A Compositional Natural Semantics and Hoare Logic for Low-Level Languages

Ando Saabas and Tarmo Uustalu [1,2]

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, EE-12618 Tallinn, Estonia

Abstract

The advent of proof-carrying code has generated significant interest in reasoning about low-level languages. It is widely believed that low-level languages with jumps must be difficult to reason about by being inherently non-modular. We argue that this is untrue. We take it seriously that, differently from statements of a high-level language, pieces of low-level code are multiple-entry and multiple-exit. And we define a piece of code to consist of either a single labelled instruction or a finite union of pieces of code. Thus we obtain a compositional natural semantics and a matching Hoare logic for a basic low-level language with jumps. By their simplicity and intuitiveness, these are comparable to the standard natural semantics and Hoare logic of WHILE. The Hoare logic is sound and complete wrt. the semantics and allows for compilation of proofs of the Hoare logic of WHILE.

Key words: Natural Semantics, Program Logics, Low-Level Languages, Compositional Reasoning, Certified Code

Dedicated to Enn Tyugu on the occasion of his 70th birthday

1 Introduction

Proof-carrying code (PCC) is a slogan name for the idea that it is the responsibility of the producer of software to ensure its safety or correctness. The software is shipped to the consumer together with a proof that the consumer can check. So the consumer only needs to trust a proof checker which would normally be a tiny program verifiable manually once and for all.

The popularity of PCC has generated significant interest in formalized reasoning about low-level languages as software is usually distributed in compiled form. Low-level languages are widely believed to be difficult to reason about

[1] Research activity partially supported by the Estonian Science Foundation under grant No. 5567. Travel supported by the EU FP5 IST project eVikings II.

[2] Email addresses: ando@cs.ioc.ee, tarmo@cs.ioc.ee

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

because of inherent non-modularity. The lack of modularity is attributed to low-level code being flat and to the prominent presence of completely unrestricted jumps. The bad consequence of a language being non-modular is that it cannot have a compositional semantics or logic.

In this paper, we argue that the non-modularity premise is untrue. While it is certainly correct that there is no explicit unambiguous structure to pieces of low-level code, which after all, are just flat finite sets of labelled instructions, they do have an inherent partial commutative monoidal structure given by finite unions of pieces of code with non-overlapping supports. In fact, any piece of code is either a single labelled instruction or a finite union of pieces of code with non-overlapping supports (clearly in many ways so, but nevertheless). We show that this seemingly banal structure provides a perfectly good "phrase structure" for low-level languages. Indeed, one only has to note that, differently from statements of a high-level language, pieces of low-level code are multiple-entry and multiple-exit, and then it is not hard to formulate a compositional natural semantics and Hoare logic that follow this phrase structure, for any reasonable low-level language. Moreover, low-level code is structured by finite unions naturally: compilation produces code that way and the same is more generally true about any process that generates code by combining smaller pieces of code together.

Technically, we formulate a structured version SGoto of a basic low-level (actually, intermediate) language Goto. We then develop a perfectly compositional natural semantics of SGoto that agrees with the standard non-compositional small-step operational semantics of Goto. We also develop a Hoare logic of SGoto that is sound and complete wrt. the natural semantics. Relevantly for PCC, we define a compilation function from While to SGoto that allows for compilation of proofs along with programs. We also show a "compilation" from SGoto to While. The rules of this backward direction of compilation provide additional insight about why the rules of our natural semantics and Hoare logic of SGoto are as they are.

Our ideas bear some similarity to those of the new paper by Tan and Appel [11] on a compositional logic for low-level languages. Differently from us, however, they do not introduce a compositional semantics (which for us serves as a very convenient link between the standard semantics and the logic) and their logic is continuation-style with a rather sophisticated interpretation of Hoare triples involving explicit fixpoint approximations. Our logic is direct-style.

The paper is organized as follows. In Section 2, we introduce our main low-level language of study, Goto, which is a Spartan language with general jumps, comparable to While. In Section 3, we present our conception of implicit structure in Goto code and explicate it in the syntax definition of a nearly identical language SGoto. Then we give a compositional natural semantics for SGoto, prove that this agrees with the standard operational semantics of Goto, present a compositional Hoare logic, and finally prove it sound and complete. In Section 4, we define compilation from While to SGoto, show that this preserves and reflects evaluations and derivable Hoare triples in a way that allows for "compilation of proofs", and present an example. In Section 5, we show that one can also translate from SGoto into While. Section 6 is a discussion of the related work and Section 7 concludes. For reference and to fix the notation, we review the syntax, natural semantics and Hoare logic of While in Appendix A.

The reader is assumed to be familiar with the operational and axiomatic approaches to programming language semantics on a basic level, and should appreciate the benefits of compositionality.

2 Goto, a low-level language

We start by defining a simple low-level language with jumps, which we call Goto, and its standard non-compositional small-step semantics. Goto will be (a variant of) the language for which we will develop a compositional semantics and logic in the rest of this paper.

The basic building blocks of Goto code are labels $\ell \in \mathbf{Label}$, arithmetical expressions $a \in \mathbf{AExp}$, boolean expressions $b \in \mathbf{BExp}$ and instructions $\mathit{instr} \in \mathbf{Instr}$. Labels are really natural numbers: $\mathbf{Label} =_{\mathrm{df}} \mathbb{N}$. Arithmetical expressions, boolean expressions and instructions are defined over a countable set of program variables $x \in \mathbf{Var}$ by the grammar[3]

$$\begin{aligned} n &\in \mathbb{Z} \\ a &::= n \mid x \mid a_0 + a_1 \mid \ldots \\ b &::= a_0 = a_1 \mid \ldots \mid \mathrm{tt} \mid \mathrm{ff} \mid \neg b \mid \ldots \\ \mathit{instr} &::= x := a \mid \mathsf{goto}\ \ell \mid \mathsf{ifnot}\ b\ \mathsf{goto}\ \ell \end{aligned}$$

Pairs of labels and instructions form labelled instructions: $\mathbf{LInstr} =_{\mathrm{df}} \mathbf{Label} \times \mathbf{Instr}$. A piece of code $c \in \mathbf{Code}$ is a finite set of labelled instructions: $\mathbf{Code} = \mathcal{P}_{\mathrm{fin}}(\mathbf{LInstr})$. A piece of code c is wellformed iff no label in the code labels two different instructions, i.e., iff $(\ell, \mathit{instr}) \in c$ and $(\ell, \mathit{instr}') \in c$ imply $\mathit{instr} = \mathit{instr}'$. The domain of a piece of code is defined as the set of labels appearing in that piece of code: $\mathrm{dom}(c) =_{\mathrm{df}} \{\ell \mid (\ell, \mathit{instr}) \in c\}$.

The semantics of Goto is defined in terms of states. A state is a pair of a label $\ell \in \mathbf{Label}$ and a store $\sigma \in \mathbf{Store} =_{\mathrm{df}} \mathbf{Var} \to \mathbb{Z}$, which determine the values of the program counter (pc) and program variables at a moment: $\mathbf{State} =_{\mathrm{df}} \mathbf{Label} \times \mathbf{Store}$. The semantics of arithmetical and boolean expressions is defined in the denotational style as for While, see Section A. The standard small-step operational semantics of pieces of code is given via an indexed single-step reduction relation $\twoheadrightarrow \in \mathbf{Code} \to \mathcal{P}(\mathbf{State} \times \mathbf{State})$ defined by the rules in Figure 1. The associated multi-step reduction relation $\twoheadrightarrow^*$

[3] The choice of using $\mathsf{ifnot}\ b\ \mathsf{goto}\ \ell$ instead of $\mathsf{if}\ b\ \mathsf{goto}\ \ell$ may seem unconventional, but is actually a more natural choice for us, considering the way while and if statements of While are usually compiled (see Section 4).

$$\frac{(\ell, x := a) \in c}{c \vdash (\ell, \sigma) \twoheadrightarrow (\ell + 1, \sigma[x \mapsto [\![a]\!]\sigma])} \; {:=} \qquad \frac{(\ell, \mathsf{goto}\ m) \in c}{c \vdash (\ell, \sigma) \twoheadrightarrow (m, \sigma)} \; \text{goto}$$

$$\frac{(\ell, \mathsf{ifnot}\ b\ \mathsf{goto}\ m) \in c \quad \sigma \models b}{c \vdash (\ell, \sigma) \twoheadrightarrow (\ell + 1, \sigma)} \; \text{ifngoto}^{\text{tt}} \qquad \frac{(\ell, \mathsf{ifnot}\ b\ \mathsf{goto}\ m) \in c \quad \sigma \not\models b}{c \vdash (\ell, \sigma) \twoheadrightarrow (m, \sigma)} \; \text{ifngoto}^{\text{ff}}$$

Fig. 1. Rules of standard operational semantics of Goto

is defined as its reflexive-transitive closure. The central shortcoming of this semantics is that it is entirely non-compositional: there is no phrase structure and all of the code has to be available all of the time because of the jump instructions.

Lemma 2.1 (Determinacy) *If* $c \vdash (\ell, \sigma) \twoheadrightarrow (\ell', \sigma')$ *and* $c \vdash (\ell, \sigma) \twoheadrightarrow (\ell'', \sigma'')$*, then* $(\ell', \sigma') = (\ell'', \sigma'')$*.*

Lemma 2.2 (Stuck states) $c \vdash (\ell, \sigma) \not\twoheadrightarrow$ *iff* $\ell \notin \mathrm{dom}(c)$*.*

Lemma 2.3 (Extension of the domain) *If* $c_0 \subseteq c_1$ *and* $\ell \in \mathrm{dom}(c_0)$*, then* $c_0 \vdash (\ell, \sigma) \twoheadrightarrow (\ell', \sigma')$ *iff* $c_1 \vdash (\ell, \sigma) \twoheadrightarrow (\ell', \sigma')$*.*

3 SGoto, a structured version

3.1 Syntax and natural semantics of SGoto

To define a structured version of Goto and a compositional (natural) semantics for it, we replace the flat, unstructured pieces of code of Goto with structured pieces of code $sc \in \mathbf{SCode}$ defined by the grammar

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

the idea being that a piece of code is either a single labelled instruction or a finite union of pieces of code. As before, we define the domain of a piece of code to consist of the labels of its instructions. More formally, the domain operation is defined inductively by the equations $\mathrm{dom}((\ell, \mathit{instr})) = \{\ell\}$, $\mathrm{dom}(\mathbf{0}) = \emptyset$, $\mathrm{dom}(sc_0 \oplus sc_1) = \mathrm{dom}(sc_0) \cup \mathrm{dom}(sc_1)$.

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. Note also that it is possible to understand domains and wellformedness as a small compositional type system on raw structured pieces of code.

An unstructured piece of code can of course be structured in many ways, so if we are to use a semantics or logic of SGoto to reason about a Goto piece of code, we face a choice regarding how to structure it. We can decide as we please, but in practice it is sensible to minimize the number of jumps between the subpieces of the given piece of code. In the converse direction, we have a forgetful function $U \in \mathbf{SCode} \to \mathbf{Code}$ defined inductively by

$U((\ell, \mathit{instr})) =_{\mathrm{df}} \{(\ell, \mathit{instr})\}$, $U(\mathbf{0}) =_{\mathrm{df}} \emptyset$, $U(\mathit{sc}_0 \oplus \mathit{sc}_1) =_{\mathrm{df}} U(\mathit{sc}_0) \cup U(\mathit{sc}_1)$.

Our compositional semantics for SGoto pieces of code is a natural semantics. The evaluation relation $\succ\!\!-\!\!\rightarrow \subseteq \mathbf{State} \times \mathbf{SCode} \times \mathbf{State}$ is defined by the rules in Figure 2. As usual, the evaluation relation relates a state at the moment of entry to a piece of code (an initial state) to the possible states at the corresponding possible moments of exit (final states), the idea being that an evaluation should correspond to a reduction sequence leading to a stuck state. The first four rules are self-explanatory. The side condition $m \neq \ell$ in the rules $\mathrm{goto}_{\mathrm{ns}}$ and $\mathrm{ifngoto}^{\mathrm{ff}}_{\mathrm{ns}}$ expresses that a goto or ifgoto instruction terminates only if it does not loop back to itself[4]. The rule $\oplus^0_{\mathrm{ns}}$ says that, if we want to evaluate $\mathit{sc}_0 \oplus \mathit{sc}_1$ starting in some state with the pc in the domain of sc_0, we need to evaluate sc_0 first and then evaluate the whole piece of code again, but from the new state where we got stuck with sc_0. The rule $\oplus^1_{\mathrm{ns}}$ is symmetric. The rule is $\mathrm{ood}_{\mathrm{ns}}$ is needed to cater for termination of the reduction sequence once the pc is outside of the program domain. (The rules could be simplified by removing the premises $\ell \in \mathrm{dom}(\mathit{sc}_i)$ from the rules $\oplus^i_{\mathrm{ns}}$. This, however would make the ruleset non-deterministic; the extra premise guarantees that, for any piece of code sc and state (ℓ, σ), exactly one rule applies.)

Notice that, as our semantics relates states to states and a state assigns a value to the pc, a piece of code can be entered from any label (not only from the beginning-label, assuming that the domain is a left-closed, right-open interval) and exited to any label (not only to the end-label). This may at the first sight look odd but really hides a central idea. A While statement is always single-entry, single-exit: it is entered from its beginning and exited through its end. But with low-level code, the situation is different: given the presence of jumps, it is perfectly meaningful to allow that we can enter from any label (even from a label outside the domain; in such situations, we are immediately stuck and thus finished, as the rule $\mathrm{ood}_{\mathrm{ns}}$ stipulates) and exit in principle to anywhere (it will be to those labels that we can reach but where we get stuck; such labels are always outside the domain). We only obtain compositionality because we treat pieces of code as multiple-entry, multiple-exit.

It is easy to prove that evaluation is deterministic (but partial—a piece of code may loop) and that the pc value in a final state is always outside the domain.

Lemma 3.1 (Determinacy) *If* $(\ell, \sigma) \succ\!\mathit{sc}\!\rightarrow (\ell', \sigma')$ *and* $(\ell, \sigma) \succ\!\mathit{sc}\!\rightarrow (\ell'', \sigma'')$, *then* $(\ell', \sigma') = (\ell'', \sigma'')$.

[4] Alternatively, we could state, e.g., the $\mathrm{goto}_{\mathrm{ns}}$ rule in the form

$$\frac{(m, \sigma) \succ\!(\ell, \mathsf{goto}\ m)\!\rightarrow (\ell', \sigma')}{(\ell, \sigma) \succ\!(\ell, \mathsf{goto}\ m)\!\rightarrow (\ell', \sigma')}\ \mathrm{goto}_{\mathrm{ns}}$$

which, in combination with the rule $\mathrm{ood}_{\mathrm{ns}}$, gives exactly the same evaluations. But that feels overly complicated: in the case of a single labelled instruction, loop detection is trivial.

$$\frac{}{(\ell,\sigma) \succ\!\!-(\ell, x := a)\!\rightarrow (\ell+1, \sigma[x \mapsto [\![a]\!]\sigma])}\ :=_{\mathrm{ns}}$$

$$\frac{m \neq \ell}{(\ell,\sigma) \succ\!\!-(\ell, \mathsf{goto}\ m)\!\rightarrow (m,\sigma)}\ \mathrm{goto}_{\mathrm{ns}}$$

$$\frac{\sigma \models b}{(\ell,\sigma) \succ\!\!-(\ell, \mathsf{ifnot}\ b\ \mathsf{goto}\ m)\!\rightarrow (\ell+1,\sigma)}\ \mathrm{ifngoto}^{\mathrm{tt}}_{\mathrm{ns}}$$

$$\frac{\sigma \not\models b \quad m \neq \ell}{(\ell,\sigma) \succ\!\!-(\ell, \mathsf{ifnot}\ b\ \mathsf{goto}\ m)\!\rightarrow (m,\sigma)}\ \mathrm{ifngoto}^{\mathrm{ff}}_{\mathrm{ns}}$$

$$\frac{\ell \in \mathrm{dom}(sc_0) \quad (\ell,\sigma) \succ\!\!- sc_0 \!\rightarrow (\ell',\sigma') \quad (\ell',\sigma') \succ\!\!- sc_0 \oplus sc_1 \!\rightarrow (\ell'',\sigma'')}{(\ell,\sigma) \succ\!\!- sc_0 \oplus sc_1 \!\rightarrow (\ell'',\sigma'')}\ \oplus^{0}_{\mathrm{ns}}$$

$$\frac{\ell \in \mathrm{dom}(sc_1) \quad (\ell,\sigma) \succ\!\!- sc_1 \!\rightarrow (\ell',\sigma') \quad (\ell',\sigma') \succ\!\!- sc_0 \oplus sc_1 \!\rightarrow (\ell'',\sigma'')}{(\ell,\sigma) \succ\!\!- sc_0 \oplus sc_1 \!\rightarrow (\ell'',\sigma'')}\ \oplus^{1}_{\mathrm{ns}}$$

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

Fig. 2. Natural semantics of SGoto

Lemma 3.2 (Postlabels) *If* $(\ell,\sigma) \succ\!\!- sc \!\rightarrow (\ell',\sigma')$*, then* $\ell' \notin \mathrm{dom}(sc)$.

More significantly, our semantics of SGoto agrees with the standard non-compositional operational semantics of Goto.

Theorem 3.3 (Preservation of evaluations)
If $(\ell,\sigma) \succ\!\!- sc \!\rightarrow (\ell',\sigma')$*, then* $U(sc) \vdash (\ell,\sigma) \twoheadrightarrow^{*} (\ell',\sigma') \not\twoheadrightarrow$.

Proof. By induction on the derivation of $(\ell,\sigma) \succ\!\!- sc \!\rightarrow (\ell',\sigma')$. □

Theorem 3.4 (Reflection of stuck reduction sequences)
If $U(sc) \vdash (\ell_0,\sigma_0) \twoheadrightarrow^{k} (\ell_k,\sigma_k) \not\twoheadrightarrow$*, then* $(\ell_0,\sigma_0) \succ\!\!- sc \!\rightarrow (\ell_k,\sigma_k)$.

Proof. By induction on the structure of sc and subordinate induction on k.□

It is an immediate consequence that the semantics of SGoto is neutral with respect to the structure imposed on a Goto program. We write $sc_0 \cong sc_1$ to say that two pieces of structured code are semantically equivalent, i.e., that, for any (ℓ,σ), (ℓ',σ'), $(\ell,\sigma) \succ\!\!- sc_0 \!\rightarrow (\ell',\sigma')$ iff $(\ell,\sigma) \succ\!\!- sc_1 \!\rightarrow (\ell',\sigma')$.

Theorem 3.5 (Neutrality wrt. phrase structure) *If* $U(sc_0) = U(sc_1)$*, then* $sc_0 \cong sc_1$.

From the partial commutative monoidal structure of set-theoretic finite unions ($\emptyset$, $\cup$) on unstructured pieces of code, we trivially get that our syntactic finite union operators ($\mathbf{0}$, $\oplus$) are a partial commutative monoidal structure on structured pieces of code up to semantic equivalence.

Corollary 3.6 (Partial commutative monoidal structure)

(i) $(sc_0 \oplus sc_1) \oplus sc_2 \cong sc_0 \oplus (sc_1 \oplus sc_2)$,

(ii) $\mathbf{0} \oplus sc \cong sc \cong sc \oplus \mathbf{0}$,

(iii) $sc_0 \oplus sc_1 \cong sc_1 \oplus sc_0$.

3.2 Hoare logic of SGoto

Similarly to the compositional natural semantics, we can define a compositional Hoare logic for SGoto. While the semantics relates states, where a state contains not only the values of the program variables but also that of the pc at some moment, the Hoare logic will enable us to relate assertions about states. As a state assigns a value to the pc, the assertion language will have a constant to refer to the pc value. Hence it is possible to make assertions that constrain the state to correspond to a certain label. This makes reasoning modular: one can make assertions only about the labels through which a particular piece of code is entered or exited, eliminating the need for a global context of invariants for all labels of the main piece of code.

The central syntactic unit of the logic are assertions $P \in \mathbf{Assn}$ that are formulae of an ambient logical language whose signature includes (a) constants for integers and function and predicate symbols for the standard integer-arithmetical operations and relations and (b) the program variables $x \in \mathbf{Var}$ as constants and a special constant pc for the pc. We write $\sigma \models_\alpha P$ to express that an assertion P holds in the structure on $\mathbb{Z}$ determined by (a) the standard meanings of the arithmetical constants, function and predicate symbols and (b) a state σ, under an assignment α of the variables of the logical language (parameters). A typical assertion would be something like $pc = 0 \wedge x = 1$. It holds in a state (ℓ, σ) iff the pc value ℓ is 0 and the variable value $\sigma(x)$ is 1. The writing $P \models Q$ means that $(\ell, \sigma) \models_\alpha P$ implies $(\ell, \sigma) \models_\alpha Q$ for any (ℓ, σ) and α.

The derivable judgements of the logic, called Hoare triples, are a relation $\{\} - \{\} \subseteq \mathbf{Assn} \times \mathbf{SCode} \times \mathbf{Assn}$ defined inductively by the rules presented in Figure 3. Just as the natural semantics, the Hoare logic is compositional. In particular, there is no global collecting of invariants. The extra disjunct $pc \neq \ell \wedge Q$ in the the precondition of the first three rules is required because of the semantic rule $\mathrm{ood}_{\mathrm{ns}}$. The disjunct $m = l$ is to account for the situation when a jump loops back to itself. Without these disjuncts the logic would be incomplete.

The rule for binary union can be seen as mix of the while and sequential composition rule for the While language: if, starting from either sc_0 or sc_1 in a state satisfying P, we end in a state satisfying P, then after running their union $sc_0 \oplus sc_1$ from a state satisfying P we are guaranteed to end in a state satisfying P (because we will be repeating sc_0 and sc_1 alternatingly). Furthermore, we know that we are out of the domains of sc_0 and sc_1. The rule of consequence is the same as in the Hoare rules of While, but note we use the formulation where the side premises refer to entailment, not deducibility in some (necessarily incomplete) proof system of the underlying logic.

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

$$\frac{}{\{(pc = \ell \wedge (Q[pc \mapsto m] \vee m = \ell)) \vee (pc \neq \ell \wedge Q)\}\,(\ell, \mathsf{goto}\ m)\,\{Q\}} \mathrm{goto_{hoa}}$$

$$\frac{}{\left\{\begin{array}{l} \quad (pc = \ell \wedge ((b \wedge Q[pc \mapsto \ell + 1]) \\ \qquad\qquad \vee (\neg b \wedge (Q[pc \mapsto m] \vee m = \ell)))) \\ \vee\ (pc \neq \ell \wedge Q) \end{array}\right\}\,(\ell, \mathsf{ifnot}\ b\ \mathsf{goto}\ m)\ \{\ Q\ \}} \mathrm{ifngoto_{hoa}}$$

$$\frac{}{\{P\}\,\mathbf{0}\,\{P\}} \mathbf{0}_{\mathrm{hoa}}$$

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

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

Fig. 3. Hoare rules of SGoto

The logic we have given is sound and complete. The proofs mimic the standard proofs for While.

Theorem 3.7 (Soundness) *If $\{P\}\,sc\,\{Q\}$, then, for any (ℓ_0, σ_0), (ℓ', σ') and α, $(\ell_0, \sigma_0) \models_\alpha P$ and $(\ell_0, \sigma_0) \succ\!sc\!\rightarrow (\ell', \sigma')$ imply $(\ell', \sigma') \models_\alpha Q$.*

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

To prove completeness, we have to assume that our language of assertions is *expressive*, following the completeness proof of the Hoare logic of While by Cook [5]. (It is enough to have a greatest fixpoint operator available.) We define $\mathrm{wlp}(sc, Q)$ to be some assertion P expressing the weakest liberal precondition of a piece of code sc wrt. an assertion Q, i.e., the property which a state (ℓ, σ) is defined to have under a valuation α iff, for any (ℓ', σ'), $(\ell, \sigma) \succ\!sc\!\rightarrow (\ell', \sigma')$ implies $(\ell', \sigma') \models_\alpha Q$.

Lemma 3.8 $\{\mathrm{wlp}(sc, Q)\}\,sc\,\{Q\}$.

Proof. By induction on the structure of sc. □

Theorem 3.9 (Completeness) *If, for any (ℓ_0, σ_0), (ℓ', σ') and α, $(\ell_0, \sigma_0) \models_\alpha P$ and $(\ell_0, \sigma_0) \succ\!sc\!\rightarrow (\ell', \sigma')$ imply $(\ell', \sigma') \models_\alpha Q$, then $\{P\}\,sc\,\{Q\}$.*

Proof. Assume that, for any (ℓ_0, σ_0), (ℓ', σ') and α, if $(\ell_0, \sigma_0) \models_\alpha P$ and $(\ell_0, \sigma_0) \succ\!sc\!\rightarrow (\ell', \sigma')$, then $(\ell', \sigma') \models_\alpha Q$. From this assumption it is immediate that $P \models \mathrm{wlp}(sc, Q)$. By Lemma 3.8 we already know that $\{\mathrm{wlp}(sc, Q)\}\,sc\,\{Q\}$. Hence rule $\mathrm{conseq_{hoa}}$ gives us $\{P\}\,sc\,\{Q\}$. □

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

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

$$\frac{s \;{}^{\ell+1}{\searrow}_{\ell''}\; sc}{\mathsf{while}\ b\ \mathsf{do}\ s \;{}^{\ell}{\searrow}_{\ell''+1}\; (\ell, \mathsf{ifnot}\ b\ \mathsf{goto}\ \ell'' + 1) \oplus (sc \oplus (\ell'', \mathsf{goto}\ \ell))}$$

Fig. 4. Rules of compilation from While to SGoto

4 Compilation from While to SGoto

4.1 Compilation and preservation/reflection of evaluations

We now proceed to defining a compilation function from While programs to SGoto programs and showing that it is reasonable, i.e., preserves and reflects evaluations. Furthermore, we will also show that it preserves and reflects derivable Hoare triples. This is nearly obvious because the logics of both While and SGoto are sound and complete. But more relevantly for PCC, the compilation also preserves and reflects the actual Hoare triple derivations that establish derivability, thus effectively allowing for compilation of proofs.

The compilation we use is really standard except that it produces structured code (we have chosen structures that are the most convenient for us) and, needless to say, it is compositional. It is defined by the rules in Figure 4. The compilation relation $-\searrow- \subseteq \mathbf{Label} \times \mathbf{Stm} \times \mathbf{SCode} \times \mathbf{Label}$ relates a label and a While statement to a piece of code and another label. The idea is that the domain of the compiled 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 4.1 (Totality and determinacy of compilation) *For any ℓ, s, there exist sc, ℓ' such that $s \;{}^{\ell}{\searrow}_{\ell'}\; sc$. If $s \;{}^{\ell}{\searrow}_{\ell'_0}\; sc_0$ and $s \;{}^{\ell}{\searrow}_{\ell'_1}\; sc_1$, then $sc_0 = sc_1$ and $\ell'_0 = \ell'_1$.*

Lemma 4.2 (Domain of compiled code) *If $s \;{}^{\ell}{\searrow}_{\ell'}\; sc$, then $\mathrm{dom}(sc) = [\ell, \ell') =_{\mathrm{df}} \{m \mid \ell \leq m < \ell'\}$.*

Compilation should of course not alter the meaning of a program. For our particular compilation, we have to take into account that While statements are morally single-entry, single-exit. This means that evaluation of a While statement and evaluation of the corresponding SGoto piece of code from not just anywhere but the right label (namely, the beginning-point of the domain) should give the same result. Moreover, if evaluation of the While statement terminates, the SGoto piece of code must be exited to the right label (namely, the end-point of the domain) and that must be the only label to which it can exit at all. It is quite easy to show that compilation preserves While evaluations and reflects those SGoto evaluations that start from the beginning-point of the domain of the compiled statement in exactly this sense. The proof of reflection is made easier by the fact that every SGoto evaluation has a unique derivation.

Theorem 4.3 (Preservation of evaluations)
If $s \,{}^{\ell}\!\searrow_{\ell'}\, sc$ *and* $\sigma \succ\!\!- s \!\rightarrow \sigma'$, *then* $(\ell, \sigma) \succ\!\!- sc \!\rightarrow (\ell', \sigma')$.

Proof. By induction on the derivation of $\sigma \succ\!\!- s \!\rightarrow \sigma'$. □

Theorem 4.4 (Reflection of evaluations)
If $s \,{}^{\ell}\!\searrow_{\ell'}\, sc$ *and* $(\ell, \sigma) \succ\!\!- sc \!\rightarrow (\ell'', \sigma')$, *then* $\ell' = \ell''$ *and* $\sigma \succ\!\!- s \!\rightarrow \sigma'$.

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

It is probably worth explaining the choice to use an ifnot b goto m instruction instead of the standard if b goto m instruction in SGoto. The reason behind it is the way while b do s statements are typically compiled: either to $\{(\ell, \mathsf{if}\ \neg b\ \mathsf{goto}\ \ell'' + 1)\} \cup sc \cup \{(\ell'', \mathsf{goto}\ \ell)\}$ in which case the loop guard must be negated, or to $\{(\ell, \mathsf{goto}\ \ell'')\} \cup sc \cup \{(\ell'', \mathsf{if}\ b\ \mathsf{goto}\ \ell + 1)\}$ in which case a jump is executed before the guard is first checked. Since neither of these is required when compiling to a language with an ifnot b goto m instruction, we consider it to be a more natural choice for a target language.

4.2 Preservation/reflection of derivable Hoare triples

PCC has made the concept of compiling proofs rather attractive. 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 SGoto program proofs, i.e., a proof compilation function.

Theorem 4.5 (Preservation of derivable Hoare triples)
If $s \,{}^{\ell}\!\searrow_{\ell'}\, sc$ *and* $\{P\}\, s\, \{Q\}$, *then* $\{pc = \ell \wedge P\}\, sc\, \{pc = \ell' \wedge Q\}$.

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 SGoto. □

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

Reflection of derivable SGoto 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 SGoto program) into a correctness proof of the While program. For the constructive proof, we have to use the fact that proofs of SGoto programs admit a certain normal form.

Theorem 4.6 (Reflection of derivable Hoare triples) *If* $s\,{}^{\ell}\!\searrow_{\ell'}\,sc$ *and* $\{P\}\,sc\,\{Q\}$*, then* $\{P[pc \mapsto \ell]\}\,s\,\{Q[pc \mapsto \ell']\}$.

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

Proof. [Constructive proof: Preservation 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: a proper inference is always followed by a consequence inference, which in turn is followed by a proper inference unless its conclusion is the end judgement of the derivation etc. 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. □

4.3 Example

As a simple example of compilation we present a While factorial program together with its proof, and then the target SGoto program with its proof. The factorial program in While is $S =_{\mathrm{df}} \mathsf{while}\ x < n\ \mathsf{do}\ (x := x{+}1; s := s{*}x)$. For this program, we have the following Hoare triple proof (we refrain here from explicitly spelling out the side conditions of consequence inferences, these are obvious from the context).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\begin{array}{c} x+1 \le n \\ \wedge\ s*(x+1) = (x+1)! \end{array}\}\,x := x+1\,\{\begin{array}{c} x \le n \\ \wedge\ s*x = x! \end{array}\}}}{\{x < n \wedge s = x!\}\,x := x+1\,\{x \le n \wedge s*x = x!\}} \quad \dfrac{}{\{x \le n \wedge s*x = x!\}\,s := s*x\,\{s = x! \wedge x \le n\}}}{\{x < n \wedge s = x!\}\,x := x+1; s := s*x\,\{x \le n \wedge s = x!\}}}{\{x < n \wedge x \le n \wedge s = x!\}\,x := x+1; s := s*x\,\{x \le n \wedge s = x!\}}}{\{x \le n \wedge s = x!\}\,S\,\{x \not< n \wedge x \le n \wedge s = x!\}}}{\{n \ge 0 \wedge x = 0 \wedge s = 1\}\,S\,\{x = n \wedge s = n!\}}$$

The compilation function gives us the SGoto program $C =_{\mathrm{df}} (1, \mathsf{ifnot}\ x < n\ \mathsf{goto}\ 5) \oplus (((2, x := x+1) \oplus (3, s := s*x)) \oplus (4, \mathsf{goto}\ 1))$. To present its proof we introduce the notations

$$
\begin{aligned}
I_1 &=_{\mathrm{df}} pc = 1 \wedge n \le 0 \wedge x = 0 \wedge s = 1 \\
I_{1'} &=_{\mathrm{df}} pc = 1 \wedge x \le n \wedge s = x! \\
I_2 &=_{\mathrm{df}} pc = 2 \wedge x < n \wedge x \le n \wedge s = x! \\
I_{2'} &=_{\mathrm{df}} pc = 2 \wedge x < n \wedge s = x! \\
I_{2''} &=_{\mathrm{df}} pc = 2 \wedge x + 1 \le n \wedge s * (x+1) = (x+1)!
\end{aligned}
$$

$$
\begin{aligned}
I_3 &=_{\mathrm{df}} pc = 3 \wedge x \leq n \wedge s * x = x! \\
I_4 &=_{\mathrm{df}} pc = 4 \wedge x \leq n \wedge s = x! \\
I_5 &=_{\mathrm{df}} pc = 5 \wedge x \not< n \wedge x \leq n \wedge s = x! \\
I_{5'} &=_{\mathrm{df}} pc = 5 \wedge x = n \wedge s = x!
\end{aligned}
$$

We will use the shorthand notation $I_{i \ldots j}$ to denote the disjunction $I_i \vee \ldots \vee I_j$.

The proof for the SGoto program is the following (the Hoare triples corresponding to those in the While version are highlighted).

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\{J_{1'}\}\,1\,\{I_{25}\}}}{\{pc = 1 \wedge I_{1'25}\}\,1\,\{I_{1'25}\}} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{J_{2''}\}\,2\,\{I_3\}}}{\{I_{2''}\}\,2\,\{I_3\}}}{\{I_{2'}\}\,2\,\{I_3\}}}{\{pc = 2 \wedge I_{2'34}\}\,2\,\{I_{2'34}\}} \quad \dfrac{\dfrac{\dfrac{}{\{J_3\}\,3\,\{I_4\}}}{\{I_3\}\,3\,\{I_4\}}}{\{pc = 3 \wedge I_{2'34}\}\,3\,\{I_{2'34}\}}}{\{I_{2'34}\}\,2 \oplus 3\,\{pc \notin [2,4) \wedge I_{2'34}\}}}{\{I'_2\}\,2 \oplus 3\,\{I_4\}}}{\{I_2\}\,2 \oplus 3\,\{I_4\}}}{\{pc \in [2,4) \wedge I_{1'24}\}\,2 \oplus 3\,\{I_{1'24}\}} \quad \dfrac{\dfrac{}{\{J_4\}\,4\,\{I_{1'}\}}}{\{pc = 4 \wedge I_{1'24}\}\,4\,\{I_{1'24}\}}}{\{I_{1'24}\}\,(2 \oplus 3) \oplus 4\,\{pc \notin [2,5) \wedge I_{1'24}\}}}{\{pc \in [2,5) \wedge I_{1'25}\}\,(2 \oplus 3) \oplus 4\,\{I_{1'25}\}}}{\{I_{1'25}\}\,C\,\{pc \notin [1,5) \wedge I_{1'25}\}}}{\{I_{1'}\}\,C\,\{I_5\}}}{\{I_1\}\,C\,\{I_{5'}\}}
$$

Here

$$
\begin{aligned}
J_{1'} &=_{\mathrm{df}} (pc = 1 \wedge ((x < n \wedge I_{25}[pc \mapsto 2]) \vee (x \not< n \wedge (I_{25}[pc \mapsto 5] \vee 5 = 1))) \\
&\qquad\quad \vee (pc \neq 1 \wedge I_{25}) \\
J_{2''} &=_{\mathrm{df}} (pc = 2 \wedge I_3[(pc, x) \mapsto (2, x + 1)]) \vee (pc \neq 2 \wedge I_3) \\
J_3 &=_{\mathrm{df}} (pc = 3 \wedge I_4[(pc, s) \mapsto (3, s * x)]) \vee (pc \neq 3 \wedge I_4) \\
J_4 &=_{\mathrm{df}} (pc = 4 \wedge (I_{1'}[pc \mapsto 4] \vee 1 = 4)) \vee (pc \neq 4 \wedge I_{1'})
\end{aligned}
$$

The example should explain the general idea of modularity of our Hoare logic: we do not need global information for a judgement, but only the invariants for the entry and exit labels of the code at hand. While not so obvious in the small case presented here, it becomes more effective the larger the code piece is. As code is composed, we can eliminate the invariants of the intermediate entries and exits that are not required any more, so at the root of the tree, we only have the entry and exit invariants of the whole program, i.e., I_1 and I'_5.

5 Compilation from SGoto to While

It is also possible to define a translation of SGoto pieces of code to While statements. This is not decompilation, which would only be possible for those SGoto pieces of code that are in the image of the compilation function of the previous section. Rather, the idea is to define a While statement that would simulate the evaluation of a given SGoto piece of code. This means we have to allocate a designated variable $x_{pc} \in \mathbf{Var}$ to store the value of the pc. Compilation from SGoto to While is defined inductively by the rules

$$\frac{}{(\ell, x := a) \nearrow \mathsf{if}\ x_{pc} = \ell\ \mathsf{then}\ x := a; x_{pc} := x_{pc} + 1\ \mathsf{else\ skip}}$$

$$\frac{}{(\ell, \mathsf{goto}\ m) \nearrow \mathsf{while}\ x_{pc} = \ell\ \mathsf{do}\ x_{pc} := m}$$

$$\frac{}{(\ell, \mathsf{ifnot}\ b\ \mathsf{goto}\ m) \nearrow \mathsf{while}\ x_{pc} = \ell\ \mathsf{do}\ (\mathsf{if}\ b\ \mathsf{then}\ x_{pc} := \ell + 1\ \mathsf{else}\ x_{pc} := m)}$$

$$\frac{}{\mathbf{0} \nearrow \mathsf{skip}} \qquad \frac{sc_0 \nearrow s_0 \quad sc_1 \nearrow s_1}{sc_0 \oplus sc_1 \nearrow \begin{array}{c} \mathsf{while}\ x_{pc} \in \mathrm{dom}(sc_0) \lor x_{pc} \in \mathrm{dom}(sc_1)\ \mathsf{do} \\ (\mathsf{if}\ x_{pc} \in \mathrm{dom}(sc_0)\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1) \end{array}}$$

Fig. 5. Rules of compilation from SGoto to While

in Figure 5.

Lemma 5.1 (Totality and determinacy of compilation) *For any sc, there exists s such that $sc \nearrow s$. If $sc \nearrow s_0$ and $sc \nearrow s_1$, then $s_0 = s_1$.*

Thinking closer about the rules of this translation, we see that, in a sense, the natural semantics rules of SGoto are "derivable" from those of While (this would be even more direct, if our rule for goto were $(\ell, \mathsf{goto}\ m) \nearrow \mathsf{if}\ x_{pc} = \ell\ \mathsf{then}\ (\mathsf{if}\ m \neq \ell\ \mathsf{then}\ x_{pc} := m\ \mathsf{else\ diverge})\ \mathsf{else\ skip}$, but we have no primitive diverge construct in While, so the rule in the figure is the shortest). This makes it very easy to prove that compilation preserves and reflects evaluations here as well.

Theorem 5.2 (Preservation and reflection of evaluations) *If $sc \nearrow s$ and $(\ell, \sigma) \succ sc \rightarrow (\ell', \sigma')$, then $\sigma[x_{pc} \mapsto \ell] \succ s \rightarrow \sigma'[x_{pc} \mapsto \ell']$. If $sc \nearrow s$ and $\sigma \succ s \rightarrow \sigma'$, then $(\sigma(x_{pc}), \sigma[x_{pc} \mapsto n]) \succ sc \rightarrow (\sigma'(x_{pc}), \sigma'[x_{pc} \mapsto n])$.*

A similar observation can be made about the Hoare rules of SGoto. With the translation from SGoto to While in mind, one can read them out from the Hoare rules of While. And similarly to the case of compilation from While to SGoto, we also have it here that derivable Hoare triples are preserved and reflected. This can again be proved in two ways: non-constructively and constructively.

Theorem 5.3 (Preservation and reflection of derivable Hoare triples) *If $sc \nearrow s$ and $\{P\}\, sc\, \{Q\}$, then $\{P[pc \mapsto x_{pc}]\}\, s\, \{Q[pc \mapsto x_{pc}]\}$. If $sc \nearrow s$ and $\{P\}\, s\, \{Q\}$, then $\{P[x_{pc} \mapsto pc]\}\, sc\, \{Q[x_{pc} \mapsto pc]\}$.*

6 Related work

In the young days of Hoare logic, quite some attention was paid to reasoning about general and restricted jumps. The first Hoare logic was formulated for While [7] and characteristic to the various proposals that were made thereafter [4,8,1,6,9] is that they deal with While or a similar structured high-level language extended with general or restricted jumps. The logics of Clint and Hoare, Kowaltowski and de Bruin [4,8,6] 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 and Alagić [1], Hoare triples have multiple postconditions, reflecting the fact that statements involving gotos are multiple-exit.

Reasoning about unstructured low-level languages has become a topic of active research only with the advent of the idea of PCC. Typical languages of interest are (subsets of) Java bytecode or .NET CIL. The logic of Quigley [10] is based on decompilation, so it applies to pieces of code in the image of a certain compiler. In Benton's logic [3], there are global label invariants as in the logic of de Bruin [6]. Bannwart and Müller's logic [2] extends Benton's logic to an object-oriented language.

Our basic idea to utilize the implicit finite unions structure of low-level languages in combination with appreciating that pieces of code are not only multiple-exit but also multiple-entry appear in the new work of Tan and Appel [11] [5]. Differently from us, their logic is continuation-style and, because of the way they have chosen to formulate their rules for binary union, they must use "approximations of falsity". A state (ℓ, σ) is k-safe for a piece of code sc iff there is no $j < k$ and (ℓ', σ') such that $U(sc) \vdash (\ell, \sigma) \twoheadrightarrow^j (\ell', \sigma') \not\twoheadrightarrow$. A state (ℓ, σ) k-falsifies P iff, for any (ℓ', σ'), $U(sc) \vdash (\ell, \sigma) \twoheadrightarrow^* (\ell', \sigma')$ and $(\ell', \sigma') \models P$ imply that (ℓ', σ') is k-safe. A Hoare triple $\{P\}\, sc\, \{Q\}$ is defined to be valid iff, for any state (ℓ, σ), if (ℓ, σ) k-falsifies Q, then (ℓ, σ) $(k + 1)$-falsifies P.

7 Conclusions and future work

We have demonstrated that the obvious but seemingly uninteresting structure on pieces of code given by finite unions is really all that a low-level language needs in order to admit a compositional natural semantics and Hoare logic with every desirable metatheoretic property. Moreover, the semantic and logic descriptions thus achieved are no more complicated than the standard ones of standard high-level languages, which we find remarkable. Our work is related to that of Tan and Appel [11], but they did not introduce a natural semantics and our logic is simpler than theirs by avoiding continuations and more conventional by interpreting Hoare triples in the standard way.

Our work is clearly relevant for PCC, first because it deals with low-level languages and second because finite unions are a natural construction in realistic situations where a larger piece of code would very typically arise as a sum of smaller pieces that are separately produced, often by different producers, and should then also be proved correct separately. A small concern with our approach from the PCC point of view might be that proofs of our logic pertain to structured pieces of code, so if a producer is to supply a consumer a piece of low-level code with a proof, she must also reveal the structure she used, which makes it possible for the consumer to recover the original high-level

[5] Confusingly, what we call 'labelled instructions' and 'pieces of code' are called 'fragments of code' and 'sets of fragments of code' in that work.

program and its proof (if the producer uses a simple non-optimizing compiler and if the consumer knows the compilation rules). But this does not matter really. Much more valuably, the consumer retains the benefit of not having to compile himself and trust a compiler for this. We consider all of this to be of secondary importance for the present work, since our focus here has been on semantic descriptions anyway.

It remains to validate the practicality of our approach in realistic code and proof presentation (certified code formats). For proof compilation, the approach seems just ideal.

References

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

[2] Bannwart, F. and P. Müller, *A program logic for bytecode*, 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., to appear.

[3] Benton, N., *A typed logic for stacks and jumps*, draft (2004).

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

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

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

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

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

[9] O'Donnell, M. J., *A critique of the foundations of Hoare style programming logics*, Commun. of ACM **25** (1982), pp. 927–935.

[10] Quigley, C. L., *A programming logic for Java bytecode programs*, in: D. A. Basin and B. Wolff, editors, *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, Berlin (2003), pp. 41–54.

[11] Tan, G. and A. W. Appel, *A compositional logic for control flow*, manuscript (2005).

A The high-level language While

This section is a summary of the syntax, natural semantics and the standard Hoare logic of the basic high-level language While [7].

A.1 Syntax

The syntax proceeds from a countable supply of arithmetic variables $x \in \mathbf{Var}$. Over these, three syntactic categories of arithmetic expressions $a \in \mathbf{AExp}$, boolean expressions $b \in \mathbf{BExp}$ and statements $s \in \mathbf{Stm}$ are defined by means of the grammar

$$\begin{aligned}
a &::= x \mid n \mid a_0 + a_1 \mid \ldots \\
b &::= a_0 = a_1 \mid \ldots \mid \mathrm{tt} \mid \mathrm{ff} \mid \neg b \mid \ldots \\
s &::= x := a \mid \mathsf{skip} \mid s_0; s_1 \mid \mathsf{if}\ b\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \mid \mathsf{while}\ b\ \mathsf{do}\ s
\end{aligned}$$

A.2 Natural semantics

The semantics is given in terms of states. The states are defined as stores $\sigma \in \mathbf{Store}$, i.e., mappings of variables to integers: $\mathbf{State} =_{\mathrm{df}} \mathbf{Store} =_{\mathrm{df}} \mathbf{Var} \to \mathbb{Z}$. The arithmetical and boolean expressions are interpreted relative to stores as integers and truth values by the semantic function $[\![-]\!] \in \mathbf{AExp} + \mathbf{BExp} \to \mathbf{Store} \to \mathbb{Z}$, defined in the denotational style by the equations

$$\begin{aligned}
[\![x]\!]\sigma &=_{\mathrm{df}} \sigma(x) \\
[\![n]\!]\sigma &=_{\mathrm{df}} n \\
[\![a_0 + a_1]\!]\sigma &=_{\mathrm{df}} [\![a_0]\!]\sigma + [\![a_1]\!]\sigma \\
\ldots &=_{\mathrm{df}} \ldots \\
[\![a_0 = a_1]\!]\sigma &=_{\mathrm{df}} [\![a_0]\!]\sigma = [\![a_1]\!]\sigma \\
\ldots &=_{\mathrm{df}} \ldots \\
[\![\mathrm{tt}]\!]\sigma &=_{\mathrm{df}} \mathrm{tt} \\
[\![\mathrm{ff}]\!]\sigma &=_{\mathrm{df}} \mathrm{ff} \\
[\![\neg b]\!]\sigma &=_{\mathrm{df}} \neg [\![b]\!]\sigma \\
\ldots &=_{\mathrm{df}} \ldots
\end{aligned}$$

We write $\sigma \models b$ to say that $[\![b]\!]\sigma = \mathrm{tt}$.

Statements are interpreted via the evaluation relation $\succ - \rightarrow\ \subseteq \mathbf{State} \times \mathbf{Stm} \times \mathbf{State}$ defined inductively by the ruleset given in Figure A.1.

Lemma A.1 (Determinacy) *If $\sigma \succ s \rightarrow \sigma'$ and $\sigma \succ s \rightarrow \sigma''$, then $\sigma' = \sigma''$.*

A.3 Hoare logic

The assertions $P \in \mathbf{Assn}$ are defined as formulae of an ambient logical language whose signature includes (a) constants for integers and function and predicate symbols for the standard integer-arithmetical operations and relations and (b) the program variables $x \in \mathbf{Var}$ as constants. For the complete-

$$\frac{}{\sigma \succ x := a \to \sigma[x \mapsto [\![a]\!]\sigma]}\ :=_{\mathrm{ns}}$$

$$\frac{}{\sigma \succ \mathsf{skip} \to \sigma}\ \mathrm{skip}_{\mathrm{ns}} \qquad \frac{\sigma \succ s_0 \to \sigma'' \quad \sigma'' \succ s_1 \to \sigma'}{\sigma \succ s_0; s_1 \to \sigma'}\ \mathrm{comp}_{\mathrm{ns}}$$

$$\frac{\sigma \models b \quad \sigma \succ s_t \to \sigma'}{\sigma \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \to \sigma'}\ \mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}} \qquad \frac{\sigma \not\models b \quad \sigma \succ s_f \to \sigma'}{\sigma \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \to \sigma'}\ \mathrm{if}^{\mathrm{ff}}_{\mathrm{ns}}$$

$$\frac{\sigma \models b \quad \sigma \succ s \to \sigma'' \quad \sigma'' \succ \mathsf{while}\ b\ \mathsf{do}\ s \to \sigma'}{\sigma \succ \mathsf{while}\ b\ \mathsf{do}\ s \to \sigma'}\ \mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}} \qquad \frac{\sigma \not\models b}{\sigma \succ \mathsf{while}\ b\ \mathsf{do}\ s \to \sigma}\ \mathrm{while}^{\mathrm{ff}}_{\mathrm{ns}}$$

Fig. A.1. Natural semantics rules of While

$$\frac{}{\{Q[x \mapsto a]\}\ x := a\ \{Q\}}\ :=_{\mathrm{hoa}}$$

$$\frac{}{\{P\}\ \mathsf{skip}\ \{P\}}\ \mathrm{skip}_{\mathrm{hoa}} \qquad \frac{\{P\}\ s_0\ \{R\} \quad \{R\}\ s_1\ \{Q\}}{\{P\}\ s_0; s_1\ \{Q\}}\ \mathrm{comp}_{\mathrm{hoa}}$$

$$\frac{\{b \wedge P\}\ s_t\ \{Q\} \quad \{\neg b \wedge P\}\ s_f\ \{Q\}}{\{P\}\ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\ \{Q\}}\ \mathrm{if}_{\mathrm{hoa}} \qquad \frac{\{b \wedge P\}\ s\ \{P\}}{\{P\}\ \mathsf{while}\ b\ \mathsf{do}\ s\ \{\neg b \wedge P\}}\ \mathrm{while}_{\mathrm{hoa}}$$

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

Fig. A.2. Hoare rules of While

ness result, the language is assumed to be expressive enough so as to allow the expression of the weakest liberal precondition of any statement wrt. any given postcondition [5]. We write $\sigma \models_\alpha P$ to express that P holds in the structure on $\mathbb{Z}$ determined by (a) the standard meanings of the arithmetical constants, function and predicate symbols and (b) a state σ, under an assignment α of the logical variables. The writing $P \models Q$ means that $\sigma \models_\alpha P$ implies $\sigma \models_\alpha Q$ for any σ, α.

The derivable judgements of the logic are given by the relation $\{\} - \{\} \subseteq$ **Assn** × **Stm** × **Assn** defined inductively by the ruleset in Figure A.2. (Note that, in the consequence rule, the side premises rely on entailment, not deducibility in some proof system of the underlying logic.)

Theorem A.2 (Soundness) *If $\{P\}\ s\ \{Q\}$, then, for any σ, σ' and α, $\sigma \models_\alpha P$ and $\sigma \succ s \to \sigma'$ imply $\sigma' \models_\alpha Q$.*

Theorem A.3 (Completeness) *If, for any σ, σ' and α, $\sigma \models_\alpha P$ and $\sigma \succ s \to \sigma'$ imply $\sigma' \models_\alpha Q$, then $\{P\}\ s\ \{Q\}$.*