

A Compositional Natural Semantics and Hoare Logic for Low-Level Languages¹

Ando Saabas² and Tarmo Uustalu^{*,2}

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

Received 6 March 2006; revised 6 September 2006; accepted 6 October 2006

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: Low-Level Languages, Natural Semantics, Hoare Logics, Compositionality, Compilation of Proofs

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

* Corresponding author.

Email addresses: ando@cs.ioc.ee (Ando Saabas), tarmo@cs.ioc.ee (Tarmo Uustalu).

¹ This paper is a significantly revised and extended version of Ref. [15].

² Research activity partially supported by the Estonian Science Foundation under grant No. 5567 and EITSA Tiigriülikool program. Travel supported by the EU FP5 IST project eVikings II.

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 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, one can closely follow the designs for high-level languages. 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 already generated code together.

Technically, we formulate a structured version `SGOTO` of a basic low-level language `GOTO` with expressions and assignment. We then develop a perfectly compositional natural semantics of `SGOTO` that agrees with the standard non-compositional small-step operational semantics of the unstructured `GOTO` language. We also develop a Hoare logic of `SGOTO` that is sound and complete with respect to the natural semantics. Both follow closely the natural semantics and Hoare logic of the textbook high-level language `WHILE`. 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 backward “compilation” function 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 work by Tan and Appel [18,19] on a compositional logic for low-level languages. Differently from

us, however, they do not introduce any compositional semantics (for us, our compositional natural semantics 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 motivated by the small-step semantics of the unstructured language and 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. 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 small-step 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. Some proofs omitted from the main text are presented in Appendix B.

The reader is assumed to be familiar with the operational and axiomatic approaches to programming language semantics on a basic level, and expected to 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 small-step semantics. **GOTO** will be the language for a structured version of 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 $instr \in \mathbf{Instr}$. Labels are really natural numbers: $\mathbf{Label} =_{\text{df}} \mathbb{N}$. Arithmetical expressions, boolean expressions and instructions are defined over a countable set of program variables $x \in \mathbf{Var}$ by the grammar³

$$n \in \mathbb{Z}$$

³ We have chosen to take `ifnot b goto ℓ` rather than `if b goto ℓ` as primitive, considering the way while and if statements of **WHILE** are usually compiled (see Section 4).

$$\begin{array}{c}
\frac{(\ell, x := a) \in c}{c \vdash (\ell, \sigma) \rightarrow (\ell + 1, \sigma[x \mapsto \llbracket a \rrbracket \sigma])} := \\
\frac{(\ell, \text{goto } m) \in c}{c \vdash (\ell, \sigma) \rightarrow (m, \sigma)} \text{goto} \\
\frac{(\ell, \text{ifnot } b \text{ goto } m) \in c \quad \sigma \models b}{c \vdash (\ell, \sigma) \rightarrow (\ell + 1, \sigma)} \text{ifngoto}^{\text{tt}} \\
\frac{(\ell, \text{ifnot } b \text{ goto } m) \in c \quad \sigma \not\models b}{c \vdash (\ell, \sigma) \rightarrow (m, \sigma)} \text{ifngoto}^{\text{ff}}
\end{array}$$

Fig. 1. Small-step semantics rules of GOTO

$$\begin{array}{l}
a ::= n \mid x \mid a_0 + a_1 \mid \dots \\
b ::= a_0 = a_1 \mid \dots \mid \text{tt} \mid \text{ff} \mid \neg b \mid \dots \\
\text{instr} ::= x := a \mid \text{goto } \ell \mid \text{ifnot } b \text{ goto } \ell
\end{array}$$

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

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 $\sigma \in \mathbf{Store} =_{\text{df}} \mathbf{Var} \rightarrow \mathbb{Z}$, which determine the values of the program counter (pc) and program variables at a moment: $\mathbf{State} =_{\text{df}} \mathbf{Label} \times \mathbf{Store}$. 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 $\rightarrow \in \mathbf{Code} \rightarrow \mathcal{P}(\mathbf{State} \times \mathbf{State})$ defined by the rules in Figure 1. (The notation $\sigma[x \mapsto z]$ denotes the state obtained from σ by updating the value of x to z . The notation $\sigma \models b$ is shorthand for $\llbracket b \rrbracket \sigma = \text{tt}$.) The associated multi-step reduction relation \rightarrow^* 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 1 (Determinacy) *If $c \vdash (\ell, \sigma) \rightarrow (\ell', \sigma')$ and $c \vdash (\ell, \sigma) \rightarrow (\ell'', \sigma'')$, then $(\ell', \sigma') = (\ell'', \sigma'')$.*

Lemma 2 (Stuck states) $c \vdash (\ell, \sigma) \not\rightarrow$ iff $\ell \notin \text{dom}(c)$.

Lemma 3 (Extension of the domain) *If $c_0 \subseteq c_1$ and $\ell \in \text{dom}(c_0)$, then $c_0 \vdash (\ell, \sigma) \rightarrow (\ell', \sigma')$ iff $c_1 \vdash (\ell, \sigma) \rightarrow (\ell', \sigma')$.*

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

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

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 $\text{dom}((\ell, instr)) = \{\ell\}$, $\text{dom}(\mathbf{0}) = \emptyset$, $\text{dom}(sc_0 \oplus sc_1) = \text{dom}(sc_0) \cup \text{dom}(sc_1)$.

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 $\text{dom}(sc_0) \cap \text{dom}(sc_1) = \emptyset$. 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 $U \in \mathbf{SCode} \rightarrow \mathbf{Code}$ defined inductively by $U((\ell, instr)) =_{\text{df}} \{(\ell, instr)\}$, $U(\mathbf{0}) =_{\text{df}} \emptyset$, $U(sc_0 \oplus sc_1) =_{\text{df}} U(sc_0) \cup U(sc_1)$.

Our compositional semantics for SGOTO pieces of code is a natural semantics, modelled after that for the textbook high-level language WHILE (see Appendix A). 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 goto_{ns} and $\text{ifngoto}_{\text{ns}}^{\text{ff}}$ expresses that a goto or ifngoto instruction terminates

$$\begin{array}{c}
\frac{}{(\ell, \sigma) \succ (\ell, x := a) \rightarrow (\ell + 1, \sigma[x \mapsto \llbracket a \rrbracket \sigma])} :=_{\text{ns}} \\
\frac{m \neq \ell}{(\ell, \sigma) \succ (\ell, \text{goto } m) \rightarrow (m, \sigma)} \text{goto}_{\text{ns}} \\
\frac{\sigma \models b}{(\ell, \sigma) \succ (\ell, \text{ifnot } b \text{ goto } m) \rightarrow (\ell + 1, \sigma)} \text{ifngoto}_{\text{ns}}^{\text{tt}} \\
\frac{\sigma \not\models b \quad m \neq \ell}{(\ell, \sigma) \succ (\ell, \text{ifnot } b \text{ goto } m) \rightarrow (m, \sigma)} \text{ifngoto}_{\text{ns}}^{\text{ff}} \\
\frac{\ell \in \text{dom}(sc_0) \quad (\ell, \sigma) \succ sc_0 \rightarrow (\ell'', \sigma'') \quad (\ell'', \sigma'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', \sigma')}{(\ell, \sigma) \succ sc_0 \oplus sc_1 \rightarrow (\ell', \sigma')} \oplus_{\text{ns}}^0 \\
\frac{\ell \in \text{dom}(sc_1) \quad (\ell, \sigma) \succ sc_1 \rightarrow (\ell'', \sigma'') \quad (\ell'', \sigma'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', \sigma')}{(\ell, \sigma) \succ sc_0 \oplus sc_1 \rightarrow (\ell', \sigma')} \oplus_{\text{ns}}^1 \\
\frac{\ell \notin \text{dom}(sc)}{(\ell, \sigma) \succ sc \rightarrow (\ell, \sigma)} \text{ood}_{\text{ns}}
\end{array}$$

Fig. 2. Natural semantics rules of SGOTO

only if it does not loop back to itself⁴. The rule \oplus_{ns}^0 says that, if we want to evaluate $sc_0 \oplus sc_1$ 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_{ns}^1 is symmetric. The rule ood_{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 $\ell \in \text{dom}(sc_i)$ from the rules \oplus_{ns}^i . 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 a piece of code to be

⁴ Alternatively, we could state, e.g., the goto_{ns} rule in the form

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

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

entered from any label (including any label outside the domain: at such labels we are immediately finished). Also, one can exit a piece of code to several labels: any jump target outside the domain is a potential final label, as is any label outside the domain that immediately succeeds the label of a non-jump instruction. 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 4 (Determinacy) *If $(\ell, \sigma) \succ_{sc} \rightarrow (\ell', \sigma')$ and $(\ell, \sigma) \succ_{sc} \rightarrow (\ell'', \sigma'')$, then $(\ell', \sigma') = (\ell'', \sigma'')$.*

Lemma 5 (Postlabels) *If $(\ell, \sigma) \succ_{sc} \rightarrow (\ell', \sigma')$, then $\ell' \notin \text{dom}(sc)$.*

More significantly, our semantics of SGOTO agrees with the standard non-compositional operational semantics of GOTO.

Theorem 6 (Preservation of evaluations as stuck reduction sequences) *If $(\ell, \sigma) \succ_{sc} \rightarrow (\ell', \sigma')$, then $U(sc) \vdash (\ell, \sigma) \rightarrow^* (\ell', \sigma') \not\rightsquigarrow$.*

PROOF. By structural induction on the derivation of $(\ell, \sigma) \succ_{sc} \rightarrow (\ell', \sigma')$. The proof appears in Appendix B.

Theorem 7 (Reflection of stuck reduction sequences as evaluations) *If $U(sc) \vdash (\ell_0, \sigma_0) \rightarrow^k (\ell_k, \sigma_k) \not\rightsquigarrow$, then $(\ell_0, \sigma_0) \succ_{sc} \rightarrow (\ell_k, \sigma_k)$.*

PROOF. By structural induction on sc . The proof appears in Appendix B.

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 8 (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 9 (Partial commutative monoidal structure)

- (1) $(sc_0 \oplus sc_1) \oplus sc_2 \cong sc_0 \oplus (sc_1 \oplus sc_2)$,
- (2) $\mathbf{0} \oplus sc \cong sc \cong sc \oplus \mathbf{0}$,
- (3) $sc_0 \oplus sc_1 \cong sc_1 \oplus sc_0$.

Clearly, instead of empty and binary unions of pieces of code, one can base the syntax of **SGOTO** on unions of any finite size, i.e., instead of forms of pieces of code $\mathbf{0}$ and $sc_0 \oplus sc_1$, use the form $\sum_{i=0}^{n-1} sc_i$ ($n \geq 0$). In this version of the language, it is possible to write completely flat code (no structure at all), by representing a piece $c = \{(\ell_i, instr_i) \mid 0 \leq i < n\}$ of **GOTO** code as $\sum_{i=0}^{n-1} (\ell_i, instr_i)$ (the union of all single instructions in c).

3.2 Hoare logic of **SGOTO**

Similarly to the compositional natural semantics, we can define a compositional Hoare logic for **SGOTO**. Again the design for **WHILE** can be taken as a model. Just as with the semantics, which related 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 write assertions that constrain the state to correspond to a certain label.

The central syntactic unit of the logic are assertions $P \in \mathbf{Assn}$. These are formulae of an unspecified underlying logic (typically first-order predicate logic with equality) over a signature consisting of (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 $(\ell, \sigma) \models_{\alpha} P$ 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 and a pc value ℓ , 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, α . We use the notation $Q[x_0, \dots, x_n \mapsto a_0, \dots, a_n]$ to denote that every free occurrence of x_i in Q has been replaced with a_i for all $i, 0 \leq i \leq n$.

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. This is a partial correctness logic, so the idea is that $\{P\} sc \{Q\}$ should hold iff $(\ell_0, \sigma_0) \models_{\alpha} P$ and $(\ell_0, \sigma_0) \succ_{sc} (\ell', \sigma')$ always imply $(\ell', \sigma') \models_{\alpha} Q$. Just as the natural semantics, the Hoare logic is compositional, with the resulting modularity. In particular, there is no need to work with one flat

$$\begin{array}{c}
\frac{}{\overline{\{(pc = \ell \wedge Q[(pc, x) \mapsto (\ell + 1, a)]) \vee (pc \neq \ell \wedge Q)\} (\ell, x := a) \{Q\}}} :=_{\text{hoa}} \\
\frac{}{\overline{\{(pc = \ell \wedge (Q[pc \mapsto m] \vee m = \ell)) \vee (pc \neq \ell \wedge Q)\} (\ell, \text{goto } m) \{Q\}}} \text{goto}_{\text{hoa}} \\
\frac{}{\overline{\left\{ \begin{array}{l} (pc = \ell \wedge ((b \wedge Q[pc \mapsto \ell + 1]) \\ \vee (\neg b \wedge (Q[pc \mapsto m] \vee m = \ell)))) \\ \vee (pc \neq \ell \wedge Q) \end{array} \right\}} (\ell, \text{ifnot } b \text{ goto } m) \{Q\}}} \text{ifngoto}_{\text{hoa}} \\
\frac{}{\overline{\{P\} \mathbf{0} \{P\}}} \mathbf{0}_{\text{hoa}} \\
\frac{\overline{\{pc \in \text{dom}(sc_0) \wedge P\} sc_0 \{P\}} \quad \overline{\{pc \in \text{dom}(sc_1) \wedge P\} sc_1 \{P\}}}{\overline{\{P\} sc_0 \oplus sc_1 \{pc \notin \text{dom}(sc_0) \wedge pc \notin \text{dom}(sc_1) \wedge P\}}} \oplus_{\text{hoa}} \\
\frac{P \models P' \quad \overline{\{P'\} sc \{Q'\}} \quad Q' \models Q}{\overline{\{P\} sc \{Q\}}} \text{conseq}_{\text{hoa}}
\end{array}$$

Fig. 3. Hoare rules of SGOTO

invariant for the whole top-level piece of code, containing assertions for all labels in the domain and all potential final labels.

The extra disjunct $pc \neq \ell \wedge Q$ in the the precondition of the first three rules is required because of the semantic rule ood_{ns} . The disjunct $m = \ell$ 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 sequence rules for the WHILE language: the assertion P serves as an invariant. If running either sc_0 or sc_1 from a state which is in the domain and satisfies 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 alternately). Furthermore, we know that we are outside of the domains of both sc_0 and sc_1 .

The rule of consequence is standard. To circumvent the inevitable incompleteness of axiomatizations of logical theories containing arithmetic, we use the version where the side conditions invoke semantic entailment rather than deducibility in some proof system for the underlying logical theory. With deducibility instead of entailment one can only obtain *relative* completeness of the Hoare logic.

It might seem very restrictive that, unlike in WHILE, where an invariant is required only for loops, in the case of low-level code an invariant is required for every single union. It is true that proofs will be more verbose because of this, but technically, building a derivation is no more complicated than in Hoare logic for WHILE. As an invariant of a union one can use a set of assertions for those pc values through which its two subpieces can be entered and exited,

$$\begin{array}{c}
\frac{\frac{\overline{\{I_2\} (2, x := y) \{I_3\}}}{\{pc = 2 \wedge I_{234}\} (2, x := y) \{I_{234}\}} \quad \frac{\overline{\{I_3\} (2, y := t) \{I_4\}}}{\{pc = 3 \wedge I_{234}\} (3, y := t) \{I_{234}\}}}{\frac{\{I_{234}\} (2, x := y) \oplus (3, y := t) \{pc \notin \{2, 3\} \wedge I_{234}\}}{\{pc \in \{2, 3\} \wedge I_{124}\} (2, x := y) \oplus (3, y := t) \{I_{124}\}}} \\
\frac{\overline{\{I_1\} (1, t := x) \{I_2\}}}{\{pc = 1 \wedge I_{124}\} (1, t := x) \{I_{124}\}} \quad / \\
\frac{\overline{\{I_{124}\} (1, t := x) \oplus ((2, x := y) \oplus (3, y := t)) \{pc \notin \{1, 2, 3\} \wedge I_{124}\}}}{\{I_1\} (1, t := x) \oplus ((2, x := y) \oplus (3, y := t)) \{I_4\}}
\end{array}$$

Fig. 4. Example of a Hoare triple derivation

i.e., an assertion of the form $(pc = \ell_0 \wedge P_0) \vee \dots \vee (pc = \ell_n \wedge P_n)$. The assertion for a particular label is expected to hold whenever this label is passed through. For a union of two contiguous and adjacent pieces of code with no jumps out of either of the two, this means three labels, provided the union is meant to be entered only from its beginning: the beginning of the union, the mid-point and the end. There is no need assert anything about any other labels.

As a simple example of straight-line code, consider $sc =_{\text{df}} (1, t := x) \oplus ((2, x := y) \oplus (3, y := t))$. This swaps the values of two variables x and y via an intermediate variable t . A correct postcondition for $I_1 =_{\text{df}} pc = 1 \wedge x = x_0 \wedge y = y_0$ as a precondition should be $I_4 =_{\text{df}} pc = 4 \wedge y = x_0 \wedge x = y_0$.

Let $I_2 =_{\text{df}} pc = 2 \wedge t = x_0 \wedge y = y_0$, $I_3 =_{\text{df}} pc = 3 \wedge t = x_0 \wedge x = y_0$, $I_{124} =_{\text{df}} I_1 \vee I_2 \vee I_4$ and $I_{234} =_{\text{df}} I_2 \vee I_3 \vee I_4$. The derivation of the Hoare triple $\{I_1\} sc \{I_4\}$ is given in Figure 4 (the side conditions of inferences by the consequence rule have been left implicit). Notice that in the invariant I_{124} for the whole code we do not need an assertion for label 3, which is internal to the domain of the second summand.

The logic we have given is sound and complete (completeness holds, if the logic is expressive, see below). The proofs mimic the standard proofs for WHILE. Soundness is the straightforward part.

Theorem 10 (Soundness) *If $\{P\} sc \{Q\}$, then, for any $\ell_0, \sigma_0, \ell', \sigma'$ and α , $(\ell_0, \sigma_0) \models_\alpha P$ and $(\ell_0, \sigma_0) \succ\text{-}sc \rightarrow (\ell', \sigma')$ imply $(\ell', \sigma') \models_\alpha Q$.*

PROOF. Assume $\{P\} sc \{Q\}$. We use structural induction on the derivation of $\{P\} sc \{Q\}$. We have the following cases.

- The derivation of $\{P\} \text{sc} \{Q\}$ is

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

Suppose $(\ell_0, \sigma_0) \models_{\alpha} (pc = \ell \wedge Q[(pc, x) \mapsto (\ell + 1, a)]) \vee (pc \neq \ell \wedge Q)$ and $(\ell_0, \sigma_0) \succ (\ell, x := a) \rightarrow (\ell', \sigma')$ for some $\ell_0, \sigma_0, \ell', \sigma'$ and α .

If $\ell_0 = \ell$, then $(\ell_0, \sigma_0) \models_{\alpha} Q[(pc, x) \mapsto (\ell + 1, a)]$. By Lemma 4 and rule :=_{ns} it must be that $(\ell', \sigma') = (\ell + 1, \sigma_0[x \mapsto \llbracket a \rrbracket \sigma_0])$. Hence $(\ell', \sigma') \models_{\alpha} Q$.

If $\ell_0 \neq \ell$, then $(\ell_0, \sigma_0) \models_{\alpha} Q$. By Lemma 4 and rule ood_{ns} , we get $(\ell', \sigma') = (\ell_0, \sigma_0)$, from where it is trivial that $(\ell', \sigma') \models_{\alpha} Q$.

- The derivation of $\{P\} \text{sc} \{Q\}$ is

$$\overline{\{(pc = \ell \wedge (Q[pc \mapsto m] \vee m = \ell)) \vee (pc \neq \ell \wedge Q)\} (\ell, \text{goto } m) \{Q\}} \text{goto}_{\text{hoa}}$$

Suppose $(\ell_0, \sigma_0) \models_{\alpha} (pc = \ell \wedge (Q[pc \mapsto m] \vee m = \ell)) \vee (pc \neq \ell \wedge Q)$ and $(\ell_0, \sigma_0) \succ (\ell, \text{goto } m) \rightarrow (\ell', \sigma')$ for some $\ell_0, \sigma_0, \ell', \sigma'$ and α .

If $\ell_0 = \ell$, then $(\ell_0, \sigma_0) \models_{\alpha} Q[pc \mapsto m] \vee m = \ell$. By Lemma 4 and rule goto_{ns} , we have $m \neq \ell$ and $(\ell', \sigma') = (m, \sigma_0)$. Hence $(\ell', \sigma') \models_{\alpha} Q$.

If $\ell_0 \neq \ell$, then $(\ell_0, \sigma_0) \models_{\alpha} Q$. By Lemma 4 and rule ood_{ns} , we get $(\ell', \sigma') = (\ell_0, \sigma_0)$, which trivially gives $(\ell', \sigma') \models_{\alpha} Q$.

- The derivation is

$$\overline{\left\{ \begin{array}{l} (pc = \ell \wedge ((b \wedge Q[pc \mapsto \ell + 1]) \\ \vee (\neg b \wedge (Q[pc \mapsto m] \vee m = \ell)))) \\ \vee (pc \neq \ell \wedge Q) \end{array} \right\} (\ell, \text{ifnot } b \text{ goto } m) \{Q\}} \text{ifngoto}_{\text{hoa}}$$

Suppose $(\ell_0, \sigma_0) \models_{\alpha} (pc = \ell \wedge ((b \wedge Q[pc \mapsto \ell + 1]) \vee (\neg b \wedge (Q[pc \mapsto m] \vee m = \ell)))) \vee (pc \neq \ell \wedge Q)$ and $(\ell_0, \sigma_0) \succ (\ell, \text{ifnot } b \text{ goto } m) \rightarrow (\ell', \sigma')$ for some $\ell_0, \sigma_0, \ell', \sigma'$ and α .

If $\ell_0 = \ell$ and $\sigma_0 \models b$, then $(\ell_0, \sigma_0) \models_{\alpha} Q[pc \mapsto \ell + 1]$. By Lemma 4 and rule $\text{ifngoto}_{\text{ns}}^{\text{tt}}$, we have $(\ell', \sigma') = (\ell + 1, \sigma_0)$. Hence $(\ell', \sigma') \models_{\alpha} Q$.

If $\ell_0 = \ell$ and $\sigma_0 \not\models b$, then $(\ell_0, \sigma_0) \models_{\alpha} Q[pc \mapsto m] \vee m = \ell$. By Lemma 4 and rule $\text{ifngoto}_{\text{ns}}^{\text{ff}}$, we have $m \neq \ell$ and $(\ell', \sigma') = (m, \sigma_0)$. Hence $(\ell', \sigma') \models_{\alpha} Q$.

If $\ell_0 \neq \ell$, then $(\ell_0, \sigma_0) \models_{\alpha} Q$. By Lemma 4 and rule ood_{ns} , we get $(\ell', \sigma') = (\ell_0, \sigma_0)$, which trivially gives $(\ell', \sigma') \models_{\alpha} Q$.

- The derivation of $\{P\} \text{sc} \{Q\}$ is

$$\overline{\{P\} \mathbf{0} \{P\}} \mathbf{0}_{\text{hoa}}$$

Suppose $(\ell_0, \sigma_0) \models_{\alpha} Q$ and $(\ell_0, \sigma_0) \succ \mathbf{0} \rightarrow (\ell', \sigma')$ for some $\ell_0, \sigma_0, \ell', \sigma'$ and α . By Lemma 4 and rule ood_{ns} , we get $(\ell', \sigma') = (\ell_0, \sigma_0)$, which trivially gives $(\ell', \sigma') \models_{\alpha} Q$.

- The derivation of $\{P\} sc \{Q\}$ is

$$\frac{\begin{array}{c} \vdots \\ \{pc \in \text{dom}(sc_0) \wedge P\} sc_0 \{P\} \end{array} \quad \begin{array}{c} \vdots \\ \{pc \in \text{dom}(sc_1) \wedge P\} sc_1 \{P\} \end{array}}{\{P\} sc_0 \oplus sc_1 \{pc \notin \text{dom}(sc_0) \wedge pc \notin \text{dom}(sc_1) \wedge P\}} \oplus_{\text{hoa}}$$

Suppose $(\ell_0, \sigma_0) \models_{\alpha} P$ and $(\ell_0, \sigma_0) \succ_{sc_0 \oplus sc_1} (\ell', \sigma')$ for some $\ell_0, \sigma_0, \ell', \sigma'$ and α . We invoke structural induction on the derivation of $(\ell_0, \sigma_0) \succ_{sc_0 \oplus sc_1} (\ell', \sigma')$.

If $\ell_0 \in \text{dom}(sc_i)$ ($i = 0$ or 1), then by Lemma 4 the last inference of the derivation of $(\ell_0, \sigma_0) \succ_{sc_0 \oplus sc_1} (\ell', \sigma')$ must be an application of rule \oplus_{ns}^i to $(\ell_0, \sigma_0) \succ_{sc_i} (\ell'', \sigma'')$ and $(\ell'', \sigma'') \succ_{sc_0 \oplus sc_1} (\ell', \sigma')$ for some ℓ'', σ'' . We have $(\ell_0, \sigma_0) \models_{\alpha} pc \in \text{dom}(sc_i) \wedge P$, from where by the outer induction hypothesis $(\ell'', \sigma'') \models_{\alpha} P$ and further by the inner induction hypothesis $(\ell', \sigma') \models_{\alpha} pc \notin \text{dom}(sc_0) \wedge pc \notin \text{dom}(sc_1) \wedge P$.

If $\ell_0 \notin \text{dom}(sc_0)$ and $\ell_0 \notin \text{dom}(sc_1)$, then by Lemma 4 and rule ood_{ns} , we get $(\ell', \sigma') = (\ell_0, \sigma_0)$, from where it follows that $(\ell', \sigma') \models_{\alpha} pc \notin \text{dom}(sc_0) \wedge pc \notin \text{dom}(sc_1) \wedge P$.

- The derivation of $\{P\} sc \{Q\}$ is

$$\frac{P \models P' \quad \begin{array}{c} \vdots \\ \{P'\} sc \{Q'\} \end{array} \quad Q' \models Q}{\{P\} sc \{Q\}} \text{conseq}_{\text{hoa}}$$

Suppose $(\ell_0, \sigma_0) \models P$ and $(\ell_0, \sigma_0) \succ_{sc} (\ell', \sigma')$ for some $\ell_0, \sigma_0, \ell', \sigma'$ and α . As $P \models P'$, we get $(\ell_0, \sigma_0) \models_{\alpha} P'$. By the induction hypothesis, therefore $(\ell', \sigma') \models_{\alpha} Q'$. From $Q' \models Q$, this gives $(\ell', \sigma') \models_{\alpha} Q$. \square

To prove completeness, we have to assume that the language of the underlying logic is *expressive*, following the completeness proof of the Hoare logic of WHILE by Cook [7]. Specifically, for any assertion Q and piece of code sc , we need an assertion $\text{wlp}(sc, Q)$ that, semantically, is the weakest precondition guaranteeing Q , in the sense that, for any ℓ, σ and α , we have $(\ell, \sigma) \models_{\alpha} \text{wlp}(sc, Q)$ iff, for any ℓ', σ' , $(\ell, \sigma) \succ_{sc} (\ell', \sigma')$ implies $(\ell', \sigma') \models_{\alpha} Q$. The wlp function is easily definable by structural induction on sc , if the logic has a greatest fixedpoint operator (which happens, e.g., in 1st-order logic extended with primitive least/greatest fixedpoint operators and in 2nd-order logic, where such operators are definable).

The following lemma states that the semantic weakest precondition is always a precondition in the sense of the proof system. Once that is established, completeness is a straightforward corollary.

Lemma 11 $\{\text{wlp}(sc, Q)\} sc \{Q\}$.

PROOF. We use structural induction on sc . There are the following cases.

- $sc = (\ell, x := a)$: By rule $:=_{\text{hoa}}$, we have

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

We also have

$$\text{wlp}((\ell, x := a), Q) \models (pc = \ell \wedge Q[(pc, x) \mapsto (\ell + 1, a)]) \vee (pc \neq \ell \wedge Q)$$

Indeed, suppose $(\ell_0, \sigma_0) \models_{\alpha} \text{wlp}((\ell, x := a), Q)$ for some ℓ_0, σ_0 and α . If $\ell_0 = \ell$, then by rule $:=_{\text{ns}}$ we have $(\ell_0, \sigma_0) \succ (\ell, x := a) \rightarrow (\ell + 1, \sigma_0[x \mapsto \llbracket a \rrbracket \sigma_0])$, so $(\ell + 1, \sigma_0[x \mapsto \llbracket a \rrbracket \sigma_0]) \models_{\alpha} Q$, from where $(\ell_0, \sigma_0) \models_{\alpha} pc = \ell \wedge Q[(pc, x) \mapsto (\ell + 1, a)]$. If $\ell_0 \neq \ell$, then by rule ood_{ns} we have $(\ell_0, \sigma_0) \succ (\ell, x := a) \rightarrow (\ell_0, \sigma_0)$, so $(\ell_0, \sigma_0) \models_{\alpha} Q$, from where $(\ell_0, \sigma_0) \models_{\alpha} pc \neq \ell \wedge Q$.

By rule $\text{conseq}_{\text{hoa}}$, we get

$$\{\text{wlp}((\ell, x := a), Q)\} (\ell, x := a) \{Q\}$$

- $sc = (\ell, \text{goto } m)$: By rule goto_{hoa} , we have

$$\{(pc = \ell \wedge (Q[pc \mapsto m] \vee m = \ell)) \vee (pc \neq \ell \wedge Q)\} (\ell, \text{goto } m) \{Q\}$$

We also have

$$\text{wlp}((\ell, \text{goto } m), Q) \models (pc = \ell \wedge (Q[pc \mapsto m] \vee m = \ell)) \vee (pc \neq \ell \wedge Q)$$

Indeed, suppose $(\ell_0, \sigma_0) \models_{\alpha} \text{wlp}((\ell, \text{goto } m), Q)$ for some ℓ_0, σ_0 and α . If $\ell_0 = \ell$ and $m \neq \ell$, then by rule goto_{ns} we have $(\ell_0, \sigma_0) \succ (\ell, \text{goto } m) \rightarrow (m, \sigma_0)$, so $(m, \sigma_0) \models_{\alpha} Q$, from where $(\ell_0, \sigma_0) \models_{\alpha} pc = \ell \wedge Q[pc \mapsto m]$. If $\ell_0 = \ell$ and $m = \ell$, then $(\ell_0, \sigma_0) \models_{\alpha} pc = \ell \wedge m = \ell$. If $\ell_0 \neq \ell$, then by rule ood_{ns} we have $(\ell_0, \sigma_0) \succ (\ell, \text{goto } m) \rightarrow (\ell_0, \sigma_0)$, so $(\ell_0, \sigma_0) \models_{\alpha} Q$, from where $(\ell_0, \sigma_0) \models_{\alpha} pc \neq \ell \wedge Q$.

By rule $\text{conseq}_{\text{hoa}}$, we get

$$\{\text{wlp}((\ell, \text{goto } m), Q)\} (\ell, \text{goto } m) \{Q\}$$

- $sc = (\ell, \text{ifnot } b \text{ goto } m)$: By rule $\text{ifngoto}_{\text{hoa}}$, we have

$$\left\{ \begin{array}{l} (pc = \ell \wedge ((b \wedge Q[pc \mapsto \ell + 1]) \\ \vee (\neg b \wedge (Q[pc \mapsto m] \vee m = \ell)))) \\ \vee (pc \neq \ell \wedge Q) \end{array} \right\} (\ell, \text{ifnot } b \text{ goto } m) \{Q\}$$

We also have

$$\begin{aligned} \text{wlp}((\ell, \text{ifnot } b \text{ goto } m), Q) \models \\ (pc = \ell \wedge ((b \wedge Q[pc \mapsto \ell + 1]) \vee (\neg b \wedge (Q[pc \mapsto m] \vee m = \ell)))) \\ \vee (pc \neq \ell \wedge Q) \end{aligned}$$

Indeed, suppose $(\ell_0, \sigma_0) \models_\alpha \text{wlp}((\ell, \text{ifnot } b \text{ goto } m), Q)$ for some ℓ_0, σ_0 and α . If $\ell_0 = \ell$ and $\sigma_0 \models b$, then by rule $\text{ifngoto}_{\text{ns}}^{\text{tt}}$ we have $(\ell_0, \sigma_0) \succ (\ell, \text{ifnot } b \text{ goto } m) \rightarrow (\ell + 1, \sigma_0)$, so $(\ell + 1, \sigma_0) \models_\alpha Q$, from where $(\ell_0, \sigma_0) \models_\alpha pc = \ell \wedge b \wedge Q[pc \mapsto \ell + 1]$. If $\ell_0 = \ell$ and $\sigma_0 \not\models b$ and $m \neq \ell$, then by rule $\text{ifngoto}_{\text{ns}}^{\text{ff}}$ we have $(\ell_0, \sigma_0) \succ (\ell, \text{ifnot } b \text{ goto } m) \rightarrow (m, \sigma_0)$, so $(m, \sigma_0) \models_\alpha Q$, from where $(\ell_0, \sigma_0) \models_\alpha pc = \ell \wedge \neg b \wedge Q[pc \mapsto m]$. If $\ell_0 = \ell$ and $\sigma_0 \not\models b$ and $m = \ell$, then $(\ell_0, \sigma_0) \models_\alpha pc = \ell \wedge \neg b \wedge m = \ell$. If $\ell_0 \neq \ell$, then by rule ood_{ns} we have $(\ell_0, \sigma_0) \succ (\ell, \text{ifnot } b \text{ goto } m) \rightarrow (\ell_0, \sigma_0)$, so $(\ell_0, \sigma_0) \models_\alpha Q$, from where $(\ell_0, \sigma_0) \models_\alpha pc \neq \ell \wedge Q$.

By rule $\text{conseq}_{\text{hoa}}$, we get

$$\{\text{wlp}((\ell, \text{ifnot } b \text{ goto } m), Q)\} (\ell, \text{ifnot } b \text{ goto } m) \{Q\}$$

- $sc = \mathbf{0}$: By rule $\mathbf{0}_{\text{hoa}}$, we have

$$\{\text{wlp}(\mathbf{0}, Q)\} \mathbf{0} \{\text{wlp}(\mathbf{0}, Q)\}$$

We also have

$$\text{wlp}(\mathbf{0}, Q) \models Q$$

Indeed, suppose $(\ell_0, \sigma_0) \models_\alpha \text{wlp}(\mathbf{0}, Q)$ for some ℓ_0, σ_0 and α . Rule ood_{ns} gives us $(\ell_0, \sigma_0) \succ \mathbf{0} \rightarrow (\ell_0, \sigma_0)$ and hence $(\ell_0, \sigma_0) \models_\alpha Q$.

Hence by rule $\text{conseq}_{\text{hoa}}$, we get

$$\{\text{wlp}(\mathbf{0}, Q)\} \mathbf{0} \{Q\}$$

- $sc = sc_0 \oplus sc_1$: By the induction hypothesis, we have

$$\{\text{wlp}(sc_i, \text{wlp}(sc_0 \oplus sc_1, Q))\} sc_i \{\text{wlp}(sc_0 \oplus sc_1, Q)\}$$

(for $i = 0$ and 1). We also have

$$pc \in \text{dom}(sc_i) \wedge \text{wlp}(sc_0 \oplus sc_1, Q) \models \text{wlp}(sc_i, \text{wlp}(sc_0 \oplus sc_1, Q))$$

(for $i = 0$ and 1). Indeed, suppose $(\ell_0, \sigma_0) \models_\alpha pc \in \text{dom}(sc_i) \wedge \text{wlp}(sc_0 \oplus sc_1, Q)$ for some ℓ_0, σ_0 and α . Then $\ell_0 \in \text{dom}(sc_i)$ and $(\ell_0, \sigma_0) \models_\alpha \text{wlp}(sc_0 \oplus sc_1, Q)$. Consider any $\ell'', \sigma'', \ell', \sigma'$ such that $(\ell_0, \sigma_0) \succ sc_i \rightarrow (\ell'', \sigma'')$ and $(\ell'', \sigma'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', \sigma')$. By rule \oplus_{ns}^i we have $(\ell_0, \sigma_0) \succ sc_0 \oplus sc_1 \rightarrow (\ell', \sigma')$, which gives us $(\ell', \sigma') \models_\alpha Q$. Hence, $(\ell_0, \sigma_0) \models_\alpha \text{wlp}(sc_i, \text{wlp}(sc_0 \oplus sc_1, Q))$ as needed.

Rule $\text{conseq}_{\text{hoa}}$ gives us

$$\{pc \in \text{dom}(sc_i) \wedge \text{wlp}(sc_0 \oplus sc_1, Q)\} sc_i \{\text{wlp}(sc_0 \oplus sc_1, Q)\}$$

(for $i = 0$ and 1). From here, rule \oplus_{hoa} gives us

$$\{\text{wlp}(sc_0 \oplus sc_1, Q)\} sc_0 \oplus sc_1 \left\{ \begin{array}{l} pc \notin \text{dom}(sc_0) \wedge pc \notin \text{dom}(sc_1) \\ \wedge \text{wlp}(sc_0 \oplus sc_1, Q) \end{array} \right\}$$

Further, we also have

$$pc \notin \text{dom}(sc_0) \wedge pc \notin \text{dom}(sc_1) \wedge \text{wlp}(sc_0 \oplus sc_1, Q) \models Q$$

Indeed, suppose $(\ell_0, \sigma_0) \models pc \notin \text{dom}(sc_0) \wedge pc \notin \text{dom}(sc_1) \wedge \text{wlp}(sc_0 \oplus sc_1, Q)$ for some ℓ_0, σ_0 and α . We then have $\ell_0 \notin \text{dom}(sc_0 \oplus sc_1)$ and $(\ell_0, \sigma_0) \models_\alpha \text{wlp}(sc_0 \oplus sc_1, Q)$. Rule ood_{ns} gives us $(\ell_0, \sigma_0) \succ_{sc_0 \oplus sc_1} (\ell_0, \sigma_0)$ and hence $(\ell_0, \sigma_0) \models_\alpha Q$.

Hence by rule $\text{conseq}_{\text{hoa}}$, we get

$$\{\text{wlp}(sc_0 \oplus sc_1, Q)\} sc_0 \oplus sc_1 \{Q\}$$

□

Theorem 12 (Completeness) *If, for any $\ell_0, \sigma_0, \ell', \sigma'$ and α , $(\ell_0, \sigma_0) \models_\alpha P$ and $(\ell_0, \sigma_0) \succ_{sc} (\ell', \sigma')$ imply $(\ell', \sigma') \models_\alpha Q$, then $\{P\} sc \{Q\}$.*

PROOF. Assume that, for any $\ell_0, \sigma_0, \ell', \sigma', \alpha$, if $(\ell_0, \sigma_0) \models_\alpha P$ and $(\ell_0, \sigma_0) \succ_{sc} (\ell', \sigma')$, then $(\ell', \sigma') \models_\alpha Q$. From this assumption it is immediate that $P \models \text{wlp}(sc, Q)$. By Lemma 11 we already know that $\{\text{wlp}(sc, Q)\} sc \{Q\}$. Hence rule $\text{conseq}_{\text{hoa}}$ gives us $\{P\} sc \{Q\}$. □

We chose that assertions are formulae in a signature containing the constant pc . Equivalently, we could exclude this constant from the signature and define an assertion to be a set of labelled formulae with the wellformedness condition that no label labels two different formulae. An assertion $\{(\ell_i, P_i) \mid 0 \leq i < n\}$ (with P_i 's not containing pc) is to be interpreted as $\bigvee_{i \in [0, n)} (pc = \ell_i \wedge P_i)$. And conversely, an assertion P (containing pc) translates to $\{(\ell, P[pc \mapsto \ell]) \mid \ell \in \mathbf{Label}\}$ (where one can drop those labelled formulae that are contradictory), based on the obvious logical equivalence $P \Leftrightarrow \bigvee_{\ell \in \mathbf{Label}} (pc = \ell \wedge P[pc \mapsto \ell])$ (a generalized Shannon expansion formula).

The modified notion of assertion leads to slightly different Hoare rules, e.g., the rule $:=_{\text{hoa}}$ takes the form

$$\frac{\{(\ell, Q[x \mapsto a]) \mid (\ell + 1, Q) \in \Pi\} \cup \Pi \upharpoonright_{\bar{\ell}}}{\{(\ell, x := a)\} \Pi} :='_{\text{hoa}}$$

and the rule \oplus_{hoa} becomes

$$\frac{\{\Pi \upharpoonright_{\text{dom}(sc_0)}\} sc_0 \{ \Pi \} \quad \{\Pi \upharpoonright_{\text{dom}(sc_1)}\} sc_1 \{ \Pi \}}{\{\Pi\} sc_0 \oplus sc_1 \{\Pi \upharpoonright_{\text{dom}(sc_0) \cup \text{dom}(sc_1)}\}} \oplus'_{\text{hoa}}$$

where $\Pi \upharpoonright_L =_{\text{df}} \{(\ell, P) \mid (\ell, P) \in \Pi, \ell \in L\}$ and $\bar{L} =_{\text{df}} \mathbf{Label} \setminus L$.

Which format is better is a matter of taste. Single formulae involving pc give some economy in the metatheory, but in actual applications sets of labelled formulae can be more handy.

We finish this section by remarking that, for the purpose of simplicity, we did not include in `GOTO` and `SGOTO` indirect jumps (also known as embedded code pointers). But all of our development applies just as well to the obvious extensions where jump targets can be any arithmetic expressions, not just numerals. The modifications needed are trivial. In the instruction syntax, we replace the instruction form `goto ℓ` with `goto a` (and similarly for `ifngoto`). The natural semantics rule for `goto` becomes

$$\frac{\llbracket a \rrbracket \sigma \neq \ell}{(\ell, \sigma) \succ (\ell, \text{goto } a) \rightarrow (\llbracket a \rrbracket \sigma, \sigma)} \text{goto}_{\text{ns}}$$

and the Hoare rule for `goto` becomes

$$\frac{\{(pc = \ell \wedge (Q[pc \mapsto a] \vee a = \ell)) \vee (pc \neq \ell \wedge Q)\} (\ell, \text{goto } a) \{Q\}}{\text{goto}_{\text{hoa}}}$$

This is quite pleasing in the view of the fact that indirect jumps are generally believed to be problematic for reasoning, not to mention compositional reasoning. It is appropriate to point out that here the syntax with single formulae involving pc is superior to sets of (constantly) labelled formulae.

We finish this section by remarking that, in principle, it is also possible to formulate a total correctness logic. Again, one can use the design for `WHILE` as an example. The rule for unions must use a variant (an expression decreasing in the sense of some wellfounded relation), similarly to the rule for while-loops. In practice, coming up with variants for unions is even more hard than finding them for while-loops and they will typically involve the special constant pc . Luckily, however, for compiled code, Hoare triple derivations are mechanically constructible from Hoare triple derivations for original programs. For our partial correctness logic, we show this in the next section.

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

$$\begin{array}{c}
\frac{}{x := a \searrow_{\ell+1} (\ell, x := a)} \quad \frac{}{\text{skip} \searrow_{\ell} \mathbf{0}} \quad \frac{s_0 \searrow_{\ell'} sc_0 \quad s_1 \searrow_{\ell'} sc_1}{s_0; s_1 \searrow_{\ell'} sc_0 \oplus sc_1} \\
\frac{s_t \searrow_{\ell'} sc_t \quad s_f \searrow_{\ell'} sc_f}{\text{if } b \text{ then } s_t \text{ else } s_f \searrow_{\ell'} (\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \oplus ((sc_t \oplus (\ell'', \text{goto } \ell')) \oplus sc_f)} \\
\frac{s_t \searrow_{\ell'} sc_t}{\text{while } b \text{ do } s_t \searrow_{\ell'+1} (\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \oplus (sc_t \oplus (\ell'', \text{goto } \ell))}
\end{array}$$

Fig. 5. Rules of compilation from WHILE to SGOTO

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 WHILE language we use here is standard. For reference, its syntax, natural semantics and Hoare logic appear in Appendix A.

The compilation relation we use is 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 5. 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, i.e., an interval specified by a beginning-point and end-point, including the beginning and excluding the end. (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 13 (Totality and determinacy of compilation) *For any ℓ, s , there exist sc, ℓ' such that $s \searrow_{\ell'} sc$. If $s \searrow_{\ell'_0} sc_0$ and $s \searrow_{\ell'_1} sc_1$, then $sc_0 = sc_1$ and $\ell'_0 = \ell'_1$.*

Lemma 14 (Domain of compiled code) *If $s \searrow_{\ell'} sc$, then $\ell \leq \ell'$ and $\text{dom}(sc) = [\ell, \ell') =_{\text{df}} \{m \mid \ell \leq m < \ell'\}$.*

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 15 (Preservation of evaluations) *If $s \stackrel{\ell}{\searrow}_{\ell'} sc$ and $\sigma \succ_s \rightarrow \sigma'$, then $(\ell, \sigma) \succ_{sc} \rightarrow (\ell', \sigma')$.*

PROOF. By structural induction on the derivation of $\sigma \succ_s \rightarrow \sigma'$. The proof appears in Appendix B.

Theorem 16 (Reflection of evaluations) *If $s \stackrel{\ell}{\searrow}_{\ell'} sc$ and $(\ell, \sigma) \succ_{sc} \rightarrow (\ell^*, \sigma')$, then (a) $\ell^* = \ell'$ and (b) $\sigma \succ_s \rightarrow \sigma'$.*

PROOF. By structural induction on s . The proof appears in Appendix B.

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 $\{(\ell, \text{if } \neg b \text{ goto } \ell'' + 1)\} \cup sc \cup \{(\ell'', \text{goto } \ell)\}$ in which case the loop guard must be negated, or to $\{(\ell, \text{goto } \ell'')\} \cup sc \cup \{(\ell'', \text{if } b \text{ goto } \ell + 1)\}$ 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 17 (Preservation of derivable Hoare triples) *If $s \stackrel{\ell}{\searrow}_{\ell'} sc$ and $\{P\} s \{Q\}$, then $\{pc = \ell \wedge P\} sc \{pc = \ell' \wedge Q\}$.*

PROOF. [Non-constructive proof] Assume $s \stackrel{\ell}{\rightsquigarrow}_{\ell'} sc$ and $\{P\} s \{Q\}$.

By soundness of the logic of the high-level language, $\sigma \models P$ and $\sigma \succ_{s \rightarrow} \sigma'$ imply $\sigma' \models Q$, for any σ, σ' .

Hence by reflection of transitions by compilation, $\sigma \models P$ and $(\ell, \sigma) \succ_{sc \rightarrow} (m', \sigma')$ imply $m' = \ell'$ and $\sigma' \models Q$, for any σ, m', σ' .

From here, $(m, \sigma) \models pc = \ell \wedge P$ and $(m, \sigma) \succ_{sc \rightarrow} (m', \sigma')$ imply $(m', \sigma') \models pc = \ell' \wedge Q$, for any m, σ, m', σ' . Indeed, suppose $(m, \sigma) \models pc = \ell \wedge P$ and $(m, \sigma) \succ_{sc \rightarrow} (m', \sigma')$ for some m, σ, m', σ' . Then $m = \ell, \sigma \models P$ and $(\ell, \sigma) \succ_{sc \rightarrow} (m', \sigma')$, which we know yield $m' = \ell'$ and $\sigma' \models Q$, thus $(m', \sigma') \models pc = \ell' \wedge Q$, as required.

Hence, by completeness of the logic of the low-level language, $\{pc = \ell \wedge P\} sc \{pc = \ell' \wedge Q\}$. \square

PROOF. [Constructive proof: Preservation of Hoare triple derivations] By structural induction on the derivation of $\{P\} s \{Q\}$. The proof is in Appendix B.

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. In this form, proper inferences come in strict alternation with consequence inferences: starting from any leaf, 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 and so on. (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.)

Theorem 18 (Reflection of derivable Hoare triples) *If $s \stackrel{\ell}{\rightsquigarrow}_{\ell'} sc$ and $\{P\} sc \{Q\}$, then $\{P[pc \mapsto \ell]\} s \{Q[pc \mapsto \ell']\}$.*

PROOF. [Non-constructive proof] Assume $s \stackrel{\ell}{\rightsquigarrow}_{\ell'} sc$ and $\{P\} sc \{Q\}$.

By soundness of the logic of the low-level language, $(m, \sigma) \models P$ and $(m, \sigma) \succ_{sc \rightarrow} (m', \sigma')$ imply $(m', \sigma') \models Q$, for any m, σ, m', σ' .

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

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

The proof for the SGOTO program is the following (the Hoare triples corresponding to those in the WHILE version are highlighted):

$$\begin{array}{c}
\frac{\frac{\frac{\overline{\{J_{2''}\} 2 \{I_3\}}}{\overline{\{I_{2''}\} 2 \{I_3\}}} \quad \frac{\overline{\{J_3\} 3 \{I_4\}}}{\overline{\{I_3\} 3 \{I_4\}}}}{\overline{\{pc = 2 \wedge I_{2'34}\} 2 \{I_{2'34}\}} \quad \overline{\{pc = 3 \wedge I_{2'34}\} 3 \{I_{2'34}\}}} \\
\frac{\overline{\{I_{2'34}\} 2 \oplus 3 \{pc \notin [2, 4] \wedge I_{2'34}\}}}{\overline{\{I_2'\} 2 \oplus 3 \{I_4\}}} \\
\frac{\overline{\{I_2\} 2 \oplus 3 \{I_4\}}}{\overline{\{pc \in [2, 4] \wedge I_{1'24}\} 2 \oplus 3 \{I_{1'24}\}}} \\
\frac{\overline{\{J_4\} 4 \{I_{1'}\}}}{\overline{\{pc = 4 \wedge I_{1'24}\} 4 \{I_{1'24}\}}} \\
\frac{\overline{\{I_{1'24}\} (2 \oplus 3) \oplus 4 \{pc \notin [2, 5] \wedge I_{1'24}\}}}{\overline{\{pc \in [2, 5] \wedge I_{1'25}\} (2 \oplus 3) \oplus 4 \{I_{1'25}\}}} \\
\frac{\overline{\{J_{1'}\} 1 \{I_{25}\}}}{\overline{\{pc = 1 \wedge I_{1'25}\} 1 \{I_{1'25}\}}} \\
\frac{\overline{\{I_{1'25}\} C \{pc \notin [1, 5] \wedge I_{1'25}\}}}{\overline{\{I_{1'}\} C \{I_5\}}} \\
\overline{\{I_1\} C \{I_{5'}\}}
\end{array}$$

where

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

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

and reflected. This can again be proved in two ways: non-constructively and constructively.

Theorem 21 (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 [10] and characteristic to the various proposals that were made thereafter [6,11,2,8,12] 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 [6,11,8] 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 [2], 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, with Java bytecode and .NET CIL being the main motivators. (There is one very notable exception though: Floyd’s logic of control-flow graphs [9].) The logic of Quigley [14] for Java bytecode is based on decompilation, so it applies to pieces of code in the image of a certain compiler. In Benton’s logic [4] for an operand-stack based language, there are global collections of label invariants as in the logic of de Bruin [8]. Bannwart and Müller’s logic [3] extends it to a subset of Java bytecode, with both an operand stack and a call stack.

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 appears in the new work of Tan and Appel [18,19]⁵. Differently from us, their logic is continuation-style and based on the small-step semantics of the unstructured language and step-counting, i.e., the “indexed model” of [1]. The validity of a Hoare triple is defined in terms of “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 $U(sc) \vdash (\ell, \sigma) \twoheadrightarrow^j (\ell', \sigma') \not\models$. 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

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.

to be valid iff any state (ℓ, σ) that k -falsifies Q also $(k + 1)$ -falsifies P . Benton’s independent new work [5] (superseding [4]) is similar by its approach to that of Tan and Appel. The very new XCAP framework of Ni and Shao [13] is continuation-style too, but avoids the need for an indexing-based notion of validity by logic-level reasoning about control flow.

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 [18,19], but they did not introduce a compositional semantics and our logic is simpler than theirs by avoiding continuations and more conventional by being based on a natural semantics and 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 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.

In a sequel to present work [16], we formulated a natural semantics and Hoare logic for a more complicated, operand-stack based low-level language SPUSH. In this language, computations can terminate abruptly with a stack error (stack underflow or wrong type operand types on the stack), hence the natural

semantics has two evaluation relations, one for normal, the other for abnormal evaluations. For this language, we also introduced a type system for attesting stack-error freedom, which weakens the logic and is sound and complete for an appropriate abstracted version of the natural semantics. Pertaining to compilation of proofs, we now know how a Hoare triple derivation can be mechanically transformed alongside optimization of a program, guided by a type derivation embodying the analysis that warrants the optimization [17].

Acknowledgements

We benefitted from the constructive remarks and suggestions of our two referees.

References

- [1] A. Appel, D. McAllester, An indexed model of recursive types for foundational proof-carrying code, *ACM Trans. on Program. Lang. and Syst.* 23(5) (2001) 657–683.
- [2] M. A. Arbib, S. Alagić, Proof rules for **gotos**, *Acta Inform.* 11 (1979) 139–148.
- [3] F. Bannwart, P. Müller, A program logic for bytecode, in: F. Spoto (Ed.), *Proc. of 1st Wksh. on Bytecode Semantics, Verification, Analysis and Transformation, BYTECODE 2005* (Edinburgh, UK, 9 Apr. 2005), *Electron. Notes in Theor. Comput. Sci.*, Vol. 141(1), Elsevier, Amsterdam, 2005, pp. 255–273.
- [4] N. Benton, A typed logic for stacks and jumps, unpublished draft (2004).
- [5] N. Benton, A typed, compositional logic for a stack-based abstract machine, in: K. Yi (Ed.), *Proc. of 3rd Asian Symp. on Programming Languages and Systems, APLAS 2005* (Tsukuba, Japan, 2–5 Nov. 2005), *Lect. Notes in Comput. Sci.* Vol. 3780, Springer, Berlin, 2005, pp. 364–380.
- [6] M. Clint, C. A. R. Hoare, Program proving: jumps and functions, *Acta Informatica* 1 (1972) 214–224.
- [7] S. A. Cook, Soundness and completeness of an axiom system for verification, *SIAM J. of Comput.* 7 (1978) 70–90.
- [8] A. de Bruin, Goto statements: Semantics and deduction systems, *Acta Inform.* 15 (1981) 385–424.
- [9] R. W. Floyd, Assigning meanings to programs, in: J. T. Schwartz (Ed.), *Mathematical Aspects of Computer Science, Proc. of Symp. in Appl. Math.*, Vol. 19, AMS, Providence, RI, 1967, pp. 19–33.

- [10] C. A. R. Hoare, An axiomatic basis for computer programming, *Commun. of ACM* 12 (1969) 576–583.
- [11] T. Kowaltowski, Axiomatic approach to side effects and general jumps, *Acta Inform.* 7 (1977) 357–360.
- [12] O’Donnell, M. J., A critique of the foundations of Hoare style programming logics, *Commun. of ACM* 25 (1982) 927–935.
- [13] Z. Ni, Z. Shao, Certified assembly programming with embedded code pointers, in: *Proc. of 33rd ACM SIGPLAN Symp. on Principles of Programming Languages, POPL 2006* (Long Beach, CA, USA, 12–14 Jan. 2006), ACM Press, New York, 2006, pp. 320–333.
- [14] C. L. Quigley, 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.*, Vol. 2758, Springer, Berlin, 2003, pp. 41–54.
- [15] A. Saabas, T. Uustalu, A compositional natural semantics and Hoare logic for low-level languages, in: P. D. Mosses, I. Ulidowski (Eds.), *Proc. of 2nd Wksh. on Structured Operational Semantics, SOS 2005* (Lisbon, Portugal, 10 July 2005), *Electron. Notes in Theor. Comput. Sci.*, Vol. 156(1), Elsevier, Amsterdam, pp. 151–168.
- [16] A. Saabas, T. Uustalu, Compositional type systems for stack-based low-level languages, in: B. Jay, J. Gudmundsson, (Eds.), *Proc. of 12th Computing, Australasian Theory Symp., CATS 2006* (Hobart, Australia, 16–19 Jan. 2006), *Confs. in Research and Practice in Inform. Techn.*, Vol. 51, Australian Comput. Soc., Sydney, 2006, pp. 27–39.
- [17] A. Saabas, T. Uustalu, Program optimizations with type systems, manuscript (2006).
- [18] G. Tan, A compositional logic for control flow and its application in proof-carrying code, PhD thesis, Dept. of Comput. Sci., Princeton University, 2005.
- [19] G. Tan, A. W. Appel, A compositional logic for control flow, in: E. A. Emerson, K. S. Namjoshi (Eds.), *Proc. of 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation, VMCAI 2006* (Charleston, SC, USA, 8–10 Jan. 2006), *Lect. Notes in Comput. Sci.*, Vol. 3855, Springer, Berlin, 2006, pp. 80–94.

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 [10].

$$\begin{array}{c}
\overline{\sigma \succ x := a \rightarrow \sigma[x \mapsto \llbracket a \rrbracket \sigma]} \text{ :=}_{\text{ns}} \\
\frac{}{\sigma \succ \text{skip} \rightarrow \sigma} \text{ skip}_{\text{ns}} \quad \frac{\sigma \succ s_0 \rightarrow \sigma'' \quad \sigma'' \succ s_1 \rightarrow \sigma'}{\sigma \succ s_0; s_1 \rightarrow \sigma'} \text{ comp}_{\text{ns}} \\
\frac{\sigma \models b \quad \sigma \succ s_t \rightarrow \sigma'}{\sigma \succ \text{if } b \text{ then } s_t \text{ else } s_f \rightarrow \sigma'} \text{ if}_{\text{ns}}^{\text{tt}} \quad \frac{\sigma \not\models b \quad \sigma \succ s_f \rightarrow \sigma'}{\sigma \succ \text{if } b \text{ then } s_t \text{ else } s_f \rightarrow \sigma'} \text{ if}_{\text{ns}}^{\text{ff}} \\
\frac{\sigma \models b \quad \sigma \succ s_t \rightarrow \sigma'' \quad \sigma'' \succ \text{while } b \text{ do } s_t \rightarrow \sigma'}{\sigma \succ \text{while } b \text{ do } s_t \rightarrow \sigma'} \text{ while}_{\text{ns}}^{\text{tt}} \quad \frac{\sigma \not\models b}{\sigma \succ \text{while } b \text{ do } s_t \rightarrow \sigma} \text{ while}_{\text{ns}}^{\text{ff}}
\end{array}$$

Fig. A.1. Natural semantics rules of WHILE

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

$$\begin{array}{l}
a ::= x \mid n \mid a_0 + a_1 \mid \dots \\
b ::= a_0 = a_1 \mid \dots \mid \text{tt} \mid \text{ff} \mid \neg b \mid \dots \\
s ::= x := a \mid \text{skip} \mid s_0; s_1 \mid \text{if } b \text{ then } s_t \text{ else } s_f \mid \text{while } b \text{ do } s_t
\end{array}$$

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: $\mathbf{State} =_{\text{df}} \mathbf{Store} =_{\text{df}} \mathbf{Var} \rightarrow \mathbb{Z}$. The arithmetical and boolean expressions are interpreted relative to stores as integers and truth values by the semantic function $\llbracket - \rrbracket \in \mathbf{AExp} + \mathbf{BExp} \rightarrow \mathbf{Store} \rightarrow \mathbb{Z}$, defined in the denotational style by the usual equations. We write $\sigma \models b$ to say that $\llbracket b \rrbracket \sigma = \text{tt}$.

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 22 (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 unspecified underlying logic over a signature consisting of (a) constants for integers and function and predicate symbols for the standard integer-arithmetical operations and

$$\begin{array}{c}
\overline{\{Q[x \mapsto a]\} x := a \{Q\}} \text{ :=}_{\text{hoa}} \\
\overline{\{P\} \text{ skip } \{P\}} \text{ skip}_{\text{hoa}} \quad \frac{\{P\} s_0 \{R\} \quad \{R\} s_1 \{Q\}}{\{P\} s_0; s_1 \{Q\}} \text{ comp}_{\text{hoa}} \\
\frac{\{b \wedge P\} s_t \{Q\} \quad \{\neg b \wedge P\} s_f \{Q\}}{\{P\} \text{ if } b \text{ then } s_t \text{ else } s_f \{Q\}} \text{ if}_{\text{hoa}} \quad \frac{\{b \wedge P\} s_t \{P\}}{\{P\} \text{ while } b \text{ do } s_t \{ \neg b \wedge P \}} \text{ while}_{\text{hoa}} \\
\frac{P \models P' \quad \{P'\} s \{Q'\} \quad Q' \models Q}{\{P\} s \{Q\}} \text{ conseq}_{\text{hoa}}
\end{array}$$

Fig. A.2. Hoare rules of WHILE

relations and (b) the program variables $x \in \mathbf{Var}$ as constants. For the completeness 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, cf. [7]. 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 $\{\} - \{\} \subseteq \mathbf{Assn} \times \mathbf{Stm} \times \mathbf{Assn}$ defined inductively by the ruleset in Figure A.2.

Theorem 23 (Soundness) *If $\{P\} s \{Q\}$, then, for any σ, σ' and α , $\sigma \models_{\alpha} P$ and $\sigma \succ\text{-}s \rightarrow \sigma'$ imply $\sigma' \models_{\alpha} Q$.*

Theorem 24 (Completeness) *If, for any σ, σ' and α , $\sigma \models_{\alpha} P$ and $\sigma \succ\text{-}s \rightarrow \sigma'$ imply $\sigma' \models_{\alpha} Q$, then $\{P\} s \{Q\}$.*

B Full proofs of Theorems 6, 7, 15, 16, 17, 18

B.1 Proof of Theorem 6

PROOF. By structural induction on the derivation of $(\ell, \sigma) \succ\text{-}sc \rightarrow (\ell', \sigma')$.

Assume $(\ell, \sigma) \succ\text{-}sc \rightarrow (\ell', \sigma')$. We use structural induction on the derivation of $(\ell, \sigma) \succ\text{-}sc \rightarrow (\ell', \sigma')$. We have the following cases:

- The derivation of $(\ell, \sigma) \succ\text{-}sc \rightarrow (\ell', \sigma')$ is

$$\overline{(\ell, \sigma) \succ (\ell, x := a) \rightarrow (\ell + 1, \sigma[x \mapsto \llbracket a \rrbracket \sigma])} \text{ :=}_{\text{ns}}$$

By rule := and Lemma 2, we have $\{(\ell, x := a)\} \vdash (\ell, \sigma) \twoheadrightarrow (\ell + 1, \sigma[x \mapsto \llbracket a \rrbracket \sigma]) \not\approx$.

- The derivation of $(\ell, \sigma) \succ\text{-}sc \rightarrow (\ell', \sigma')$ is

$$\frac{m \neq \ell}{(\ell, \sigma) \succ (\ell, \text{goto } m) \rightarrow (m, \sigma)} \text{goto}_{\text{ns}}$$

By rule goto and Lemma 2, we have $\{(\ell, \text{goto } m)\} \vdash (\ell, \sigma) \rightarrow (m, \sigma) \not\rightsquigarrow$.

- The derivation of $(\ell, \sigma) \succ\text{-}sc \rightarrow (\ell', \sigma')$ is

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

By rule ifngoto^{tt} and Lemma 2, we have $\{(\ell, \text{ifnot } b \text{ goto } m)\} \vdash (\ell, \sigma) \rightarrow (\ell + 1, \sigma) \not\rightsquigarrow$.

- The derivation of $(\ell, \sigma) \succ\text{-}sc \rightarrow (\ell', \sigma')$ is

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

By rule ifngoto^{ff} and Lemma 2, we have $\{(\ell, \text{ifnot } b \text{ goto } m)\} \vdash (\ell, \sigma) \rightarrow (m, \sigma) \not\rightsquigarrow$.

- The derivation of $(\ell, \sigma) \succ\text{-}sc \rightarrow (\ell', \sigma')$ is of the form

$$\frac{\begin{array}{c} \vdots \\ \ell \in \text{dom}(sc_i) \quad (\ell, \sigma) \succ\text{-}sc_i \rightarrow (\ell'', \sigma'') \quad (\ell'', \sigma'') \succ\text{-}sc_0 \oplus sc_1 \rightarrow (\ell', \sigma') \\ \vdots \end{array}}{(\ell, \sigma) \succ\text{-}sc_0 \oplus sc_1 \rightarrow (\ell', \sigma')} \oplus_{\text{ns}}^i$$

where $i = 0$ or 1 : By the induction hypothesis, we have $U(sc_i) \vdash (\ell, \sigma) \rightarrow^* (\ell'', \sigma'') \not\rightsquigarrow$ and $U(sc_0) \cup U(sc_1) \vdash (\ell'', \sigma'') \rightarrow^* (\ell', \sigma') \not\rightsquigarrow$. By Lemma 3, we have $U(sc_0) \cup U(sc_1) \vdash (\ell, \sigma) \rightarrow^* (\ell'', \sigma'')$. Hence, $U(sc_0) \cup U(sc_1) \vdash (\ell, \sigma) \rightarrow^* (\ell', \sigma') \not\rightsquigarrow$.

- The derivation of $(\ell, \sigma) \succ\text{-}sc \rightarrow (\ell', \sigma')$ is

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

By Lemma 2, we have $U(sc) \vdash (\ell, \sigma) \not\rightsquigarrow$. □

B.2 Proof of Theorem 7

PROOF. We use structural induction on sc . Assume we have a stuck reduction sequence $(\ell_0, \sigma_0) \rightarrow \dots \rightarrow (\ell_k, \sigma_k) \not\rightsquigarrow$ for $U(sc)$ ($k \geq 0$) with the implications that $\ell_0, \dots, \ell_{k-1} \in \text{dom}(sc)$, $\ell_k \notin \text{dom}(sc)$. There are the following cases.

- $sc = (\ell, x := a)$: If $\ell_0 = \ell$, then by Lemma 1, rule $:=$ and Lemma 2, the sequence can only be $(\ell, \sigma_0) \twoheadrightarrow (\ell + 1, \sigma_0[x \mapsto \llbracket a \rrbracket \sigma_0]) \not\rightarrow$. We have $(\ell, \sigma_0) \succ (\ell, x := a) \rightarrow (\ell + 1, \sigma_0[x \mapsto \llbracket a \rrbracket \sigma_0])$ by rule $:=_{\text{ns}}$.
If $\ell_0 \neq \ell$, then by Lemma 2 the sequence can only be $(\ell_0, \sigma_0) \not\rightarrow$. We have $(\ell_0, \sigma_0) \succ (\ell, x := a) \rightarrow (\ell_0, \sigma_0)$ by rule ood_{ns} .
- $sc = (\ell, \text{goto } m)$: If $\ell_0 = \ell$ and $m \neq \ell$, then by Lemma 1, rule goto and Lemma 2, the sequence can only be $(\ell, \sigma_0) \twoheadrightarrow (m, \sigma_0) \not\rightarrow$ (we have $m \notin \{\ell\} = \text{dom}(sc)$). We have $(\ell, \sigma_0) \succ (\ell, \text{goto } m) \rightarrow (m, \sigma_0)$ by rule goto_{ns} .
If $\ell_0 = \ell$ and $m = \ell$, then by Lemma 1 and rule goto a stuck reduction sequence is an impossibility (the only reduction sequence of (ℓ, σ_0) is $(\ell, \sigma_0) \twoheadrightarrow (\ell, \sigma_0) \twoheadrightarrow \dots$ and that never reaches a stuck state).
If $\ell_0 \neq \ell$, then by Lemma 2 the sequence can only be $(\ell_0, \sigma_0) \not\rightarrow$. We have $(\ell_0, \sigma_0) \succ (\ell, \text{goto } m) \rightarrow (\ell_0, \sigma_0)$ by rule ood_{ns} .
- $sc = (\ell, \text{ifnot } b \text{ goto } m)$: If $\ell_0 = \ell$ and $\sigma_0 \models b$, then by Lemma 1, rule $\text{ifngoto}^{\text{tt}}$ and Lemma 2 the sequence can only be $(\ell, \sigma_0) \twoheadrightarrow (\ell + 1, \sigma_0) \not\rightarrow$. We have $(\ell, \sigma_0) \succ (\ell, \text{ifnot } b \text{ goto } m) \rightarrow (\ell + 1, \sigma_0)$ by rule $\text{ifngoto}_{\text{ns}}^{\text{tt}}$.
If $\ell_0 = \ell$, $\sigma_0 \not\models b$ and $m \neq \ell$, then by Lemma 1, rule $\text{ifngoto}^{\text{ff}}$ and Lemma 2 the sequence can only be $(\ell, \sigma_0) \twoheadrightarrow (m, \sigma_0) \not\rightarrow$. We have $(\ell, \sigma_0) \succ (\ell, \text{ifnot } b \text{ goto } m) \rightarrow (m, \sigma_0)$ by rule $\text{ifngoto}_{\text{ns}}^{\text{ff}}$.
If $\ell_0 = \ell$, $\sigma_0 \not\models b$ and $m = \ell$, then by Lemma 1 and rule $\text{ifngoto}^{\text{ff}}$ a stuck reduction sequence is an impossibility (the only reduction sequence of (ℓ, σ_0) is $(\ell, \sigma_0) \twoheadrightarrow (\ell, \sigma_0) \twoheadrightarrow \dots$ and that never reaches a stuck state).
If $\ell_0 \neq \ell$, then by Lemma 2 the sequence can only be $(\ell_0, \sigma_0) \not\rightarrow$. We have $(\ell_0, \sigma_0) \succ (\ell, \text{ifnot } b \text{ goto } m) \rightarrow (\ell_0, \sigma_0)$ by rule ood_{ns} .
- $sc = \mathbf{0}$: By Lemma 2 the sequence can only be $(\ell_0, \sigma_0) \not\rightarrow$. We have $(\ell_0, \sigma_0) \succ \mathbf{0} \rightarrow (\ell_0, \sigma_0)$ by rule ood_{ns} .
- $sc = sc_0 \oplus sc_1$: We also invoke mathematical induction on k .
If $\ell \in \text{dom}(sc_i)$ for $i = 0$ or 1 , then it must be that $k > 0$ and that $\ell_1, \dots, \ell_{k-1} \in \text{dom}(sc_0) \cup \text{dom}(sc_1)$ whereas $\ell_k \notin \text{dom}(sc_0) \cup \text{dom}(sc_1)$. Hence there must be a number j , $0 < j \leq k$, such that $\ell_1, \dots, \ell_{j-1} \in \text{dom}(sc_i)$, but $\ell_j \notin \text{dom}(sc_i)$. By Lemma 3 and Lemma 2 this has the consequence that our non-zero length stuck reduction sequence $(\ell_0, \sigma_0) \twoheadrightarrow (\ell_1, \sigma_1) \twoheadrightarrow^* (\ell_k, \sigma_k) \not\rightarrow$ for $U(sc_0) \cup U(sc_1)$ splits into a non-zero length stuck reduction sequence $(\ell_0, \sigma_0) \twoheadrightarrow (\ell_1, \sigma_1) \twoheadrightarrow^* (\ell_j, \sigma_j) \not\rightarrow$ for $U(sc_i)$ and a shorter than k stuck reduction sequence $(\ell_j, \sigma_j) \twoheadrightarrow^* (\ell_k, \sigma_k) \not\rightarrow$ for $U(sc_0) \cup U(sc_1)$. By the outer induction hypothesis, we have $(\ell_0, \sigma_0) \succ sc_i \rightarrow (\ell_j, \sigma_j)$. By the inner induction hypothesis, we have $(\ell_j, \sigma_j) \succ sc_0 \oplus sc_1 \rightarrow (\ell_k, \sigma_k)$. Hence we have $(\ell_0, \sigma_0) \succ sc_0 \oplus sc_1 \rightarrow (\ell_k, \sigma_k)$ by rule \oplus_{ns}^i .
If $\ell \notin \text{dom}(sc_0)$ and $\ell \notin \text{dom}(sc_1)$, then by Lemma 2 the sequence can only be $(\ell_0, \sigma_0) \not\rightarrow$. We have $(\ell_0, \sigma_0) \succ sc_0 \oplus sc_1 \rightarrow (\ell_0, \sigma_0)$ by rule ood_{ns} . \square

B.3 Proof of Theorem 15

PROOF. Assume that $s \stackrel{\ell}{\searrow}_{\ell'} sc$ and $\sigma \succ_s \rightarrow \sigma'$. We have to show that $(\ell, \sigma) \succ_{sc} \rightarrow (\ell', \sigma')$. The proof is by structural induction on the derivation of $\sigma \succ_s \rightarrow \sigma'$.

- The derivation of $\sigma \succ_s \rightarrow \sigma'$ is

$$\frac{}{\sigma \succ_x := a \rightarrow \sigma[x \mapsto \llbracket a \rrbracket \sigma]} :=_{\text{ns}}$$

Then $sc = (\ell, x := a)$ and $\ell' = \ell + 1$. For $(\ell, \sigma) \succ_{sc} \rightarrow (\ell', \sigma')$, we have the derivation

$$\frac{}{(\ell, \sigma) \succ (\ell, x := a) \rightarrow (\ell + 1, \sigma[x \mapsto \llbracket a \rrbracket \sigma])} :=_{\text{ns}}$$

- The derivation of $\sigma \succ_s \rightarrow \sigma'$ is

$$\frac{}{\sigma \succ_{\text{skip}} \rightarrow \sigma} \text{skip}_{\text{ns}}$$

Then $sc = \mathbf{0}$ and $\ell' = \ell$. We have the derivation

$$\frac{}{(\ell, \sigma) \succ_{\mathbf{0}} \rightarrow (\ell, \sigma)} \text{ood}_{\text{ns}}$$

- The derivation of $\sigma \succ_s \rightarrow \sigma'$ is of the form

$$\frac{\begin{array}{c} \vdots \\ \sigma \succ_{s_0} \rightarrow \sigma'' \end{array} \quad \begin{array}{c} \vdots \\ \sigma'' \succ_{s_1} \rightarrow \sigma' \end{array}}{\sigma \succ_{s_0; s_1} \rightarrow \sigma'} \text{comp}_{\text{ns}}$$

Then $s_0 \stackrel{\ell}{\searrow}_{\ell''} sc_0$, $s_1 \stackrel{\ell''}{\searrow}_{\ell'} sc_1$ and $sc = sc_0 \oplus sc_1$ for some ℓ'' , sc_0 , sc_1 . We know that $\text{dom}(sc_0) = [\ell, \ell'')$ and $\text{dom}(sc_1) = [\ell'', \ell']$. If $\ell < \ell''$ and $\ell'' < \ell'$, we have the derivation

$$\frac{\begin{array}{c} \vdots \text{ IH} \\ (\ell, \sigma) \succ_{sc_0} \rightarrow (\ell'', \sigma'') \end{array} \quad \frac{\begin{array}{c} \vdots \text{ IH} \\ (\ell'', \sigma'') \succ_{sc_1} \rightarrow (\ell', \sigma') \end{array} \quad \frac{}{(\ell', \sigma') \succ_{sc_0 \oplus sc_1} \rightarrow (\ell', \sigma')} \text{ood}_{\text{ns}}}{\frac{(\ell, \sigma) \succ_{sc_0} \rightarrow (\ell'', \sigma'') \quad (\ell'', \sigma'') \succ_{sc_0 \oplus sc_1} \rightarrow (\ell', \sigma')}{(\ell, \sigma) \succ_{sc_0 \oplus sc_1} \rightarrow (\ell', \sigma')} \oplus_{\text{ns}}^0} \oplus_{\text{ns}}^1$$

If $\ell = \ell''$, then $s_0 = \text{skip}$ and $\sigma = \sigma''$, and we have the same derivation without the first subderivation by the induction hypothesis and the subsequent inference by \oplus_{ns}^0 . If $\ell'' = \ell'$, then $s_1 = \text{skip}$ and $\sigma'' = \sigma'$, and we have the same derivation without the second subderivation by the induction hypothesis and the subsequent inference by \oplus_{ns}^1 .

- The derivation of $\sigma \succ_s \rightarrow \sigma'$ is of the form

$$\frac{\begin{array}{c} \vdots \\ \sigma \models b \quad \sigma \succ_{s_t} \rightarrow \sigma' \end{array}}{\sigma \succ_{\text{if } b \text{ then } s_t \text{ else } s_f} \rightarrow \sigma'} \text{if}_{\text{ns}}^{\text{tt}}$$

Then $s_t^{\ell+1} \searrow_{\ell''} sc_t, s_f^{\ell''+1} \searrow_{\ell'} sc_f$ and

$$sc = (\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \oplus \underbrace{((sc_t \oplus (\ell'', \text{goto } \ell')) \oplus sc_f)}_{sc_1}^{sc_2}$$

for some ℓ'', sc_t, sc_f . We know that $\text{dom}(sc_t) = [\ell + 1, \ell'']$. If $\ell + 1 < \ell''$, then we have the derivation

$$\frac{\frac{\frac{\frac{\frac{\sigma \models b}{(\ell, \sigma) \succ (\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \rightarrow (\ell + 1, \sigma)}{\sigma \models b}}{(\ell + 1, \sigma) \succ sc_t \rightarrow (\ell'', \sigma')}{\vdots \text{ IH}}}{(\ell + 1, \sigma) \succ sc_2 \rightarrow (\ell', \sigma')} \quad \frac{(\ell', \sigma') \succ sc_2 \rightarrow (\ell', \sigma')}{(\ell', \sigma') \succ sc_1 \rightarrow (\ell', \sigma')}}{(\ell + 1, \sigma) \succ sc_1 \rightarrow (\ell', \sigma')}}{\frac{(\ell, \sigma) \succ (\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \rightarrow (\ell + 1, \sigma)}{\sigma \models b} \quad \frac{(\ell', \sigma') \succ sc \rightarrow (\ell', \sigma')}{(\ell + 1, \sigma) \succ sc \rightarrow (\ell', \sigma')}}{(\ell, \sigma) \succ sc \rightarrow (\ell', \sigma')}}}$$

If $\ell + 1 = \ell''$, then $s_t = \text{skip}$ and $\sigma = \sigma'$, and we have the same derivation without the subderivation by the induction hypothesis and the subsequent inference by \oplus_{ns}^0 .

- The derivation of $\sigma \succ_s \rightarrow \sigma'$ is of the form

$$\frac{\frac{\vdots}{\sigma \not\models b} \quad \sigma \succ_{s_f} \rightarrow \sigma'}{\sigma \succ_{\text{if } b \text{ then } s_t \text{ else } s_f} \rightarrow \sigma'} \text{iff}_{\text{ns}}^{\text{ff}}$$

Then $s_t^{\ell+1} \searrow_{\ell''} sc_t, s_f^{\ell''+1} \searrow_{\ell'} sc_f$ and

$$sc = (\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \oplus \underbrace{((sc_t \oplus (\ell'', \text{goto } \ell')) \oplus sc_f)}_{sc_1}$$

for some ℓ'', sc_t, sc_f . We know that $\text{dom}(sc_f) = [\ell'' + 1, \ell']$. If $\ell'' + 1 < \ell'$, then we have the derivation

$$\frac{\frac{\frac{\frac{\sigma \not\models b}{(\ell, \sigma) \succ (\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \rightarrow (\ell'' + 1, \sigma)}{\sigma \not\models b}}{(\ell'' + 1, \sigma) \succ sc_f \rightarrow (\ell', \sigma')} \quad \frac{(\ell', \sigma') \succ sc_1 \rightarrow (\ell', \sigma')}{(\ell'' + 1, \sigma) \succ sc_1 \rightarrow (\ell', \sigma')}}{\frac{(\ell, \sigma) \succ (\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \rightarrow (\ell'' + 1, \sigma)}{\sigma \not\models b} \quad \frac{(\ell', \sigma') \succ sc \rightarrow (\ell', \sigma')}{(\ell'' + 1, \sigma) \succ sc \rightarrow (\ell', \sigma')}}{(\ell, \sigma) \succ sc \rightarrow (\ell', \sigma')}}{\vdots \text{ IH}}$$

- $s = x := a$: Then $\ell' = \ell + 1$ and $sc = (\ell, x := a)$. The only possible derivation of $(\ell, \sigma) \succ_{sc} \rightarrow (\ell^*, \sigma')$ is

$$\overline{(\ell, \sigma) \succ (\ell, x := a) \rightarrow (\ell + 1, \sigma[x \mapsto \llbracket a \rrbracket \sigma])} \stackrel{:=_{\text{ns}}}{}$$

Hence $\ell^* = \ell + 1 = \ell'$ and the derivation of $\sigma \succ_s \rightarrow \sigma'$ is

$$\overline{\sigma \succ x := a \rightarrow \sigma[x \mapsto \llbracket a \rrbracket \sigma]} \stackrel{:=_{\text{ns}}}{}$$

- $s = \text{skip}$: $\ell' = \ell$ and $sc = \mathbf{0}$. The only possible derivation of $(\ell, \sigma) \succ_{sc} \rightarrow (\ell^*, \sigma')$ is

$$\overline{(\ell, \sigma) \succ \mathbf{0} \rightarrow (\ell, \sigma)} \text{ ood}_{\text{ns}}$$

Hence $\ell^* = \ell = \ell'$ and the derivation of $\sigma \succ_s \rightarrow \sigma'$ is

$$\overline{\sigma \succ \text{skip} \rightarrow \sigma} \text{ skip}_{\text{ns}}$$

- $s = s_0; s_1$: Then $s_0 \stackrel{\ell}{\searrow}_{\ell''} sc_0$, $s_1 \stackrel{\ell''}{\searrow}_{\ell'} sc_1$ and $sc = sc_0 \oplus sc_1$ for some ℓ'', sc_0, sc_1 .

If $\ell < \ell''$ and $\ell'' < \ell'$, then the derivation of $(\ell, \sigma) \succ_{sc} \rightarrow (\ell^*, \sigma')$ must be of the form

$$\frac{\begin{array}{c} \vdots \\ \vdots \\ \overline{(\ell'', \sigma'') \succ_{sc_1} \rightarrow (\ell', \sigma')} \end{array} \quad \overline{(\ell', \sigma') \succ_{sc_0 \oplus sc_1} \rightarrow (\ell', \sigma')}}{\overline{(\ell'', \sigma'') \succ_{sc_0 \oplus sc_1} \rightarrow (\ell', \sigma')}} \oplus_{\text{ns}}^0 \quad \text{ood}_{\text{ns}} \oplus_{\text{ns}}^1$$

Indeed, we know that $\ell \in [\ell, \ell'') = \text{dom}(sc_0)$, so the last inference must be \oplus_{ns}^0 . By the induction hypothesis (a), if $(\ell, \sigma) \succ_{sc_0} \rightarrow (k, \sigma'')$, then $k = \ell''$. Further, $\ell'' \in [\ell'', \ell') = \text{dom}(sc_1)$, so the preceding inference must be \oplus_{ns}^1 . Hence, by the induction hypothesis (a), if $(\ell'', \sigma'') \succ_{sc_1} \rightarrow (k, \sigma')$, then $k = \ell'$. Finally, $\ell' \notin [\ell, \ell') = \text{dom}(sc_0 \oplus sc_1)$, so the inference before is ood_{ns} and hence $\ell^* = \ell'$.

We obtain the following derivation.

$$\frac{\begin{array}{c} \vdots \text{ IH}(b) \\ \sigma \succ_{s_0} \rightarrow \sigma'' \end{array} \quad \begin{array}{c} \vdots \text{ IH}(b) \\ \sigma'' \succ_{s_1} \rightarrow \sigma' \end{array}}{\sigma \succ_{s_0; s_1} \rightarrow \sigma'} \text{ comp}_{\text{ns}}$$

If $\ell = \ell''$, then the derivation of $(\ell, \sigma) \succ_{sc} \rightarrow (\ell^*, \sigma')$ has $\sigma = \sigma''$ and is without the subderivation of $(\ell, \sigma) \succ_{sc_0} \rightarrow (\ell'', \sigma'')$ and the subsequent inference by \oplus_{ns}^0 . In this special case $s_0 = \text{skip}$ and $\sigma \succ_{s_0} \rightarrow \sigma''$ is derived by the rule skip_{ns} . The special case $\ell'' = \ell'$ is handled similarly.

- $s = \text{if } b \text{ then } s_t \text{ else } s_f$: Then $s_t \stackrel{\ell+1}{\searrow}_{\ell''} sc_t$, $s_f \stackrel{\ell''+1}{\searrow}_{\ell'} sc_f$ and $sc = (\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \oplus \underbrace{((sc_t \oplus (\ell'', \text{goto } \ell')) \oplus sc_f)}_{sc_1}$ for some ℓ'', sc_t, sc_f . The derivation of $(\ell, \sigma) \succ_{sc} \rightarrow (\ell^*, \sigma)$ can be of two forms, depending on the value of b in σ .

If it however happens that $\ell'' + 1 = \ell'$, then $\sigma = \sigma'$, $s_f = \text{skip}$, so that $\sigma \succ_{s_f} \rightarrow \sigma'$ is derived by skip_{ns} .

- $s = \text{while } b \text{ do } s_t$: Then $s_t^{\ell+1} \searrow_{\ell''} s_{c_t}$, $\ell' = \ell'' + 1$ and $sc = (\ell, \text{ifnot } b \text{ goto } \ell') \oplus \underbrace{(s_{c_t} \oplus (\ell'', \text{goto } \ell))}_{sc_1}$ for some ℓ'', s_{c_t} . Here we must also invoke structural in-

duction on the derivation of $(\ell, \sigma) \succ_{sc} \rightarrow (\ell^*, \sigma')$. There are two possible derivations forms of $(\ell, \sigma) \succ_{sc} \rightarrow (\ell^*, \sigma)$, depending on the value of b in σ .

- If $\sigma \models b$ and also $\ell + 1 < \ell''$, then the derivation must be of the form

$$\frac{\frac{\frac{\vdots}{(\ell + 1, \sigma) \succ_{s_{c_t}} \rightarrow (\ell'', \sigma'')}}{\frac{\frac{\overline{(\ell'', \sigma'') \succ_{(\ell'', \text{goto } \ell)} \rightarrow (\ell, \sigma'')}}{\overline{(\ell'', \sigma'') \succ_{sc_1} \rightarrow (\ell, \sigma'')}}}{(\ell + 1, \sigma) \succ_{s_{c_t} \oplus (\ell'', \text{goto } \ell)} \rightarrow (\ell, \sigma'')}}{\sigma \models b} \quad \frac{\frac{\vdots}{(\ell, \sigma'') \succ_{sc} \rightarrow (\ell', \sigma')}}{(\ell + 1, \sigma) \succ_{sc} \rightarrow (\ell', \sigma')}}{(\ell, \sigma) \succ_{(\ell, \text{ifnot } b \text{ goto } \ell' + 1)} \rightarrow (\ell + 1, \sigma)} \quad \frac{}{(\ell + 1, \sigma) \succ_{sc} \rightarrow (\ell', \sigma')}}{(\ell, \sigma) \succ_{sc} \rightarrow (\ell', \sigma')}$$

By the outer induction hypothesis (a), $(\ell + 1, \sigma) \succ_{s_{c_t}} \rightarrow (k, \sigma'')$ implies $k = \ell''$. That $(\ell, \sigma'') \succ_{sc} \rightarrow (k, \sigma')$ implies $k = \ell'$ is a result of the inner induction hypothesis (a). Hence, $\ell^* = \ell'$ and we have the derivation

$$\frac{\frac{\vdots \text{ IH}(b)(\text{outer})}{\sigma \models b} \quad \frac{\vdots \text{ IH}(b)(\text{inner})}{\sigma'' \succ_{\text{while } b \text{ do } s_t} \rightarrow \sigma'}}{\sigma \succ_{\text{while } b \text{ do } s_t} \rightarrow \sigma'} \text{while}_{\text{ns}}^{\text{tt}}$$

In the boundary case $\ell + 1 = \ell''$, we have $\sigma = \sigma''$, $s_t = \text{skip}$, so that $\sigma \succ_{s_t} \rightarrow \sigma''$ is derived by skip_{ns} .

- If $\sigma \not\models b$, then the derivation must be of the form

$$\frac{\frac{\sigma \not\models b}{(\ell, \sigma) \succ_{(\ell, \text{ifnot } b \text{ goto } \ell')} \rightarrow (\ell', \sigma)} \quad \overline{(\ell', \sigma) \succ_{s_{c_t}} \rightarrow (\ell', \sigma)}}{(\ell, \sigma) \succ_{sc} \rightarrow (\ell', \sigma)}$$

Hence $\ell^* = \ell'$ and the derivation of $\sigma \succ_s \rightarrow \sigma'$ is

$$\frac{\sigma \not\models b}{\sigma \succ_{\text{while } b \text{ do } s_t} \rightarrow \sigma} \text{while}_{\text{ns}}^{\text{ff}}$$

□

B.5 Constructive proof of Theorem 17

PROOF. [Preservation of Hoare triple derivations] Assume $s^{\ell} \searrow_{\ell'} sc$ and $\{P\} s \{Q\}$. We have to demonstrate that $\{pc = \ell \wedge P\} sc \{pc = \ell' \wedge Q\}$. We use structural induction on the derivation of $\{P\} s \{Q\}$. In order to save

space, we do not explicitly spell out the side conditions of inferences by the consequence rule.

We have the following cases.

- The derivation of $\{P\} s \{Q\}$ is

$$\overline{\{Q[x \mapsto a]\} x := a \{Q\}}$$

Then $sc = (\ell, x := a)$ and $\ell' = \ell + 1$ and the low-level logic derivation is

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

(Note that Q does not contain pc , so $Q[x \mapsto a] = Q[pc, x \mapsto \ell + 1, a]$.)

- The derivation of $\{P\} s \{Q\}$ is

$$\overline{\{Q\} \text{skip} \{Q\}}$$

Then $sc = \mathbf{0}$ and $\ell' = \ell$ and the desired low-level derivation is

$$\overline{\{pc = \ell \wedge Q\} \mathbf{0} \{pc = \ell \wedge Q\}}$$

- The derivation of $\{P\} s \{Q\}$ is

$$\frac{\begin{array}{c} \vdots \\ \{P\} s_0 \{R\} \end{array} \quad \begin{array}{c} \vdots \\ \{R\} s_1 \{Q\} \end{array}}{\{P\} s_0; s_1 \{Q\}}$$

Then $s_0 \xrightarrow{\ell} \searrow_{\ell''} sc_0$, $s_1 \xrightarrow{\ell''} \searrow_{\ell'} sc_1$ and $sc = sc_0 \oplus sc_1$ for some sc_0 , sc_1 and ℓ'' .
Let

$$I =_{\text{df}} (pc = \ell \wedge P) \vee (pc = \ell'' \wedge R) \vee (pc = \ell' \wedge Q)$$

We have the following derivation at the low level:

$$\frac{\frac{\frac{\begin{array}{c} \vdots \text{ IH} \\ \{pc = \ell \wedge P\} sc_0 \{pc = \ell'' \wedge R\} \end{array}}{\{pc \in [\ell, \ell''] \wedge I\} sc_0 \{I\}} \quad \frac{\frac{\begin{array}{c} \vdots \text{ IH} \\ \{pc = \ell'' \wedge R\} sc_1 \{pc = \ell' \wedge Q\} \end{array}}{\{pc \in [\ell'', \ell'] \wedge I\} sc_1 \{I\}}}{\{I\} (sc_0 \oplus sc_1) \{pc \notin [\ell, \ell''] \wedge pc \notin [\ell'', \ell'] \wedge I\}}}{\{pc = \ell \wedge P\} (sc_0 \oplus sc_1) \{pc = \ell' \wedge Q\}}$$

- The derivation of $\{P\} s \{Q\}$ is

$$\frac{\begin{array}{c} \vdots \\ \{b \wedge P\} s_t \{Q\} \end{array} \quad \begin{array}{c} \vdots \\ \{\neg b \wedge P\} s_f \{Q\} \end{array}}{\{P\} \text{if } b \text{ then } s_t \text{ else } s_f \{Q\}}$$

B.6 Constructive proof of Theorem 18

PROOF. [Reflection of normal Hoare triple derivations] Assume $s \stackrel{\ell}{\searrow}_{\ell'} sc$ and $\{P\} sc \{Q\}$. We must show that $\{P[pc \mapsto \ell]\} s \{Q[pc \mapsto \ell']\}$. We use induction on the structure of s . We have the following cases. (As before, we refrain from displaying the side conditions of consequence inferences.)

- $s = x := a$: Then $\ell' = \ell + 1$ and $sc = (\ell, x := a)$. The only possible derivation of $\{P\} sc \{Q\}$ is

$$\frac{\overline{\{(pc = \ell \wedge Q'[x, pc \mapsto a, \ell + 1]) \vee (pc \neq \ell \wedge Q')\} (\ell, x := a) \{Q'\}}}{\{P\} (\ell, x := a) \{Q\}}$$

For the WHILE Hoare triple, we have the following derivation:

$$\frac{\overline{\{Q'[x, pc \mapsto a, \ell + 1]\} x := a \{Q'[pc \mapsto \ell + 1]\}}}{\{P[pc \mapsto \ell]\} x := a \{Q[pc \mapsto \ell + 1]\}}$$

- $s = \text{skip}$: Then $\ell' = \ell$ and $sc = \mathbf{0}$. The only possible derivation of $\{P\} sc \{Q\}$ is

$$\frac{\overline{\{I\} \mathbf{0} \{I\}}}{\{P\} \mathbf{0} \{Q\}}$$

For the WHILE Hoare triple, we have the following derivation:

$$\frac{\overline{\{I[pc \mapsto \ell]\} \text{skip} \{I[pc \mapsto \ell]\}}}{\{P[pc \mapsto \ell]\} \text{skip} \{Q[pc \mapsto \ell]\}}$$

- $s = s_0; s_1$: Then $s_0 \stackrel{\ell}{\searrow}_{\ell''} sc_0$, $s_1 \stackrel{\ell''}{\searrow}_{\ell'} sc_1$ and $sc = sc_0 \oplus sc_1$ for some ℓ'' , sc_0 , sc_1 . The only possible form of a derivation of $\{P\} sc \{Q\}$ is

$$\frac{\frac{\overline{\{pc \in [\ell, \ell''] \wedge I\} sc_0 \{I\}} \quad \overline{\{pc \in [\ell'', \ell'] \wedge I\} sc_1 \{I\}}}{\{I\} (sc_0 \oplus sc_1) \{pc \notin [\ell, \ell''] \wedge pc \notin [\ell'', \ell'] \wedge I\}}}{\{P\} (sc_0 \oplus sc_1) \{Q\}}$$

If $\ell < \ell''$ and $\ell'' < \ell'$, then the derivation in the Hoare logic of WHILE is

$$\frac{\frac{\overline{\{I \in [\ell, \ell''] \wedge I[pc \mapsto \ell]\} s_0 \{I[pc \mapsto \ell'']\}} \quad \overline{\{I'' \in [\ell'', \ell'] \wedge I[pc \mapsto \ell']\} s_1 \{I[pc \mapsto \ell']\}}}{\{I[pc \mapsto \ell]\} s_0 \{I[pc \mapsto \ell'']\} \quad \{I[pc \mapsto \ell'']\} s_1 \{I[pc \mapsto \ell']\}}}{\frac{\overline{\{I[pc \mapsto \ell]\} s_0; s_1 \{I[pc \mapsto \ell']\}}}{\{P[pc \mapsto \ell]\} s_0; s_1 \{Q[pc \mapsto \ell']\}}}$$

If $\ell = \ell''$, then $s_0 = \text{skip}$ and $\{I[pc \mapsto \ell]\} s_0 \{I[pc \mapsto \ell'']\}$ must be derived by skip_{hoa} instead. The special case $\ell'' = \ell'$ is similar.

- $s = \text{if } b \text{ then } s_t \text{ else } s_f$: Then $s_t \stackrel{\ell+1}{\searrow}_{\ell''} sc_t, s_f \stackrel{\ell''+1}{\searrow}_{\ell'} sc_f$ and $sc = (\ell, \text{ifnot } b \text{ goto } \ell'' +$

1) $\oplus (\overbrace{(sc_t \oplus (\ell'', \text{goto } \ell'))}^{sc_2} \oplus sc_f)$ for some ℓ'', sc_t, sc_f . The only possible form of a derivation of $\{P\} sc \{Q\}$ is

$$\begin{array}{c}
\vdots \\
\frac{\{pc \in [\ell + 1, \ell''] \wedge I_2\} sc_t \{I_2\} \quad \frac{\overline{\{C'\}(\ell'', \text{goto } \ell') \{C\}}}{\{pc = \ell'' \wedge I_2\}(\ell'', \text{goto } \ell') \{I_2\}} \quad 9, 10}{\{I_2\} sc_2 \{pc \notin [\ell + 1, \ell''] \wedge pc \neq \ell'' \wedge I_2\}} \quad 7, 8}{\{pc \in [\ell + 1, \ell'' + 1] \wedge I_1\} sc_2 \{I_1\}} \\
\vdots \\
\frac{\overline{\{B'\}(\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \{B\}}}{\{pc = \ell \wedge I\}(\ell, \text{ifnot } b \text{ goto } \ell'' + 1) \{I\}} \quad 3, 4 \quad \frac{\{I_1\} sc_1 \{pc \notin [\ell + 1, \ell'' + 1] \wedge pc \notin [\ell'' + 1, \ell'] \wedge I_1\}}{\{pc \in [\ell + 1, \ell'] \wedge I\} sc_1 \{I\}} \quad 5, 6}{\{I\} sc \{pc \neq \ell \wedge pc \notin [\ell + 1, \ell'] \wedge I\}} \quad 1, 2}{\{P\} sc \{Q\}}
\end{array}$$

where

$$\begin{aligned}
B' &=_{\text{df}} (pc = \ell \wedge ((b \wedge B[pc \mapsto \ell + 1]) \\
&\quad \vee (\neg b \wedge (B[pc \mapsto \ell'' + 1] \vee \ell'' + 1 = \ell)))) \vee (pc \neq \ell \wedge B) \\
C' &=_{\text{df}} (pc = \ell'' \wedge (C[pc \mapsto \ell'] \vee \ell' = \ell'')) \vee (pc \neq \ell'' \wedge C)
\end{aligned}$$

If $\ell + 1 < \ell''$ and $\ell'' + 1 < \ell'$, then the derivation in the Hoare logic of WHILE is

$$\begin{array}{c}
\vdots \text{ IH} \\
\frac{\{ \ell + 1 \in [\ell + 1, \ell''] \} s_t \{I_2[pc \mapsto \ell'']\}}{\{I_2[pc \mapsto \ell + 1]\} s_t \{I_2[pc \mapsto \ell'']\}} \quad \text{c, d} \\
\frac{\{ \ell'' + 1 \in [\ell'' + 1, \ell'] \} s_f \{I_1[pc \mapsto \ell']\}}{\{I_1[pc \mapsto \ell'' + 1]\} s_f \{I_1[pc \mapsto \ell']\}} \quad \text{e, f}}{\frac{\{I[pc \mapsto \ell]\} \text{if } b \text{ then } s_t \text{ else } s_f \{I[pc \mapsto \ell']\}}{\{P[pc \mapsto \ell]\} \text{if } b \text{ then } s_t \text{ else } s_f \{Q[pc \mapsto \ell']\}} \quad \text{a, b}}
\end{array}$$

The entailment $P[pc \mapsto \ell] \models^a I[pc \mapsto \ell]$ follows from entailment 1. The entailment $I[pc \mapsto \ell'] \models^b Q[pc \mapsto \ell']$ follows from entailment 2 using $\ell' \notin [\ell, \ell']$.

The entailment $b \wedge I[pc \mapsto \ell] \models^c I_2[pc \mapsto \ell + 1]$ follows from the following

$$\begin{aligned}
B' &=_{\text{df}} (pc = \ell \wedge (b \wedge B[pc \mapsto \ell + 1]) \vee (\neg b \wedge (B[pc \mapsto \ell'] \vee \ell' = \ell))) \\
&\quad \vee (pc \neq \ell \wedge B) \\
C' &=_{\text{df}} (pc = \ell'' \wedge (C[pc \mapsto \ell] \vee \ell = \ell'')) \vee (pc \neq \ell'' \wedge C)
\end{aligned}$$

If $\ell + 1 < \ell''$, then the derivation in the Hoare logic of WHILE is

$$\frac{\frac{\frac{\vdots \text{IH}}{\{\ell + 1 \in [\ell + 1, \ell''] \wedge I_1[pc \mapsto \ell + 1]\} s_t \{I_1[pc \mapsto \ell'']\}}{\{I_1[pc \mapsto \ell + 1]\} s_t \{I_1[pc \mapsto \ell'']\}} \text{ c, d}}{\{b \wedge I[pc \mapsto \ell]\} s_t \{I[pc \mapsto \ell]\}}}{\frac{\{I[pc \mapsto \ell]\} \text{ while } b \text{ do } s_t \{\neg b \wedge I[pc \mapsto \ell]\}}{\{P[pc \mapsto \ell]\} \text{ while } b \text{ do } s_t \{Q[pc \mapsto \ell']\}} \text{ a, b}}$$

The entailment $P[pc \mapsto \ell] \models^a I[pc \mapsto \ell]$ follows from entailment 1. The entailment $\neg b \wedge I[pc \mapsto \ell] \models^b Q[pc \mapsto \ell']$ follows from the following chain of entailments:

$$\begin{aligned}
\neg b \wedge I[pc \mapsto \ell] &\models \neg b \wedge B'[pc \mapsto \ell] \text{ by 3 using } \ell = \ell \\
&\models B[pc \mapsto \ell'] \\
&\models I[pc \mapsto \ell'] \quad \text{by 4} \\
&\models Q[pc \mapsto \ell'] \quad \text{by 2 using } \ell' \notin [\ell, \ell']
\end{aligned}$$

The entailment $b \wedge I[pc \mapsto \ell] \models^c I_1[pc \mapsto \ell + 1]$ follows from the following chain of entailments:

$$\begin{aligned}
b \wedge I[pc \mapsto \ell] &\models b \wedge B'[pc \mapsto \ell] \text{ by 3 using } \ell = \ell \\
&\models B[pc \mapsto \ell + 1] \\
&\models I[pc \mapsto \ell + 1] \text{ by 4} \\
&\models I_1[pc \mapsto \ell + 1] \text{ by 5 using } \ell + 1 \in [\ell + 1, \ell']
\end{aligned}$$

The entailment $I_1[pc \mapsto \ell''] \models^d I[pc \mapsto \ell]$ follows from the following chain of entailments:

$$\begin{aligned}
I_1[pc \mapsto \ell''] &\models C'[pc \mapsto \ell''] \text{ by 7 using } \ell'' = \ell'' \\
&\models C[pc \mapsto \ell] \\
&\models I_1[pc \mapsto \ell] \text{ by 8} \\
&\models I[pc \mapsto \ell] \text{ by 6 using } \ell \notin [\ell + 1, \ell']
\end{aligned}$$

If $\ell + 1 = \ell''$, then $s_t = \text{skip}$ and $\{I_1[pc \mapsto \ell + 1]\} s_t \{I_1[pc \mapsto \ell'']\}$ must be derived by skip_{hoa} . \square