\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ \ell)\rightarrow(\ell+1,\zeta,\sigma)}\ \mathrm{gotoF}^{=}_{\mathrm{ns}}$$

$$\frac{\mathit{ffs}\in\{\mathrm{ff}\}^*\quad\forall b\in\mathbb{B},\zeta'\in(\mathbb{Z}\cup\mathbb{B})^*.\,\zeta\neq b::\zeta'}{(\ell,\mathit{ffs}+\!\!+\zeta,\sigma)\succ(\ell,\mathsf{gotoF}\ \ell)\twoheadrightarrow(\ell,\zeta,\sigma)}\ \mathrm{gotoF}^{=ab}_{\mathrm{ns}}$$

$$\frac{\ell\in\mathrm{dom}(sc_i)\quad(\ell,\zeta,\sigma)\succ sc_i\rightarrow(\ell'',\zeta'',\sigma'')\quad(\ell'',\zeta'',\sigma'')\succ sc_0\oplus sc_1\rightarrow(\ell',\zeta',\sigma')}{(\ell,\zeta,\sigma)\succ sc_0\oplus sc_1\rightarrow(\ell',\zeta',\sigma')}\ \oplus_{\mathrm{ns}}$$

$$\frac{\ell\in\mathrm{dom}(sc_i)\quad(\ell,\zeta,\sigma)\succ sc_i\twoheadrightarrow(\ell',\zeta',\sigma')}{(\ell,\zeta,\sigma)\succ sc_0\oplus sc_1\twoheadrightarrow(\ell',\zeta',\sigma')}\ \oplus^{abn}_{\mathrm{ns}}$$

$$\frac{\ell\in\mathrm{dom}(sc_i)\quad(\ell,\zeta,\sigma)\succ sc_i\rightarrow(\ell'',\zeta'',\sigma'')\quad(\ell'',\zeta'',\sigma'')\succ sc_0\oplus sc_1\twoheadrightarrow(\ell',\zeta',\sigma')}{(\ell,\zeta,\sigma)\succ sc_0\oplus sc_1\twoheadrightarrow(\ell',\zeta',\sigma')}\ \oplus^{abl}_{\mathrm{ns}}$$

$$\frac{\ell\notin\mathrm{dom}(sc)}{(\ell,\zeta,\sigma)\succ sc\rightarrow(\ell,\zeta,\sigma)}\ \mathrm{ood}_{\mathrm{ns}}$$

Figure 2: Natural semantics rules of SPush

We have spelled out the rules for goto m and gotoF m instructions in two different ways: a recursive style (in square brackets) and a direct style. The two styles are equivalent, but we comment only the direct style. The recursive style could be seen as a formal explanation of the direct style. The issue is that, differently from other single-instruction pieces of code, a goto or gotoF instruction can loop back on itself. This happens when the labelling label and the target label coincide.

The side condition in the $\text{goto}^{\neq}_{\text{ns}}$ rule states that a goto m instruction only terminates, if it does not loop back on itself. The $\text{gotoF}^{\neq\text{tt}}_{\text{ns}}$ rule should be self-explanatory, however the gotoF m rules for the case there is a ff on the top of the stack should be explained. The complication here is that just like goto m, gotoF m can loop back on itself. Unlike goto m however, it cannot loop infinitely, since every successful jump removes an element from the stack. Instead it can either exit the loop at some point (when it encounters a tt on top of the stack), or cause an error if it either encounters an integer on the stack or the stack runs empty. Therefore, two rules ($\text{gotoF}^{=}_{\text{ns}}$ and $\text{gotoF}^{=ab}_{\text{ns}}$) are needed for normal and abnormal behavior of gotoF m for the case when it loops back on itself. The rule $\text{gotoF}^{\neq ab}_{\text{ns}}$ covers the case when there is no boolean value at the top of the stack.

The rule $\oplus_{\text{ns}}$ says that, to evaluate the union $sc_0 \oplus sc_1$ starting from some state such that the pc is in the domain of sc_i, one first needs to evaluate sc_i, and then evaluate the whole union again, but starting from the new intermediate state reached. Finally, the ood_{ns} rule is needed to reflect the case where the reduction sequence is normally terminated because the pc has landed outside the domain of the code.

It is fairly straightforward that the pc in the final state of a normally terminating evaluation of a code is outside its domain while the pc in the final state of an abnormally terminating evaluation is inside. Evaluation is deterministic in the sense that any piece of code terminates either normally or abnormally in a definite state, if it terminates at all.

Every SPush piece of code can be mapped into a Push piece of code using a forgetful function $U \in \mathbf{SCode} \to \mathbf{Code}$, defined by $U((\ell, instr)) =_{\text{df}} \{(\ell, instr)\}$, $U(\mathbf{0}) =_{\text{df}} \emptyset$, $U(sc_0 \oplus sc_1) =_{\text{df}} U(sc_0) \oplus U(sc_1)$. The compositional natural semantics of SPush agrees with the non-compositional semantics of Push in the following technical sense.

Theorem 1 ***(Preservation of evaluations by*** U***)*** *(i) If* $(\ell,\zeta,\sigma) \succ sc \to (\ell',\zeta',\sigma')$*, then* $U(sc) \vdash (\ell,\zeta,\sigma) \twoheadrightarrow^* (\ell',\zeta',\sigma') \not\twoheadrightarrow$ *and* $\ell' \notin \text{dom}(sc)$*. (ii) If* $(\ell,\zeta,\sigma) \succ sc \twoheadrightarrow (\ell',\zeta',\sigma')$*, then* $U(sc) \vdash (\ell,\zeta,\sigma) \twoheadrightarrow^* (\ell',\zeta',\sigma') \not\twoheadrightarrow$ *and* $\ell' \in \text{dom}(sc)$*.*

Proof. By induction on the derivation of $(\ell,\zeta,\sigma) \succ sc \to (\ell',\zeta',\sigma')$ or $(\ell,\zeta,\sigma) \succ sc \twoheadrightarrow (\ell',\zeta',\sigma')$. □

Theorem 2 ***(Reflection of stuck reduction sequences by*** U***)*** *(i) If* $U(sc) \vdash (\ell,\zeta,\sigma) \twoheadrightarrow^* (\ell',\zeta',\sigma') \not\twoheadrightarrow$ *and* $\ell' \notin \text{dom}(sc)$*, then* $(\ell,\zeta,\sigma) \succ sc \to (\ell',\zeta',\sigma')$*. (ii) If* $U(sc) \vdash (\ell,\zeta,\sigma) \twoheadrightarrow^* (\ell',\zeta',\sigma') \not\twoheadrightarrow$ *and* $\ell' \in \text{dom}(sc)$*, then* $(\ell,\zeta,\sigma) \succ sc \twoheadrightarrow (\ell',\zeta',\sigma')$

Proof. By induction on the structure of sc and subordinate induction on the length of the reduction sequence. □

From these theorems it is immediate that the SPush semantics of a structured version of a piece of Push code cannot depend on the way it is structured: if $U(sc) = U(sc')$, then sc and sc' have exactly the same evaluations (although with different derivations).

4 Hoare logic

The compositional natural semantics of SPush is a good basis for developing a compositional Hoare logic of it. Just as evaluations relate an initial and a terminal state, Hoare triples relate pre- and postconditions about states. Since a state contains a pc value and stack content, it must be possible to refer to these in assertions. In our logic, we have special individual constants pc and st to refer to them. Using the constant pc, we can make assertions about particular program points by constraining the state to correspond to a certain pc value. This allows us to make assertions only about program points through which the particular piece of code is entered or exited, thus eliminating the need for global contexts of invariants and making reasoning modular.

The logic we define is an error-free partial correctness logic: for a Hoare triple to be derivable, the postcondition must be satisfied by the terminal state of any normal evaluation and abnormal evaluations from the allowed initial states must be impossible. (We would get a more expressive partial correctness logic with triples with two postconditions, one for normal terminations, the other for abnormal terminations; in the case of a programming language with error-handling constructs, that approach is the only reasonable one, see, e.g., (Schröder & Mossakowski 2004). Our logic corresponds to the case where the abnormal postcondition is always $\bot$, so there is no need to ever spell it out. A different version where it is always $\top$ would correspond to error-ignoring partial correctness.)

The signature of the Hoare logic contains, as extra-arithmetical and extra-list constants, special individual constants pc, st and the program variables **Var**, to refer to the values of the program counter, stack and program variables in a state. The assertions $P, Q \in \mathbf{Assn}$ are formulae over that signature in an ambient logical language containing the signature of arithmetic and lists of integers and booleans. We use the notation $Q[x_0,..,x_n \mapsto t_0,..,t_n]$ to denote that every occurrence of x_i in Q has been replaced with t_i. The derivable Hoare triples $\{\} - \{\} \subseteq \mathbf{Assn} \times \mathbf{SCode} \times \mathbf{Assn}$ are defined inductively by the rules in Figure 3.

The extra disjunct $pc \neq \ell \wedge Q$ in the rules for primitive instructions is required because of the semantic rule ood_{ns}: if we evaluate the instruction starting from outside the domain of the instruction (i.e. $pc \neq \ell$), we have immediately terminated and have hence remained in the same state, therefore any assertion holding before evaluating the instruction will also hold after. The disjunct $m = \ell$ in the rule for goto m accounts for the case when goto m loops back on itself. We have a similar case with the gotoF m rule, but here the situation is more subtle. As explained in Section 3, when gotoF m loops back on itself, it can either exit normally to the next instruction (in case there is some number of ffs on the stack, followed by a tt), or raise an error. The disjunct $m = \ell \wedge ..$ accounts for that case.

The rule for unions can be seen as mix of the while and sequence rules of the Hoare logic of While: if, evaluating either sc_0 or sc_1 starting from a state that satisfies P and has the pc value in the domain of sc_0 resp. sc_1, we end in a state satisfying P, then, after evaluating their union $sc_0 \oplus sc_1$ starting from a state satisfying P, we are guaranteed to be in a state satisfying P. Furthermore, we know that we are then outside the domains of both sc_0 and sc_1. The rule of consequence is the same as in the standard Hoare logic. Note that we have circumvented the inevitable *incompleteness* of any axiomatization of logics containing arithmetic by invoking semantic entailment

instead of deducibility in the premises of the conseq rule.

The Hoare logic is sound and complete.

Theorem 3 ***(Soundness of Hoare logic)*** *If* $\{P\}\, sc\, \{Q\}$ *and* $(\ell, \zeta, \sigma) \models_\alpha P$*, then (i) for any* (ℓ', ζ', σ') *such that* $(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell', \zeta', \sigma')$*, we have* $(\ell', \zeta', \sigma') \models_\alpha Q$*, and (ii) there is no* (ℓ', ζ', σ') *such that* $(\ell, \zeta, \sigma) \succ sc \twoheadrightarrow (\ell', \zeta', \sigma')$*.*

Proof. By induction on the derivation of $\{P\}\, sc\, \{Q\}$. □

To get completeness, we have to assume that the underlying logical language is *expressive*. For any assertion Q, we need an assertion $\mathrm{wlp}(sc, Q)$ that, semantically, is its weakest precondition, i.e., for any state (ℓ, ζ, σ) and valuation α of free variables, we have $(\ell, \zeta, \sigma) \models \mathrm{wlp}(sc, Q)$ iff $(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell', \zeta', \sigma')$ implies $(\ell', \zeta', \sigma') \models Q$ for any (ℓ', ζ', σ'). The wlp function is available for example when the underlying logical language has a greatest fixedpoint operator.

Lemma 1 $\{\mathrm{wlp}(sc, Q)\}\, sc\, \{Q\}$.

Proof. By induction on the structure of sc. □

Theorem 4 ***(Completeness of Hoare logic)*** *If, for any* (ℓ, ζ, σ) *and* α *such that* $(\ell, \zeta, \sigma) \models_\alpha P$*, it holds that (i) for any* (ℓ', ζ', σ') *such that* $(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell', \zeta', \sigma')$*, we have* $(\ell', \zeta', \sigma') \models_\alpha Q$*, and (ii) there is no* (ℓ', ζ', σ') *such that* $(\ell, \zeta, \sigma) \succ sc \twoheadrightarrow (\ell', \zeta', \sigma')$*, then* $\{P\}\, sc\, \{Q\}$*.*

Proof. Immediate from the lemma using that any precondition of an assertion entails its wlp. □

5 Abstract natural semantics

We now proceed to defining an abstract natural semantics for SPush that operates on type names as abstract values instead of concrete values. This allows us to later prove a type system for basic safety sound and complete. While soundness of the type system could also be shown wrt. the concrete natural semantics, completeness cannot.

The abstract semantics is defined in terms of abstract states, which are pairs of labels $\ell \in \mathbf{Label}$ and abstract stack contents $\psi \in \mathbf{AbsStack}$: $\mathbf{AbsState} =_{\mathrm{df}} \mathbf{Label} \times \mathbf{AbsStack}$. Instead of values, abstract stacks stack names of value types: $\mathbf{AbsStack} =_{\mathrm{df}} \{\mathrm{int}, \mathrm{bool}\}^*$. We do not have an abstract store component in an abstract state. Since variables can only be integers in a concrete store, there is no interesting information to record. To relate a concrete state to an abstract state, we have a function $\mathrm{abs} \in \mathbf{State} \to \mathbf{AbsState}$, defined by $\mathrm{abs}(\ell, \zeta, \sigma) =_{\mathrm{df}} (\ell, \mathrm{abs}(\zeta))$ where $\mathrm{abs} \in \mathbf{Stack} \to \mathbf{AbsStack}$ replaces concrete values in a stack with the names of their types: $\mathrm{abs}([]) =_{\mathrm{df}} []$, $\mathrm{abs}(n :: \zeta) =_{\mathrm{df}} \mathrm{int} :: \mathrm{abs}(\zeta)$ for $n \in \mathbb{Z}$, and $\mathrm{abs}(b :: \zeta) =_{\mathrm{df}} \mathrm{bool} :: \mathrm{abs}(\zeta)$ for $b \in \mathbb{B}$.

The abstract semantics is a rather straightforward rewrite of the concrete semantics to work on abstract states, but it is important to notice that this makes evaluation nondeterministic. Just like in the concrete semantics, we need to distinguish between abnormal and normal evaluations, so there are two evaluation relations $\succ - \rightarrow, \succ - \twoheadrightarrow \subseteq \mathbf{AbsState} \times \mathbf{SCode} \times \mathbf{AbsState}$. The rules of the abstract natural semantics are given in Figure 4. Mimicking those of the concrete semantics from Figure 2, they should be self-explanatory. As before, we have spelled out the rules for goto and gotoF in two alternative styles, recursive and direct. The nondeterminism stems from the non-exclusive rules of gotoF.

Concrete evaluations are preserved by abstraction.

Theorem 5 ***(Preservation of evaluations by abstraction)*** *(i) If* $(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell', \zeta', \sigma')$*, then* $\mathrm{abs}(\ell, \zeta, \sigma) \succ sc \rightarrow \mathrm{abs}(\ell', \zeta', \sigma')$*.*
(ii) If $(\ell, \zeta, \sigma) \succ sc \twoheadrightarrow (\ell', \zeta', \sigma')$*, then* $\mathrm{abs}(\ell, \zeta, \sigma) \succ sc \twoheadrightarrow \mathrm{abs}(\ell', \zeta', \sigma')$*.*

6 Type system from the Hoare logic

With the abstract semantics defined, we are now ready to show that the Hoare logic we have formulated for SPush can be weakened into a type system for establishing basic code safety—the absence of operand type and stack underflow errors in an Push program. The abstract semantics allows us to prove the type system not only sound, but also complete.

Instead of relating assertions as Hoare triples do, typings relate state types. The intuitive meaning of a typing is analogous to that of a Hoare triple: it says that if the given piece of code is run from an initial state in the given pretype, then if it terminates normally, the final state is in the posttype, and, moreover, it cannot terminate abnormally. Contrarily to assertions, state types are designed to record only that state information that is necessary for guaranteeing error-freedom.

The building blocks for state types are value types $\tau \in \mathbf{ValType}$ and stack types $\Psi \in \mathbf{StackType}$, defined by the grammars

$$\begin{array}{rcl} \tau & ::= & \bot \mid \mathrm{int} \mid \mathrm{bool} \mid ? \\ \Psi & ::= & \bot \mid [] \mid \tau :: \Psi \mid * \end{array}$$

(note the overloading of the $\bot$ sign). A state type $\Pi \in \mathbf{StateType}$ is a finite set of labelled stack types, i.e., pairs of a label and a stack type: $\mathbf{StateType} =_{\mathrm{df}} \mathcal{P}_{\mathrm{fin}}(\mathbf{Label} \times \mathbf{StackType})$. A state type Π is well-formed iff no label ℓ in it labels more than one stack type, i.e., $(\ell, \Psi) \in \Pi$ and $(\ell, \Psi') \in \Pi$ imply $\Psi = \Psi'$. The domain $\mathrm{dom}(\Pi)$ of a state type is the set of labels appearing in it, i.e., $\mathrm{dom}(\Pi) =_{\mathrm{df}} \{\ell \mid (\ell, \Psi) \in \Pi\}$.

We will use the notation $\Pi\restriction_L$ for the restriction of a state type Π to a domain $L \subseteq \mathbf{Label}$, i.e., $\Pi\restriction_L =_{\mathrm{df}} \{(\ell, \Psi) \mid (\ell, \Psi) \in \Pi, \ell \in L\}$, and write $\overline{L}$ for the complement of L, i.e., $\overline{L} =_{\mathrm{df}} \mathbf{Label} \backslash L$.

The meanings of value, stack and state types are set-theoretic, they denote sets of abstract values, abstract stacks and abstract states. The semantic functions $(\!|-|\!) \in \mathbf{ValType} \to \mathcal{P}(\{\mathrm{int}, \mathrm{bool}\})$, $(\!|-|\!) \in \mathbf{StackType} \to \mathcal{P}(\mathbf{AbsStack})$, $(\!|-|\!) \in \mathbf{StateType} \to \mathcal{P}(\mathbf{AbsState})$ are defined as follows:

$$\begin{array}{rcl} (\!|\bot|\!) & =_{\mathrm{df}} & \emptyset \\ (\!|\mathrm{int}|\!) & =_{\mathrm{df}} & \{\mathrm{int}\} \\ (\!|\mathrm{bool}|\!) & =_{\mathrm{df}} & \{\mathrm{bool}\} \\ (\!|?|\!) & =_{\mathrm{df}} & \{\mathrm{int}, \mathrm{bool}\} \\ (\!|\bot|\!) & =_{\mathrm{df}} & \emptyset \\ (\!|[]|\!) & =_{\mathrm{df}} & \{[]\} \\ (\!|\tau :: \Psi|\!) & =_{\mathrm{df}} & \{\delta :: \psi \mid \delta \in (\!|\tau|\!), \psi \in (\!|\Psi|\!)\} \\ (\!|*|\!) & =_{\mathrm{df}} & \{\mathrm{int}, \mathrm{bool}\}^* \\ (\!|\Pi|\!) & =_{\mathrm{df}} & \{(\ell, \psi) \mid (\ell, \Psi) \in \Pi, \psi \in (\!|\Psi|\!)\} \end{array}$$

On each of the three categories of types, we define a subtyping relation by the rules in Figure 5. These are relations $\leq \subseteq \mathbf{ValType} \times \mathbf{ValType}$, $\leq \subseteq \mathbf{StackType} \times \mathbf{StackType}$, $\leq \subseteq \mathbf{StateType} \times \mathbf{StateType}$.

$$\frac{}{\{(pc = \ell \wedge Q[pc, st \mapsto \ell + 1, x :: st]) \vee (pc \neq \ell \wedge Q)\}\,(\ell, \mathsf{load}\ x)\,\{Q\}}\ \mathrm{load}_{\mathrm{hoa}}$$

$$\frac{}{\{(pc = \ell \wedge \exists z \in \mathbb{Z}, w \in (\mathbb{Z} \cup \mathbb{B})^*.\ st = z :: w \wedge Q[pc, st, x \mapsto \ell + 1, w, z]) \vee (pc \neq \ell \wedge Q)\}\,(\ell, \mathsf{store}\ x)\,\{Q\}}\ \mathrm{store}_{\mathrm{hoa}}$$

$$\frac{}{\{(pc = \ell \wedge Q[pc, st \mapsto \ell + 1, n :: st]) \vee (pc \neq \ell \wedge Q)\}\,(\ell, \mathsf{push}\ n)\,\{Q\}}\ \mathrm{push}_{\mathrm{hoa}}$$

$$\frac{}{\left\{\begin{array}{ll} & (pc = \ell \wedge \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z} \cup \mathbb{B})^*.\ st = z_0 :: z_1 :: w \wedge Q[pc, st \mapsto \ell + 1, z_0 + z_1 :: w]) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\}\,(\ell, \mathsf{add})\,\{\ Q\ \}}\ \mathrm{add}_{\mathrm{hoa}}$$

$$\cdots$$

$$\frac{}{\left\{\begin{array}{ll} & (pc = \ell \wedge ((m \neq \ell \wedge Q[pc \mapsto m]) \vee m = \ell)) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\}\,(\ell, \mathsf{goto}\ m)\,\{\ Q\ \}}\ \mathrm{goto}_{\mathrm{hoa}}$$

$$\frac{}{\left\{\begin{array}{lll} & (pc = \ell & \wedge((m \neq \ell \quad \wedge((\exists w \in (\mathbb{Z} \cup \mathbb{B})^*.\ st = \mathrm{tt} :: w \wedge Q[pc, st \mapsto \ell + 1, w]) \\ & & \quad\ \vee(\exists w \in (\mathbb{Z} \cup \mathbb{B})^*.\ st = \mathrm{ff} :: w \wedge Q[pc, st \mapsto m, w]))) \\ & & \vee(m = \ell \wedge \exists \mathit{ffs} \in \{\mathrm{ff}\}^*, w \in (\mathbb{Z} \cup \mathbb{B})^*.st = \mathit{ffs} +\!\!+ \mathrm{tt} :: w \wedge Q[pc, st \mapsto \ell + 1, w]))) \\ \vee & (pc \neq \ell & \wedge Q) \end{array}\right\}\,(\ell, \mathsf{gotoF}\ m)\,\{\ Q\ \}}\ \mathrm{gotoF}_{\mathrm{hoa}}$$

$$\frac{}{\{P\}\,\mathbf{0}\,\{P\}}\ \mathbf{0}_{\mathrm{hoa}} \qquad \frac{\{pc \in \mathrm{dom}(sc_0) \wedge P\}\,sc_0\,\{P\} \quad \{pc \in \mathrm{dom}(sc_1) \wedge P\}\,sc_1\,\{P\}}{\{P\}\,sc_0 \oplus sc_1\,\{pc \notin \mathrm{dom}(sc_0) \wedge pc \notin \mathrm{dom}(sc_1) \wedge P\}}\ \oplus_{\mathrm{hoa}}$$

$$\frac{P \models P' \quad \{P'\}\,sc\,\{Q'\} \quad Q' \models Q}{\{P\}\,sc\,\{Q\}}\ \mathrm{conseq}_{\mathrm{hoa}}$$

Figure 3: Hoare rules of SPush

$$\frac{}{(\ell, \psi) \succ (\ell, \mathsf{load}\ x) \rightarrow (\ell + 1, \mathrm{int} :: \psi)}\ \mathrm{load}_{\mathrm{ans}}$$

$$\frac{}{(\ell, \mathrm{int} :: \psi) \succ (\ell, \mathsf{store}\ x) \rightarrow (\ell + 1, \psi)}\ \mathrm{store}_{\mathrm{ans}} \qquad \frac{\forall \psi' \in \{\mathrm{int}, \mathrm{bool}\}^*.\ \psi \neq \mathrm{int} :: \psi'}{(\ell, \psi) \succ (\ell, \mathsf{store}\ x) \twoheadrightarrow (\ell, \psi)}\ \mathrm{store}^{ab}_{\mathrm{ans}}$$

$$\frac{}{(\ell, \psi) \succ (\ell, \mathsf{push}\ n) \rightarrow (\ell + 1, \mathrm{int} :: \psi)}\ \mathrm{push}_{\mathrm{ans}}$$

$$\frac{}{(\ell, \mathrm{int} :: \mathrm{int} :: \psi) \succ (\ell, \mathsf{add}) \rightarrow (\ell + 1, \mathrm{int} :: \psi)}\ \mathrm{add}_{\mathrm{ans}} \qquad \frac{\forall \psi' \in \{\mathrm{int}, \mathrm{bool}\}^*.\ \psi \neq \mathrm{int} :: \mathrm{int} :: \psi'}{(\ell, \psi) \succ (\ell, \mathsf{add}) \twoheadrightarrow (\ell, \psi)}\ \mathrm{add}^{ab}_{\mathrm{ans}}$$

$$\cdots$$

$$\left[\begin{array}{l} \dfrac{(m, \psi) \succ (\ell, \mathsf{goto}\ m) \rightarrow (\ell', \psi')}{(\ell, \psi) \succ (\ell, \mathsf{goto}\ m) \rightarrow (\ell', \psi')} \\ \dfrac{(m, \psi) \succ (\ell, \mathsf{goto}\ m) \twoheadrightarrow (\ell', \psi')}{(\ell, \psi) \succ (\ell, \mathsf{goto}\ m) \twoheadrightarrow (\ell', \psi')} \end{array}\right] \qquad \frac{m \neq \ell}{(\ell, \psi) \succ (\ell, \mathsf{goto}\ m) \rightarrow (m, \psi)}\ \mathrm{goto}^{\neq}_{\mathrm{ans}}$$

$$\left[\begin{array}{l} \dfrac{}{(\ell, \mathrm{bool} :: \psi) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell + 1, \psi)} \\ \dfrac{(m, \psi) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell', \psi')}{(\ell, \mathrm{bool} :: \psi) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell', \psi')} \\ \dfrac{(m, \psi) \succ (\ell, \mathsf{gotoF}\ m) \twoheadrightarrow (\ell', \psi')}{(\ell, \mathrm{bool} :: \psi) \succ (\ell, \mathsf{gotoF}\ m) \twoheadrightarrow (\ell', \psi')} \\ \dfrac{\forall \psi' \in \{\mathrm{int}, \mathrm{bool}\}^*.\ \psi \neq \mathrm{bool} :: \psi'}{(\ell, \psi) \succ (\ell, \mathsf{gotoF}\ m) \twoheadrightarrow (\ell, \psi)} \end{array}\right]$$

$$\frac{m \neq \ell}{(\ell, \mathrm{bool} :: \psi) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell + 1, \psi)}\ \mathrm{gotoF}^{\neq \mathrm{tt}}_{\mathrm{ans}}$$

$$\frac{m \neq \ell}{(\ell, \mathrm{bool} :: \psi) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (m, \psi)}\ \mathrm{gotoF}^{\neq \mathrm{ff}}_{\mathrm{ans}}$$

$$\frac{m \neq \ell \quad \forall \psi' \in \{\mathrm{int}, \mathrm{bool}\}^*.\ \psi \neq \mathrm{bool} :: \psi'}{(\ell, \psi) \succ (\ell, \mathsf{gotoF}\ m) \twoheadrightarrow (\ell, \psi)}\ \mathrm{gotoF}^{\neq ab}_{\mathrm{ans}}$$

$$\frac{\mathit{bools} \in \{\mathrm{bool}\}^*}{(\ell, \mathit{bools} +\!\!+ \mathrm{bool} :: \psi) \succ (\ell, \mathsf{gotoF}\ \ell) \rightarrow (\ell + 1, \psi)}\ \mathrm{gotoF}^{=}_{\mathrm{ans}}$$

$$\frac{\mathit{bools} \in \{\mathrm{bool}\}^* \quad \forall \psi' \in \{\mathrm{int}, \mathrm{bool}\}^*.\ \psi \neq \mathrm{bool} :: \psi'}{(\ell, \mathit{bools} +\!\!+ \psi) \succ (\ell, \mathsf{gotoF}\ \ell) \twoheadrightarrow (\ell, \psi)}\ \mathrm{gotoF}^{=ab}_{\mathrm{ans}}$$

$$\frac{\ell \in \mathrm{dom}(sc_i) \quad (\ell, \psi) \succ sc_i \rightarrow (\ell'', \psi'') \quad (\ell'', \psi'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', \psi')}{(\ell, \psi) \succ sc_0 \oplus sc_1 \rightarrow (\ell', \psi')}\ \oplus_{\mathrm{ans}}$$

$$\frac{\ell \in \mathrm{dom}(sc_i) \quad (\ell, \psi) \succ sc_i \twoheadrightarrow (\ell', \psi')}{(\ell, \psi) \succ sc_0 \oplus sc_1 \twoheadrightarrow (\ell', \psi')}\ \oplus^{abn}_{\mathrm{ans}}$$

$$\frac{\ell \in \mathrm{dom}(sc_i) \quad (\ell, \psi) \succ sc_i \rightarrow (\ell'', \psi'') \quad (\ell'', \psi'') \succ sc_0 \oplus sc_1 \twoheadrightarrow (\ell', \psi')}{(\ell, \psi) \succ sc_0 \oplus sc_1 \twoheadrightarrow (\ell', \psi')}\ \oplus^{abl}_{\mathrm{ans}}$$

$$\frac{\ell \notin \mathrm{dom}(sc)}{(\ell, \psi) \succ sc \rightarrow (\ell, \psi)}\ \mathrm{ood}_{\mathrm{ans}}$$

Figure 4: Abstract natural semantics rules of SPush

$$\frac{}{\tau \le \tau} \quad \frac{}{\bot \le \tau} \quad \frac{}{\tau \le ?}$$

$$\frac{}{\Psi \le \Psi} \quad \frac{\Psi \le \Psi'' \quad \Psi'' \le \Psi'}{\Psi \le \Psi'} \quad \frac{}{\bot :: \Psi \le \bot} \quad \frac{}{\tau :: \bot \le \bot} \quad \frac{}{\bot \le \Psi} \quad \frac{}{\Psi \le *} \quad \frac{\tau \le \tau' \quad \Psi \le \Psi'}{\tau :: \Psi \le \tau' :: \Psi'}$$

$$\frac{\forall \ell, \Psi.\, (\ell, \Psi) \in \Pi \supset \Psi = \bot \vee \exists \Psi'.\, (\ell, \Psi') \in \Pi' \wedge \Psi \le \Psi'}{\Pi \le \Pi'}$$

Figure 5: Subtyping rules of SPush

The subtyping relations thus introduced are sound and complete for the intended interpretation of subtyping as set inclusion.

Theorem 6 ***(Soundness and completeness of subtyping)*** *(i)* $\tau \le \tau'$ *iff* $(\!|\tau|\!) \subseteq (\!|\tau'|\!)$. *(ii)* $\Psi \le \Psi'$ *iff* $(\!|\Psi|\!) \subseteq (\!|\Psi'|\!)$. *(iii)* $\Pi \le \Pi'$ *iff* $(\!|\Pi|\!) \subseteq (\!|\Pi'|\!)$.

Very pleasantly, the ranges $\mathcal{P}(\{\mathrm{int}, \mathrm{bool}\})$, $\{(\!|\Psi|\!) \mid \Psi \in \mathbf{StackType}\}$, $\{(\!|\Pi|\!) \mid \Pi \in \mathbf{StateType}\}$ of each of the three type interpretation functions are ω-complete lower semilattices with inclusion as the underlying partial order: set-theoretic binary intersections and intersections of nonincreasing ω-chains do not take us out of the range. (Note that the analogous statement about unions is not true, e.g., the set $(\!|[]|\!) \cup (\!|\mathrm{int} :: []|\!)$ has no type denotation. Note also that there are nonincreasing ω-chains that do not stabilize in a finite number of steps, e.g., $*$, $\mathrm{int} :: *$, $\mathrm{int} :: \mathrm{int} :: *$, ..., but all such chains have $\bot$ as their glb.) Because of the soundness and completeness of subtyping, we can reflect this at the syntactic level: we can define a syntatic binary glb operator $\wedge$ on types and a syntactic glb operator $\bigwedge$ on deductively nonincreasing ω-sequences of types that are glb operators deductively ('deductively' meaning 'in the sense of the subtyping relation').

The typing relation $- : - \longrightarrow - \subseteq \mathbf{StateType} \times \mathbf{SCode} \times \mathbf{StateType}$ is defined by the rules in Figure 6. The typing rules for instructions are presented in a "weakest pretype" style, where the pretype is obtained by applying appropriate substitutions in the given posttype. For example the rule $\mathrm{load}_{\mathrm{ts}}$ for $(\ell, \mathsf{load}\ x)$ states that if stack type $\tau :: \Psi$ (where τ is int or ?) or $*$ is required at label $\ell + 1$, then the suitable stack types for label ℓ are Ψ and $*$, respectively. Any other posttype at label $\ell + 1$ does not have a suitable pretype. At first sight, it might seem that wellformedness can be lost in the pretype by taking the union. This is in fact not the case: there is at most one stack type associated with label $\ell + 1$ in Π, hence both sets have at most one element and one of them must be empty. The rest of the non-jump instruction rules are defined in similar fashion.

The jump rules might need some explanation. The $\mathrm{goto}^{=}_{\mathrm{ts}}$ rule allows to derive pretype $*$ for label ℓ: since the instruction does not terminate, any posttype will be satisfied by any pretype at label ℓ. The $\mathrm{gotoF}^{\neq}_{\mathrm{ts}}$ rule combines two posttypes; since gotoF can branch, both posttypes must be satisfied at the entry, meaning that the pretype is the intersection of the posttypes. No pretype at ℓ can guarantee any posttype in the case of $(\ell, \mathsf{gotoF}\ \ell)$, since such instruction could always terminate abnormally. The consequence rule could also be called subsumption, given that we are speaking about a type system: that is what it is really.

The type system is sound and complete wrt. the abstract natural semantics in the sense of error-free partial correctness.

Theorem 7 ***(Soundness of typing)*** *If* $sc : \Pi \longrightarrow \Pi'$ *and* $(\ell, \psi) \in (\!|\Pi|\!)$*, then (i) for any* (ℓ', ψ') *such that* $(\ell, \psi) \succ sc \rightarrow (\ell', \psi')$*, we have* $(\ell', \psi') \in (\!|\Pi'|\!)$*, and (ii) there is no* (ℓ', ψ') *such that* $(\ell, \psi) \succ sc \rightarrow\!\!\!| (\ell', \psi')$.

Proof. By induction on the derivation of $sc : \Pi \longrightarrow \Pi'$, using that subtyping is sound. □

From the preservation of evaluations by abstraction, it is immediate that therewith we also have soundness wrt. the concrete natural semantics.

Corollary 1 *If* $sc : \Pi \longrightarrow \Pi'$ *and* $\mathrm{abs}(\ell, \zeta, \sigma) \in (\!|\Pi|\!)$*, then (i) for any* (ℓ', ζ', σ') *such that* $(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell', \zeta', \sigma')$*, we have* $\mathrm{abs}(\ell', \zeta', \sigma') \in (\!|\Pi'|\!)$*, and (ii) there is no* (ℓ', ζ', σ') *such that* $(\ell, \zeta, \sigma) \succ sc \rightarrow\!\!\!| (\ell', \zeta', \sigma')$.

To prove completeness, we introduce a syntactic pretype function $\mathrm{wpt} \in \mathbf{SCode} \times \mathbf{StateType} \to \mathbf{StateType}$. The definition is given in Figure 7. The ω-sequence glb in the clause for $\oplus$ is welldefined because the operator S is monotone, making the sequence a nonincreasing chain. As S is also continuous, the glb is the greatest fixedpoint of S.

The following lemmata show that the wpt of a state type is semantically larger than any pretype and deductively (i.e., in the sense of typing) a pretype.

Lemma 2 *If (i) for any* (ℓ', ψ') *such that* $(\ell, \psi) \succ sc \rightarrow (\ell', \psi')$*, we have* $(\ell', \psi') \in (\!|\Pi'|\!)$*, and (ii) there is no* (ℓ', ψ') *such that* $(\ell, \psi) \succ sc \rightarrow\!\!\!| (\ell', \psi')$*, then* $(\ell, \psi) \in (\!|\mathrm{wpt}(sc, \Pi')|\!)$.

Lemma 3 $sc : \mathrm{wpt}(sc, \Pi') \longrightarrow \Pi'$.

Proof. By induction on the structure of sc. □

Theorem 8 ***(Completeness of typing)*** *If, for any* $(\ell, \psi) \in (\!|\Pi|\!)$*, it holds that (i) for any* (ℓ', ψ') *such that* $(\ell, \psi) \succ sc \rightarrow (\ell', \psi')$*, we have* $(\ell', \psi') \in (\!|\Pi'|\!)$*, and (ii) there is no* (ℓ', ψ') *such that* $(\ell, \psi) \succ sc \rightarrow\!\!\!| (\ell', \psi')$*, then* $sc : \Pi \longrightarrow \Pi'$.

Proof. From the two lemmata, using that subtyping is complete. □

It is fairly obvious that state types can be translated to assertions. We can define concretization functions $\mathrm{conc} \in \mathbf{ValType} \to \mathcal{P}(\mathbb{Z} \cup \mathbb{B})$, $\mathrm{conc} \in \mathbf{StackType} \to \mathcal{P}(\mathbf{Stack})$, $\mathrm{conc} \in \mathbf{StateType} \to \mathbf{Assn}$, taking us from the language of the type system to the language of the logic, by

$$\begin{aligned}
\mathrm{conc}(\bot) &=_{\mathrm{df}} \emptyset \\
\mathrm{conc}(\mathrm{int}) &=_{\mathrm{df}} \mathbb{Z} \\
\mathrm{conc}(\mathrm{bool}) &=_{\mathrm{df}} \mathbb{B} \\
\mathrm{conc}(?) &=_{\mathrm{df}} \mathbb{Z} \cup \mathbb{B} \\
\mathrm{conc}(\bot) &=_{\mathrm{df}} \emptyset \\
\mathrm{conc}([]) &=_{\mathrm{df}} \{[]\} \\
\mathrm{conc}(\tau :: \Psi) &=_{\mathrm{df}} \{z :: w \mid z \in \mathrm{conc}(\tau), w \in \mathrm{conc}(\Psi)\} \\
\mathrm{conc}(*) &=_{\mathrm{df}} (\mathbb{Z} \cup \mathbb{B})^* \\
\mathrm{conc}(\Pi) &=_{\mathrm{df}} \\
&\quad \bigvee \{pc = \ell \wedge st \in \mathrm{conc}(\Psi) \mid (\ell, \Psi) \in \Pi\}
\end{aligned}$$

Concretization preserves and reflects derivable subtypings/entailments.

$$\frac{}{(\ell, \mathsf{load}\ x) : \{(\ell, \Psi) \mid (\ell+1, \tau :: \Psi) \in \Pi, \mathrm{int} \leq \tau\} \cup \{(\ell, *) \mid (\ell+1, *) \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi}\ \mathrm{load}_{\mathrm{ts}}$$

$$\frac{}{(\ell, \mathsf{store}\ x) : \{(\ell, \mathrm{int} :: \Psi) \mid (\ell+1, \Psi) \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi}\ \mathrm{store}_{\mathrm{ts}}$$

$$\frac{}{(\ell, \mathsf{push}\ n) : \{(\ell, \Psi) \mid (\ell+1, \tau :: \Psi) \in \Pi, \mathrm{int} \leq \tau\} \cup \{(\ell, *) \mid (\ell+1, *) \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi}\ \mathrm{push}_{\mathrm{ts}}$$

$$\frac{}{(\ell, \mathsf{add}) : \{(\ell, \mathrm{int} :: \mathrm{int} :: \Psi) \mid (\ell+1, \tau :: \Psi) \in \Pi, \mathrm{int} \leq \tau\} \cup \{(\ell, \mathrm{int} :: \mathrm{int} :: *) \mid (\ell+1, *) \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi}\ \mathrm{add}_{\mathrm{ts}}$$

$$\cdots$$

$$\frac{m \neq \ell}{(\ell, \mathsf{goto}\ m) : \{(\ell, \Psi) \mid (m, \Psi) \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi}\ \mathrm{goto}^{\neq}_{\mathrm{ts}} \qquad \frac{}{(\ell, \mathsf{goto}\ \ell) : \{(\ell, *)\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi}\ \mathrm{goto}^{=}_{\mathrm{ts}}$$

$$\frac{m \neq \ell}{(\ell, \mathsf{gotoF}\ m) : \{(\ell, \mathrm{bool} :: (\Psi \wedge \Psi')) \mid (\ell+1, \Psi), (m, \Psi') \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi}\ \mathrm{gotoF}^{\neq}_{\mathrm{ts}} \qquad \frac{}{(\ell, \mathsf{gotoF}\ \ell) : \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi}\ \mathrm{gotoF}^{=}_{\mathrm{ts}}$$

$$\frac{}{\mathbf{0} : \Pi \longrightarrow \Pi}\ \mathbf{0}_{\mathrm{ts}} \qquad \frac{sc_0 : \Pi\restriction_{\mathrm{dom}(sc_0)} \longrightarrow \Pi \quad sc_1 : \Pi\restriction_{\mathrm{dom}(sc_1)} \longrightarrow \Pi}{sc_0 \oplus sc_1 : \Pi \longrightarrow \Pi\restriction_{\overline{\mathrm{dom}(sc_0) \cup \mathrm{dom}(sc_1)}}}\ \oplus_{\mathrm{ts}}$$

$$\frac{\Pi'_0 \leq \Pi_0 \quad sc : \Pi_0 \longrightarrow \Pi_1 \quad \Pi_1 \leq \Pi'_1}{sc : \Pi'_0 \longrightarrow \Pi'_1}\ \mathrm{conseq}_{\mathrm{ts}}$$

Figure 6: Typing rules of SPush

$$\begin{array}{rcl}
\mathrm{wpt}((\ell, \mathsf{load}\ x), \Pi') & =_{\mathrm{df}} & \{(\ell, \Psi) \mid (\ell+1, \tau :: \Psi) \in \Pi', \mathrm{int} \leq \tau\} \cup \{(\ell, *) \mid (\ell+1, *) \in \Pi'\} \cup \Pi'\restriction_{\overline{\{\ell\}}} \\
\mathrm{wpt}((\ell, \mathsf{store}\ x), \Pi') & =_{\mathrm{df}} & \{(\ell, \mathrm{int} :: \Psi) \mid (\ell+1, \Psi) \in \Pi'\} \cup \Pi'\restriction_{\overline{\{\ell\}}} \\
\mathrm{wpt}((\ell, \mathsf{push}\ n), \Pi') & =_{\mathrm{df}} & \{(\ell, \Psi) \mid (\ell+1, \tau :: \Psi) \in \Pi', \mathrm{int} \leq \tau\} \cup \{(\ell, *) \mid (\ell+1, *) \in \Pi'\} \cup \Pi'\restriction_{\overline{\{\ell\}}} \\
\mathrm{wpt}((\ell, \mathsf{add}), \Pi') & =_{\mathrm{df}} & \{(\ell, \mathrm{int} :: \mathrm{int} :: \Psi) \mid (\ell+1, \tau :: \Psi) \in \Pi', \mathrm{int} \leq \tau\} \cup \{(\ell, \mathrm{int} :: \mathrm{int} :: *) \mid (\ell+1, *) \in \Pi'\} \cup \Pi'\restriction_{\overline{\{\ell\}}} \\
\mathrm{wpt}((\ell, \mathsf{goto}\ m), \Pi') & =_{\mathrm{df}} & \begin{cases} \{(\ell, \Psi) \mid (m, \Psi) \in \Pi'\} \cup \Pi'\restriction_{\overline{\{\ell\}}} & \text{if } m \neq \ell \\ \{(\ell, *)\} \cup \Pi'\restriction_{\overline{\{\ell\}}} & \text{if } m = \ell \end{cases} \\
\mathrm{wpt}((\ell, \mathsf{gotoF}\ m), \Pi') & =_{\mathrm{df}} & \begin{cases} \{(\ell, \mathrm{bool} :: (\Psi \wedge \Psi')) \mid (\ell+1, \Psi), (m, \Psi') \in \Pi'\} \cup \Pi'\restriction_{\overline{\{\ell\}}} & \text{if } m \neq \ell \\ \Pi'\restriction_{\overline{\{\ell\}}} & \text{if } m = \ell \end{cases} \\
\mathrm{wpt}(\mathbf{0}, \Pi') & =_{\mathrm{df}} & \Pi' \\
\mathrm{wpt}(sc_0 \oplus sc_1, \Pi') & =_{\mathrm{df}} & \bigwedge_{i<\omega} \Pi_i \text{ where} \\
 & & \begin{array}{lcl} \Pi_0 & =_{\mathrm{df}} & \{(\ell, *) \mid \ell \in \mathrm{dom}(sc_0 \oplus sc_1) \cup \mathrm{dom}(\Pi')\} \\ \Pi_{i+1} & =_{\mathrm{df}} & S(\Pi_i) \\ S(\Pi) & =_{\mathrm{df}} & \mathrm{wpt}(sc_0, \Pi)\restriction_{\mathrm{dom}(sc_0)} \cup \mathrm{wpt}(sc_1, \Pi)\restriction_{\mathrm{dom}(sc_1)} \cup \Pi'\restriction_{\overline{\mathrm{dom}(sc_0 \oplus sc_1)}} \end{array}
\end{array}$$

Figure 7: Weakest pretype calculus

Theorem 9 ***(Preservation of subtypings and reflection of entailments by concretization)*** *(i)* $\tau \leq \tau'$ *iff* $\mathrm{conc}(\tau) \models \mathrm{conc}(\tau')$. *(ii)* $\Psi \leq \Psi'$ *iff* $\mathrm{conc}(\Psi) \models \mathrm{conc}(\Psi')$. *(iii)* $\Pi \leq \Pi'$ *iff* $\mathrm{conc}(\Pi) \models \mathrm{conc}(\Pi')$.

Preservation holds also of typing.

Theorem 10 ***(Preservation of typings by concretization)*** *If* $sc : \Pi \longrightarrow \Pi'$, *then* $\{\mathrm{conc}(\Pi)\}\, sc\, \{\mathrm{conc}(\Pi')\}$.

We do not get reflection of Hoare triples by concretization, however. Consider, for example, the code $sc =_{\mathrm{df}} (0, \mathsf{push}\ \mathrm{tt}) \oplus ((1, \mathsf{gotoF}\ 3) \oplus (2, \mathsf{push}\ 17))$. We have $\mathrm{conc}((0, [])) = pc = 0 \wedge st = []$, $\mathrm{conc}((3, [\mathrm{int}])) = pc = 3 \wedge \exists z \in \mathbb{Z}.\ st = [z]$ and can derive $\{pc = 0 \wedge st = []\}\, sc\, \{pc = 3 \wedge \exists z \in \mathbb{Z}.\ st = [z]\}$, while we cannot derive $sc : \{(0, [])\} \longrightarrow \{(3, [\mathrm{int}])\}$. The type system does not discover that the false branch will never be taken. The best posttype we can get for $\{(0, [])\}$ is $\{(3, *)\}$.

We finish the discussion of the type system by remarking that introducing the value type ? and the stack type $*$ was not inevitable. But a version without these constructs would only type pieces of code for which the operand stack has a definite depth and value type content for every label through which its evaluations may pass. More generally, there is a design issue here. We could, for example, introduce additional stack types int^*, bool^* for stacks of unspecified length, consisting of integers or booleans only. Yet another design choice would be to define $\mathbf{StackType} =_{\mathrm{df}} \mathcal{P}_{\mathrm{fin}}(\mathbf{AbsStack})$ instead. Under this discipline, some pieces of code with finitely unbalanced stack usage would receive more precise types, e.g., for the code

```
0  gotoF 3
1  push 17
2  goto 5
3  push tt
4  push ff
```

and pretype $\{(0, [\mathrm{bool}])\}$, the best posttype we can get in our type system is $\{(5, ? :: *)\}$, but the alternative posttype $\{(5, \{[\mathrm{int}], [\mathrm{bool}, \mathrm{bool}]\})\}$ is clearly more informative. On the other hand, a piece of code with infinite variation such as

```
0  load x
1  geq0
2  gotoF 8
3  push 17
4  load x
5  dec
6  store x
7  goto 0
```

and the pretype $\{(0, [])\}$ have $\{(8, *)\}$ as the strongest posttype in our type system but no posttype under the alternative approach.

7 Compilation

We shall now define a compilation function from While programs to SPush pieces of code.

The compilation function is standard except that it produces structured code (we have chosen structures that are the most convenient for us) and is compositional. The compilation rules are given in Figure 8. The compilation relation for expressions $- \searrow - \subseteq \mathbf{Label} \times (\mathbf{AExp} \cup \mathbf{BExp}) \times \mathbf{SCode} \times \mathbf{Label}$ relates a label and a While expression to a piece of code and another label. The relation for statements $- \searrow - \subseteq \mathbf{Label} \times \mathbf{Stm} \times \mathbf{SCode} \times \mathbf{Label}$ is similar. The idea is that the domain of a compiled expression or statement will be a left-closed, right-open interval. (It may be an empty interval, which does not even contain its beginning-point.) The first label is the beginning-point of the interval and the second is the corresponding end-point.

Compilation is total and deterministic, i.e., a function, and produces a piece of code whose support is exactly the desired interval.

Lemma 4 ***(Totality and determinacy of compilation)*** *(i) For any* ℓ, e, *there exist* sc, ℓ' *such that* $e \overset{\ell}{\searrow}_{\ell'} sc$. *If* $e \overset{\ell}{\searrow}_{\ell_0} sc_0$ *and* $e \overset{\ell}{\searrow}_{\ell_1} sc_1$, *then* $sc_0 = sc_1$ *and* $\ell_0 = \ell_1$. *(ii) For any* ℓ, s, *there exist* sc, ℓ' *such that* $s \overset{\ell}{\searrow}_{\ell'} sc$. *If* $s \overset{\ell}{\searrow}_{\ell_0} sc_0$ *and* $s \overset{\ell}{\searrow}_{\ell_1} sc_1$, *then* $sc_0 = sc_1$ *and* $\ell_0 = \ell_1$.

Lemma 5 ***(Domain of compiled code)*** *(i) If* $e \overset{\ell}{\searrow}_{\ell'} sc$, *then* $\mathrm{dom}(sc) = [\ell, \ell')$. *(ii) If* $s \overset{\ell}{\searrow}_{\ell'} sc$, *then* $\mathrm{dom}(sc) = [\ell, \ell')$.

That compilation does not alter the meaning of an expression or statement is demonstrated by the facts that While evaluations are preserved and SPush evaluations are reflected by it. We must however take into account the fact a compiled While expression or statement is intended to be entered from its beginning-point.

Theorem 11 ***(Preservation of evaluations)*** *(i) If* $e \overset{\ell}{\searrow}_{\ell'} sc$, *then* $(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell', [\![e]\!]\sigma :: \zeta, \sigma)$. *(ii) If* $s \overset{\ell}{\searrow}_{\ell'} sc$ *and* $\sigma \succ s \rightarrow \sigma'$, *then* $(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell', \zeta, \sigma')$.

Proof. By induction on the structure of e or the derivation of $\sigma \succ s \rightarrow \sigma'$. □

Theorem 12 ***(Reflection of evaluations)*** *(i) If* $e \overset{\ell}{\searrow}_{\ell'} sc$ *and* $(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell'', \zeta', \sigma')$, *then* $\ell'' = \ell'$, $\zeta' = [\![e]\!]\sigma :: \zeta$ *and* $\sigma' = \sigma$. *(ii) If* $s \overset{\ell}{\searrow}_{\ell'} sc$ *and* $(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell'', \zeta', \sigma')$, *then* $\ell'' = \ell'$, $\zeta' = \zeta$ *and* $\sigma \succ s \rightarrow \sigma'$.

Proof. By induction on the structure of sc and subordinate induction on the derivation of $(\ell, \zeta, \sigma) \succ sc \rightarrow (\ell'', \zeta', \sigma')$. □

It is easy to show that compilation preserves derivable While Hoare triples (in a suitable format that takes into account that a While statement proof assumes entry from the beginning-point and guarantees exit to the end-point). But one can also give a constructive proof: a proof by defining a compositional translation of While program proofs to SPush program proofs, i.e., a proof compilation function.

Theorem 13 ***(Preservation of derivable Hoare triples)*** *(i) If* $e \overset{\ell}{\searrow}_{\ell'} sc$ *and* P *is a* While *assertion, then* $\{pc = \ell \wedge st = \zeta \wedge P\}\, sc\, \{pc = \ell' \wedge st = e :: \zeta \wedge P\}$. *(ii) If* $s \overset{\ell}{\searrow}_{\ell'} sc$ *and* $\{P\}\, s\, \{Q\}$, *then* $\{pc = \ell \wedge st = \zeta \wedge P\}\, sc\, \{pc = \ell' \wedge st = \zeta \wedge Q\}$.

Proof. [Non-constructive proof] Straightforward from soundness of the Hoare logic of While, reflection of evaluations by compilation and completeness of the Hoare logic of SPush. □

Proof. [Constructive proof: Preservation Hoare triple derivations] By induction on the structure of e or the derivation of $\{P\}\, s\, \{Q\}$. □

Reflection of derivable SPush Hoare triples by compilation can also be shown. As with preservation,

proving reflection non-constructively is a straightforward matter, but again there is also a constructive proof. Given a WHILE program, we can "decompile" the correctness proof of its compiled form (a SPUSH piece of code) into a correctness proof of the WHILE program. For the constructive proof, we have to use the fact that proofs of SPUSH programs admit a certain normal form.

Theorem 14 ***(Reflection of derivable Hoare triples)*** *(i) If* $e \,{}^{\ell}\!\searrow_{\ell'} sc$ *and* $\{P\}\, sc\, \{Q\}$*, then* $P[pc, st \mapsto \ell, \zeta] \models Q[pc, st \mapsto \ell', e :: \zeta]$*. (ii) If* $s \,{}^{\ell}\!\searrow_{\ell'} sc$ *and* $\{P\}\, sc\, \{Q\}$*, then* $\{P[pc, st \mapsto \ell, \zeta]\}\, s\, \{Q[pc, st \mapsto \ell', \zeta]\}$*.*

Proof. [Non-constructive proof] From soundness of the Hoare logic of SPUSH, preservation of evaluations by compilation and completeness of the Hoare logic of WHILE. □

Proof. [Constructive proof: Reflection of Hoare triple derivations] By induction on the structure of *sc*, using the fact that any Hoare logic derivation can be normalized to a form where proper inferences come in strict alternation with consequence inferences. (Normalization is trivial: a sequence of several consecutive consequence inferences can be compressed into one and a missing consequence inference can be expanded into a trivial consequence inference.) □

For the type system of SPUSH, we can prove the following analogous results. The first of them means that we can strengthen our compilation function to accompany the SPUSH code it produces from a WHILE-program with a typing derivation.

Theorem 15 ***(Typing from compilation)*** *(i) If* $a \,{}^{\ell}\!\searrow_{\ell'} sc$*, then* $sc : \{(\ell, \psi)\} \longrightarrow \{(\ell', \mathrm{int} :: \psi)\}$*. If* $b \,{}^{\ell}\!\searrow_{\ell'} sc$*, then* $sc : \{(\ell, \psi)\} \longrightarrow \{(\ell', \mathrm{bool} :: \psi)\}$*. (ii) If* $s \,{}^{\ell}\!\searrow_{\ell'} sc$*, then* $sc : \{(\ell, \psi)\} \longrightarrow \{(\ell', \psi)\}$*.*

Theorem 16 ***(Possible typings)*** *(i) If* $a \,{}^{\ell}\!\searrow_{\ell'} sc$ *and* $sc : \{(\ell, \psi)\} \longrightarrow \Pi$*, then* $\{(\ell', \mathrm{int} :: \psi)\} \leq \Pi$*. If* $b \,{}^{\ell}\!\searrow_{\ell'} sc$ *and* $sc : \{(\ell, \psi)\} \longrightarrow \Pi$*, then* $\{(\ell', \mathrm{bool} :: \psi)\} \leq \Pi$*. (ii) If* $s \,{}^{\ell}\!\searrow_{\ell'} sc$ *and* $sc : \{(\ell, \psi)\} \longrightarrow \Pi$*, then* $\{(\ell', \psi)\} \leq \Pi$*.*

8 Abstract natural semantics and type system for secure information flow

Besides stack-error freedom, it is possible to devise systems to present dataflow analyses. Here we sketch an abstract natural semantics and type system for secure information flow analysis. For space reasons, this description is very Spartan, but it should make sense to anyone familiar with secure information flow analyses for high-level imperative languages à la Denning & Denning (1977).

Central for both the abstract natural semantics and type system for secure information flow is a distributive lattice $(\mathrm{D}, \leq, \wedge, \vee, \mathrm{L}, \mathrm{H})$ of security levels for information flowing in the program (stack positions, variables and the pc). Abstract states are quadruples of a label $\ell \in \mathbf{Label}$, a security level $d \in \mathrm{D}$ for the current pc value, and an abstract stack and an abstract store: $\mathbf{AbsState} =_{\mathrm{df}} \mathbf{Label} \times \mathrm{D} \times \mathbf{AbsStack} \times \mathbf{AbsStore}$. An abstract stack $\psi \in \mathbf{AbsStack}$ is a list over D corresponding to the security levels of the stack positions in the imaginable concrete state, an abstract store $\Sigma \in \mathbf{AbsStore}$ similarly records the security levels of the variables in the imaginable concrete state: $\mathbf{AbsStack} =_{\mathrm{df}} \mathrm{D}^*$, $\mathbf{AbsStore} =_{\mathrm{df}} \mathbf{Var} \to \mathrm{D}$.

The abstract semantics is sensitive to stack underflow, but ignores the possibility in the concrete semantics of operand type errors (confuses them with normal terminations). An important concept in the semantics is the notion of a single-exit piece of code: this is a piece of code *sc* for which one can single out a label ℓ^* such that every target label (successor label or jump target, depending on the kind of the instruction) of any labelled instruction in *sc* is in $\mathrm{dom}(sc) \cup \{\ell^*\}$; we call ℓ^* the exit-point of *sc*. Single-exit unions are analogous to single-exit compound blocks in control-flow diagrams; compare these to if- or while-statements of WHILE, which are single-exit as all WHILE statements but special in that their control-flow diagrams enclose inner branchings. The rules of the semantics are presented in Figure 9. Because of the single-exit union rule, this abstract semantics is not neutral wrt. the structure imposed on an unstructured PUSH piece of code: depending on how small or large the smallest single-exit union enclosing a branching instruction gotoF is in the structure imposed on a code, a given initial security state can take us to a more or less optimistic terminal security state.

In the type system, the state types $\Pi \in \mathbf{StateType}$ are quadruples of a label, security level (for the pc), stack type and abstract store (there is no difference between an abstract store and a store type!): $\mathbf{StateType} =_{\mathrm{df}} \mathcal{P}_{\mathrm{fin}}(\mathbf{Label} \times \mathrm{D} \times \mathbf{StackType} \times \mathbf{AbsStore})$ where no label may occur twice in a wellformed statetype. Stack types $\Psi \in \mathbf{StackType}$ are defined by the grammar

$$\Psi ::= \bot \mid [] \mid d :: \Psi \mid *$$

Stack types have a set-theoretic meaning defined as follows:

$$\begin{aligned} (\!| \bot |\!) &=_{\mathrm{df}} \emptyset \\ (\!| [] |\!) &=_{\mathrm{df}} \{[]\} \\ (\!| d :: \Psi |\!) &=_{\mathrm{df}} \{d' :: \psi \mid d' \leq d, \psi \in (\!| \Psi |\!)\} \\ (\!| * |\!) &=_{\mathrm{df}} \mathrm{D}^* \end{aligned}$$

The type system is derived from the abstract natural semantics—the typing rules are in the weakest pretype style—and attests stack-underflow-error free information flow security. The type system may type pieces of code that can terminate abnormally due to wrong operand types. The subtyping rules are in Figure 10 while the typing rules appear in Figure 11. ($\psi \vee d$ denotes the list resulting from joining d to every element of ψ; $\bigvee ds$ denotes the join of all elements of ds; $\bigwedge \Psi$ denotes the meet of all elements of Ψ.)

9 Related work

In the young days of Hoare logic, quite some attention was paid to general and restricted jumps in high-level languages. Hoare's original logic (1969) was for WHILE and characteristic to the various proposals that were made thereafter is that they deal with extensions of WHILE or a similar language. The logics of Clint & Hoare (1972), Kowaltowski (1977) and de Bruin (1981) use conditional Hoare triples (so the proof system is a natural deduction system) to be able to make and use assumptions about label invariants. In the solution of Arbib & Alagić (1979), Hoare triples have multiple postconditions, reflecting the fact that statements involving gotos are multiple-exit.

Logics for low-level languages without phrase structure have only become a topic of active research with the advent of PCC, with Java bytecode and .NET CIL being the main motivators. (There is

$$\frac{}{n \overset{\ell}{\searrow}_{\ell+1} (\ell, \mathsf{push}\ n)} \quad \frac{}{x \overset{\ell}{\searrow}_{\ell+1} (\ell, \mathsf{load}\ x)} \quad \frac{a_0 \overset{\ell}{\searrow}_{\ell''} sc_0 \quad a_1 \overset{\ell''}{\searrow}_{\ell'} sc_1}{a_0 + a_1 \overset{\ell}{\searrow}_{\ell'+1} (sc_0 \oplus sc_1) \oplus \mathsf{add}} \quad \frac{b_0 \overset{\ell}{\searrow}_{\ell''} sc_0 \quad b_1 \overset{\ell''}{\searrow}_{\ell'} sc_1}{b_0 = b_1 \overset{\ell}{\searrow}_{\ell'+1} (sc_0 \oplus sc_1) \oplus \mathsf{eq}}$$

$$\frac{a \overset{\ell}{\searrow}_{\ell'} sc}{x := a \overset{\ell}{\searrow}_{\ell'+1} (sc \oplus \mathsf{store}\ x)} \quad \frac{}{\mathsf{skip} \overset{\ell}{\searrow}_{\ell} \mathbf{0}} \quad \frac{s_0 \overset{\ell}{\searrow}_{\ell''} sc_0 \quad s_1 \overset{\ell''}{\searrow}_{\ell'} sc_1}{s_0; s_1 \overset{\ell}{\searrow}_{\ell'} sc_0 \oplus sc_1}$$

$$\frac{b \overset{\ell}{\searrow}_{\ell''} sc_b \quad s_t \overset{\ell''+1}{\searrow}_{\ell'''} sc_t \quad s_f \overset{\ell'''+1}{\searrow}_{\ell'} sc_f}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \overset{\ell}{\searrow}_{\ell'} (sc_b \oplus (\ell'', \mathsf{gotoF}\ \ell''' + 1)) \oplus ((sc_t \oplus (\ell''', \mathsf{goto}\ \ell')) \oplus sc_f)}$$

$$\frac{b \overset{\ell}{\searrow}_{\ell''} sc_b \quad s \overset{\ell''+1}{\searrow}_{\ell'} sc}{\mathsf{while}\ b\ \mathsf{do}\ s \overset{\ell}{\searrow}_{\ell'+1} (sc_b \oplus (\ell'', \mathsf{gotoF}\ \ell' + 1)) \oplus (sc \oplus (\ell', \mathsf{goto}\ \ell))}$$

Figure 8: Rules of compilation from WHILE to SPUSH

$$\frac{}{(\ell, d, \psi, \Sigma) \succ (\ell, \mathsf{load}\ x) \rightarrow (\ell + 1, d, \Sigma(x) \vee d :: \psi, \Sigma)}\ \mathrm{load}_{\mathrm{ans}}$$

$$\frac{}{(\ell, d, d' :: \psi, \Sigma) \succ (\ell, \mathsf{store}\ x) \rightarrow (\ell + 1, d, \psi, \Sigma[x \mapsto d' \vee d])}\ \mathrm{store}_{\mathrm{ans}} \quad \frac{\forall d \in \mathrm{D}, \psi' \in \mathrm{D}^*.\ \psi \neq d :: \psi'}{(\ell, d, \psi, \Sigma) \succ (\ell, \mathsf{store}\ x) \twoheadrightarrow (\ell, d, \psi, \Sigma)}\ \mathrm{store}^{ab}_{\mathrm{ans}}$$

$$\frac{}{(\ell, d, \psi, \Sigma) \succ (\ell, \mathsf{push}\ n) \rightarrow (\ell + 1, d, d :: \psi, \Sigma)}\ \mathrm{push}_{\mathrm{ans}}$$

$$\frac{}{(\ell, d, d_0 :: d_1 :: \psi, \Sigma) \succ (\ell, \mathsf{add}) \rightarrow (\ell + 1, d, d_0 \vee d_1 \vee d :: \psi, \Sigma)}\ \mathrm{add}_{\mathrm{ans}} \quad \frac{\forall d_0, d_1 \in \mathrm{D}, \psi' \in \mathrm{D}^*.\ \psi \neq d_0 :: d_1 :: \psi'}{(\ell, d, \psi, \Sigma) \succ (\ell, \mathsf{add}) \twoheadrightarrow (\ell, d, \psi, \Sigma)}\ \mathrm{add}^{ab}_{\mathrm{ans}}$$

$$\frac{m \neq \ell}{(\ell, d, \psi, \Sigma) \succ (\ell, \mathsf{goto}\ m) \rightarrow (m, d, \psi, \Sigma)}\ \mathrm{goto}^{\neq}_{\mathrm{ans}}$$

$$\frac{m \neq \ell}{(\ell, d, d' :: \psi, \Sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell + 1, d \vee d', \psi \vee (d \vee d'), \Sigma)}\ \mathrm{gotoF}^{\neq \mathrm{tt}}_{\mathrm{ans}}$$

$$\frac{m \neq \ell}{(\ell, d, d' :: \psi, \Sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (m, d \vee d', \psi \vee (d \vee d'), \Sigma)}\ \mathrm{gotoF}^{\neq \mathrm{ff}}_{\mathrm{ans}}$$

$$\frac{m \neq l \quad \forall d \in \mathrm{D}, \psi' \in \mathrm{D}^*.\ \psi \neq d :: \psi'}{(\ell, d, \psi, \Sigma) \succ (\ell, \mathsf{gotoF}\ m) \twoheadrightarrow (\ell, d, \psi, \Sigma)}\ \mathrm{gotoF}^{\neq ab}_{\mathrm{ans}}$$

$$\frac{ds \in \mathrm{D}^*}{(\ell, d, ds +\!\!+ d' :: \psi, \Sigma) \succ (\ell, \mathsf{gotoF}\ \ell) \rightarrow (\ell + 1, d, \psi \vee (d \vee \bigvee ds \vee d'), \Sigma)}\ \mathrm{gotoF}^{=}_{\mathrm{ans}} \quad \frac{ds \in \mathrm{D}^*}{(\ell, d, ds, \Sigma) \succ (\ell, \mathsf{gotoF}\ \ell) \twoheadrightarrow (\ell + 1, d, [], \Sigma)}\ \mathrm{gotoF}^{=ab}_{\mathrm{ans}}$$

$$\frac{\ell \in \mathrm{dom}(sc_i) \quad (\ell, d, \psi, \Sigma) \succ sc_i \rightarrow (\ell'', d'', \psi'', \Sigma'') \quad (\ell'', d'', \psi'', \Sigma'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', d', \psi', \Sigma') \quad sc_0 \oplus sc_1\ \text{multiple-exit}}{(\ell, d, \psi, \Sigma) \succ sc_0 \oplus sc_1 \rightarrow (\ell', d', \psi', \Sigma')}\ \oplus_{\mathrm{ans}}$$

$$\frac{\ell \in \mathrm{dom}(sc_i) \quad (\ell, d, \psi, \Sigma) \succ sc_i \rightarrow (\ell'', d'', \psi'', \Sigma'') \quad (\ell'', d'', \psi'', \Sigma'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', d', \psi', \Sigma') \quad sc_0 \oplus sc_1\ \text{single-exit}}{(\ell, d, \psi, \Sigma) \succ sc_0 \oplus sc_1 \rightarrow (\ell', d, \psi', \Sigma')}\ \oplus_{\mathrm{ans}}$$

$$\frac{\ell \in \mathrm{dom}(sc_i) \quad (\ell, d, \psi, \Sigma) \succ sc_i \twoheadrightarrow (\ell'', d, \psi'', \Sigma'')}{(\ell, d, \psi, \Sigma) \succ sc_0 \oplus sc_1 \twoheadrightarrow (\ell', d', \psi', \Sigma')}\ \oplus^{abn}_{\mathrm{ans}}$$

$$\frac{\ell \in \mathrm{dom}(sc_i) \quad (\ell, d, \psi, \Sigma) \succ sc_i \rightarrow (\ell'', d, \psi'', \Sigma'') \quad (\ell'', d'', \psi'', \Sigma'') \succ sc_0 \oplus sc_1 \twoheadrightarrow (\ell', d', \psi', \Sigma')}{(\ell, d, \psi, \Sigma) \succ sc_0 \oplus sc_1 \twoheadrightarrow (\ell', d', \psi', \Sigma')}\ \oplus_{\mathrm{ans}}$$

$$\frac{\ell \notin \mathrm{dom}(sc)}{(\ell, d, \psi, \Sigma) \succ sc \rightarrow (\ell, d, \psi, \Sigma)}\ \mathrm{ood}_{\mathrm{ans}}$$

Figure 9: Abstract natural semantics rules of SPUSH for secure information flow

$$\frac{}{\Psi \leq \Psi} \quad \frac{\Psi \leq \Psi'' \quad \Psi'' \leq \Psi'}{\Psi \leq \Psi'} \quad \frac{}{\tau :: \bot \leq \bot} \quad \frac{}{\bot \leq \Psi} \quad \frac{}{\Psi \leq *} \quad \frac{\tau \leq \tau' \quad \Psi \leq \Psi'}{\tau :: \Psi \leq \tau' :: \Psi'}$$

$$\frac{\forall x.\ \Sigma(x) \leq \Sigma'(x)}{\Sigma \leq \Sigma'}$$

$$\frac{\forall \ell, d, \Psi, \Sigma.\ (\ell, d, \Psi, \Sigma) \in \Pi \supset \Psi = \bot \vee \exists \Psi'.\ (\ell, d', \Psi', \Sigma') \in \Pi' \wedge d \leq d' \wedge \Psi \leq \Psi' \wedge \Sigma \leq \Sigma'}{\Pi \leq \Pi'}$$

Figure 10: Subtyping rules of SPUSH for secure information flow

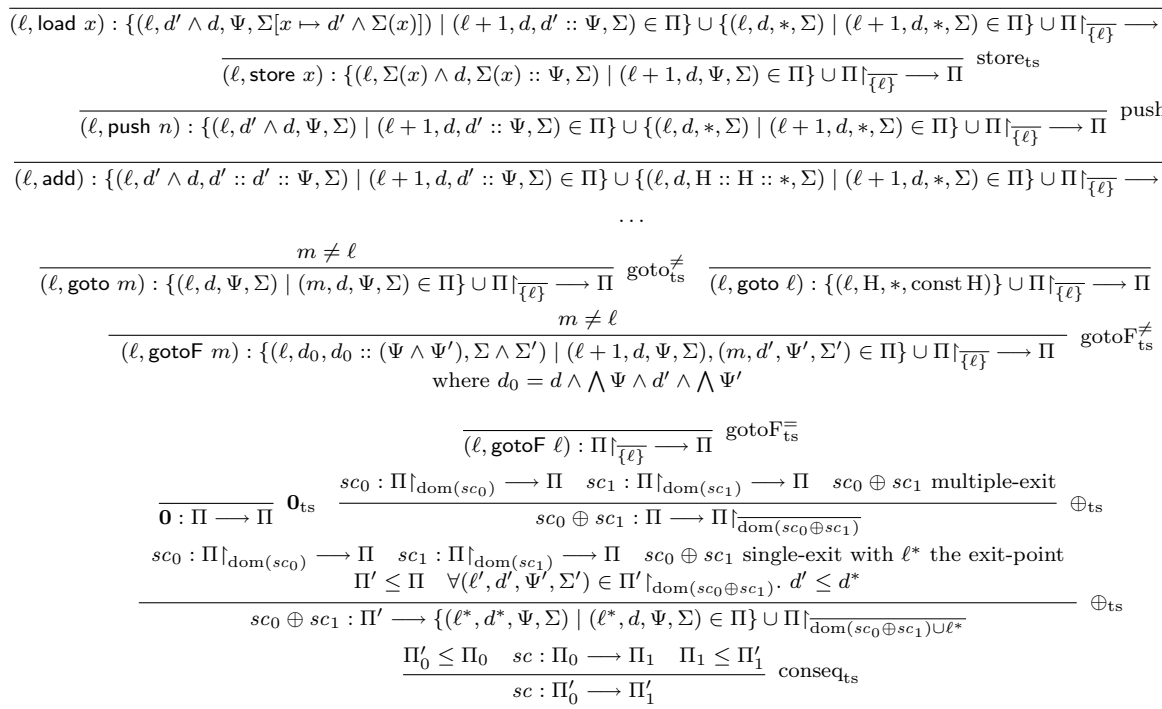

Figure 11: Typing rules of SPush for secure information flow

one very notable exception though: Floyd's logic of control-flow graphs (1967).) The logic of Quigley (2003) for Java bytecode is based on decompilation, so it applies to pieces of code in the image of a fixed compiler. Benton's (2004) logic for a Push-like stack-based language involves global contexts of label invariants as de Bruin's logic. Bannwart & Müller's (2005) logic extends it to a subset of Java bytecode, with both an operand stack and a call stack, leaving out exceptions.

The work of Huisman & Jacobs (2000) describes a Hoare logic for Java, incl. exceptions. Schröder & Mossakowski (2003) and Schröder & Mossakowski (2004) discuss a systematic method for designing Hoare logics for languages with monadic side-effects, in particular, exceptions.

The present paper builds upon our recent work (Saabas & Uustalu 2005), where a compositional natural semantics and Hoare logic based on the implicit finite unions structure are introduced for a simple low-level language Goto with expressions. The same structure is used by Tan & Appel (2005) and Tan (2005), who study the same language. But instead of introducing a natural semantics for the structured version of the language, they proceed from a small-step ideology. As a result, they arrive at a continuation-style Hoare logic explainable by Appel & McAllester's 'indexed model' (2001), with a rather convoluted interpretation of Hoare triples involving explicit fixedpoint approximations. Apparently unaware of Tan & Appel's work, Benton (2005) defines a similar logic for a stack-based language with a typing component ensuring that the stack is used safely.

Presenting program analyses especially for functional languages in terms of type systems is a popular topic. Naik & Palsberg (2005) have related model checking and type systems for While. A different general method to produce type systems for While equivalent to dataflow analyses is described in the work of Laud et al. (2005). As for low-level languages, Morrisett et al. (1999) imposed a memory-safety type system on an assembly language and Morrisett et al. (2003) extended it for a stack-based language. Stata & Abadi (1999) were the first to describe the Java bytecode verifier as a type system. All such systems are again non-compositional and make use of global contexts of label invariants (where an invariant is associated to every instruction or every basic block of the global piece of code), except for the type system component in Benton's (2005) logic.

A static analysis for secure information flow was first described by Denning & Denning (1977). They worked with a While-like language, but also proposed a way to handle languages with goto instructions. Kobayashi & Kirane (2002) and Barthe & Rezk (2005) use the same idea of control dependence regions in type systems equivalent to secure information flow analyses for sequential Java bytecode.

10 Conclusions and future work

We have shown that our original idea of structuring low-level languages with finite unions to obtain compositional natural semantics and Hoare logics (Saabas & Uustalu 2005) applies to stack-based languages just as well as to languages with store only. The possibility of abnormal terminations can be handled well, and the semantics and logics obtained are neat and enjoy every desirable metatheoretic property. Moreover, in the richer setting of a stack-based language, it is meaningful to consider abstracted semantics and type systems too. Notably, one can obtain a type system to attest safe stack usage, but also produce type systems for other purposes. We have demonstrated this on the example of a type system equivalent to a secure information flow analysis.

We plan to apply the method also to a language with both an operand stack and call stack, cf. (Benton 2005). We will also validate the practicality of our approach in realistic code and proof / type-derivation presentation (certified code formats). For proof compilation and generation of type derivations the approach seems just ideal and we intend to implement a proof compiler / type derivation generator.

On the theoretical side, we intend to carry out a detailed comparison of our natural-semantics based

direct approach to the continuation-style approach of Tan & Appel (2005) and Benton (2005) that relies on Appel & McAllester's (2001) 'indexed model'.

Acknowledgements This work was partially supported by the Estonian Science Foundation under grant No. 5567 and by the EU FP6 IST project MOBIUS.

References

Appel, A. & McAllester, D. (2001), An indexed model of recursive types for foundational proof-carrying code, *ACM Trans. on Program. Lang. and Syst.* **23**(5), pp. 657–683.

Arbib, M. A. & Alagić, S. (1979), Proof rules for **goto**s, *Acta Inform.* **11**, pp. 139–148.

Bannwart, F. & Müller, P. (2005), A program logic for bytecode, to appear *in* 'Proc. of 1st Wksh. on Bytecode Semantics, Verification, Analysis and Transformation, BYTECODE 2005 (Edinburgh, UK, 9 Apr. 2005)', *Electr. Notes in Theor. Comput. Sci.*, Elsevier.

Barthe, G. & Rezk, T. (2005), Non-interference for a JVM-like language, *in* G. Morrisett, M. Fähndrich, eds., 'Proc. of 2005 ACM SIGPLAN Int. Wksh. on Types in Languages Design and Implementation, TLDI '05 (Long Beach, CA, Jan. 2005)', ACM Press, pp. 103–112.

Benton, N. (2004), A typed logic for stacks and jumps, draft.

Benton, N. (2005), A typed, compositional logic for a stack-based abstract machine, Tech. report MSR-TR-2005-84, Microsoft Research, Cambridge; shorter version to appear *in* K. Yi, ed., 'Proc. of 3rd Asian Symp. on Programming Languages and Systems, APLAS 2005 (Tsukuba, Nov. 2005)', *Lect. Notes in Comput. Sci.* **3780**, Springer-Verlag.

Clint, M. & Hoare, C. A. R. (1972), Program proving: Jumps and functions, *Acta Inform.* **1**, pp. 214–224.

Cook, S. A. (1978), Soundness and completeness of an axiom system for verification, *SIAM J. of Comput.* **7**, pp. 70–90.

de Bruin, A. (1981), Goto statements: Semantics and deduction systems, *Acta Inform.* **15**, pp. 385–424.

Denning, D. E. & Denning, P. J. (1977), Certification of programs for secure information flow, *Commun. of ACM* **20**, pp. 504–513.

Floyd, R. W. (1967), Assigning meanings to programs, *in* J. T. Schwartz, ed., 'Mathematical Aspects of Computer Science', *Proc. of Symp. in Appl. Math.* **19**, AMS, pp. 19–33.

Hoare, C. A. R. (1969), An axiomatic basis for computer programming, *Commun. of ACM* **12**, pp. 576–583.

Huisman, M. & Jacobs, B. (2000), Java program verification via a Hoare Logic with abrupt termination, *in* T. Maibaum, ed., 'Proc. of 3rd Int. Conf. on Fundamental Approaches to Software Engineering, FASE 2000 (Berlin, March/Apr. 2000)', *Lect. Notes in Comput. Sci.* **1783**, Springer-Verlag, pp. 284-303.

Kobayashi, N. & Kirane, K. (2002), Type-based information analysis for low-level languages, *in* 'Proc. of 3rd Asian Wksh. on Programming Lanaguages and Systems, APLAS'02 (Shanghai, Nov./Dec. 2002)', Shanghai Jiao Tong University, pp. 302–316.

Kowaltowski, T. (1977), Axiomatic approach to side effects and general jumps, *Acta Inform.* **7**, pp. 357–360.

Laud, P., Uustalu, T. & Vene, V. (2005), Type systems equivalent to dataflow analyses for imperative languages, in M. Hofmann & H.-W. Loidl, eds., 'Proc. of 3rd APPSEM II Wksh., APPSEM '05 (Frauenchiemsee, Sept. 2005), 12 pp., Ludwig-Maximilians-Univ. München.

Morrisett, J. G., Walker, D., Crary, K. & Glew, N. (1999), From system F to typed assembly language, *ACM Trans. on Program. Lang. and Syst.* **21**(3), pp. 527–568.

Morrisett, J. G., Crary, K., Glew, N., & Walker, D. (2002), Stack-based typed assembly language, *J. of Funct. Program.* **12**(1), pp. 3–88. Correction, ibid. **13**(5) (2003), pp. 957–959.

Naik, M. & Palsberg, J. (2005), *in* S. Sagiv, ed., 'Proc. of 14th European Symp. on Programming, ESOP 2005 (Edinburgh, Apr. 2005)', *Lect. Notes in Comput. Sci.* **3444**, Springer-Verlag, pp. 374–388.

Quigley, C. L. (2003), A programming logic for Java bytecode programs, *in* D. A. Basin & B. Wolff, eds., 'Proc. of 16th Int. Conf. on Theorem Proving in Higher-Order Logics, TPHOLs 2003 (Rome, Italy, 8–12 Sept. 2003)', *Lect. Notes in Comput. Sci.* **2758**, Springer-Verlag, pp. 41–54.

Saabas, A. & Uustalu, T. (2005), A compositional natural semantics and Hoare logic for low-level languages, to appear *in* P. Mosses & I. Ulidowski, eds., 'Proc. of 2nd Wksh. on Structured Operational Semantics, SOS 2005 (Lisbon, July 2005)', *Electr. Notes in Theor. Comput. Sci.*, Elsevier.

Schröder, L. & Mossakowski, T. (2003), Monad-independent Hoare logic in HASCASL, *in* M. Pezzè, ed., 'Proc. of 6th Int. Conf. on Fundamental Approaches to Software Engineering, FASE 2003 (Warsaw, Apr. 2003)', *Lect. Notes in Comput. Sci.* **2621**, Springer-Verlag, pp. 261–277.

Schröder, L. & Mossakowski, T. (2004), Generic exception handling and the Java monad, *in* C. Rattray, S. Maharaj & C. Shankland, eds., 'Proc. of 10th Int. Conf. on Algebraic Methodology and Software Technology, AMAST 2004 (Stirling, July 2004)', *Lect. Notes in Comput. Sci.* **3116**, Springer-Verlag, pp. 443-459.

Stata, R. & Abadi, M. (1999), A type system for Java bytecode subroutines, *ACM Trans. on Program. Lang. and Syst.* **21**(1), pp. 90–137.

Tan, G. & Appel, A. W. (2005), A compositional logic for control flow, manuscript.

Tan, G. (2005), A compositional logic for control flow and its application for proof-carrying code, PhD thesis, Princeton Univ.