On the Translation of Procedures to Finite Machines

Abstraction Allows a Clean Proof

Markus Müller-Olm[1] and Andreas Wolf[2] *

[1] Universität Dortmund, Fachbereich Informatik, LS V, 44221 Dortmund, Germany
mmo@ls5.cs.uni-dortmund.de
[2] Christian-Albrechts-Universität, Institut für Informatik und Praktische Mathematik, Olshausenstraße 40, 24098 Kiel, Germany
awo@informatik.uni-kiel.de

Abstract. We prove the correctness of the translation of a prototypic While-language with nested, parameterless procedures to an abstract assembler language with finite stacks. A variant of the well-known wp and wlp predicate transformers, the *weakest relative precondition transformer* wrp, together with a symbolic approach for describing semantics of assembler code allows us to explore assembler programs in a manageable way and to ban finiteness from the scene early.

Keywords: compiler, correctness, refinement, resource-limitation, predicate transformer, procedure, verification

1 Introduction

The construction of compilers is one of the oldest and best studied topics in computer science and neither the interest in this subject nor its importance has declined. Though the range of application of compiler technology has grown, there is still a great need for further understanding the classical setup of program translation. Even if we trust a source program or prove it correct, we cannot rely on the executed object code, if compilation may be erroneous. This motivates us to study the question of how to construct *verified* compilers.

Trusted compilers would permit to certify safety-critical code on the source code level, which promises to be less time-consuming, cheaper, and more reliable than the current practice to inspect the generated machine code [7, 13]. The ultimate goal of compiler verification [1, 2, 4, 6, 8, 9, 11, 12] is to justify such confidence into compilers.

In [10] we studied the question what semantic relationship we can expect to hold between a target program and the source program from which is was generated. Two natural candidate properties from the point of view of program verification are *preservation of total correctness (PTC)* and *preservation of partial correctness (PPC)*. They require that all total or partial correctness assertions valid for the source program remain valid for the target program. Another

* The work of the second author is supported by DFG grant La 426/15-2.

G. Smolka (Ed.): ESOP 2000, LNCS 1782, pp. 290–304, 2000.

characterization is as refinement of the wp and wlp transformers [3] associated to the source and target program. We argued, however, that neither PTC nor PPC is guaranteed by practical compilers. Limited resources on the target processor prohibit the former: PTC implies that the target program terminates regularly, i.e. without a run-time error, whenever regular termination is guaranteed for the source program. But when we implement a source language with full recursion on a finite machine, "StackOverflow" errors will be observed every now and then. On the other hand, optimizing compilers generally do not preserve partial correctness because common transformations, like dead-code elimination, may eliminate code from the program that causes a run-time error. Thus, run-time errors may be replaced by arbitrary results.

As a remedy we proposed in [10] the more general notion of *preservation of relative correctness (PRC)* (recalled in Sect. 3). Relative correctness is *parameterized* in a set A of *accepted failures* and allows thus – in contrast to partial or total correctness – to treat runtime errors and divergence differently. We also studied a corresponding family of predicate transformers wrp_A. It is convenient to refer to predicate transformer (PT) semantics in compiler proofs because there is a powerful data refinement theory for PTs and refinement proofs can be presented in a calculational style by using algebraic laws [5, 6, 9]. PTs also interface directly to correctness proofs for source programs. wrp is meant to permit an elegant treatment of runtime errors and finiteness of machines while staying in the familiar and well-studied realm of predicates and predicate transformers.

The main purpose of the current paper is to show that wrp keeps this promise. More specifically, we employ wrp-based reasoning to prove correct the translation of a prototypic While-language with nested, parameterless procedures to an abstract assembler language with finite stacks, a proof that is also of independent interest. We focus on the control flow implementation by jumps and a return address stack. Due to finiteness of stacks, regular termination of target programs generated from terminating source programs cannot be guaranteed. Nevertheless, wrp allows to establish a variant of PTC in which "StackOverflow" is treated as an accepted failure. As intended, finiteness of stacks vanishes from the scene very early: by taking into account that "StackOverflow" is an accepted error, the laws about wrp derived from the operational semantics are akin to the ones of an idealized assembler with unbounded stacks. Thus, wrp allows to reason about implementations on finite machines without burdening the verification. Another interesting aspect of our proof is that we employ *symbolic* ways of reasoning about assembler language semantics instead of referring to more conventional descriptions by means of an instruction pointer.

The remainder of this paper is organized as follows. Sect. 2 recalls the basics of predicates and predicate transformers. Preservation of relative correctness is discussed briefly in Sect. 3. The abstract assembler language which will serve as the target language is presented in Sect. 4 before the source language, a more common high-level language, is introduced in Sect. 5. The translation scheme is defined in Sect. 6 and the actual correctness proof is given in Sections 7 and 8. We conclude with some remarks in Sect. 9.

2 Preliminaries

Predicates. Assume given a set Σ of *states* s; typically a state is a mapping from variables to values. We identify predicates with the set of satisfying states, so *predicates* are of type $\mathit{Pred} = 2^{\Sigma}$ ranged over by ϕ and ψ. *Pred*, ordered by set inclusion, forms a complete Boolean lattice with top-element $\mathtt{true} = \Sigma$ and bottom-element $\mathtt{false} = \emptyset$.

Predicate transformers. A *predicate transformer (PT)* is a mapping $f : \mathit{Pred} \rightarrow \mathit{Pred}$. Sequential composition of two predicate transformers f and g is defined by $(f;g)(\phi) = f(g(\phi))$ and, hence, is associative and has the identity Id, $\mathit{Id}(\psi) = \psi$, as unit. We restrict the set of PTs to the monotonic ones because this makes sequential composition monotonic. $\mathit{PTrans} \stackrel{\text{def}}{=} (\mathit{Pred} \stackrel{\text{mon.}}{\longrightarrow} \mathit{Pred})$ together with the lifted order $\leq$ defined by $f \leq g \iff \forall \phi \in \mathit{Pred} : f(\phi) \subseteq g(\phi)$ for $f, g \in \mathit{PTrans}$, is also a complete lattice with top-element $\top$, $\top(\psi) = \mathtt{true}$, and bottom-element $\bot$, $\bot(\psi) = \mathtt{false}$.

Fixpoints in complete lattices. The famous theorem of Knaster and Tarski ensures that every monotonic function f on a complete lattice $(L, \leq)$ has a least fixpoint μf and a greatest fixpoint νf. A well-known means for proving properties concerning fixpoints is the following.

Theorem 1 (Fixpoint induction). *For $P \subseteq L$ one has $\mu f \in P$ provided that:*
1. *$\forall C \subseteq P :$ C is totally ordered $: \bigvee C \in P$.* *(Admissibility)*
2. *$\bot \in P$.* *(Base Case)*
3. *$\forall x \in P :$ $x \leq f(x) \Longrightarrow f(x) \in P$.* *(Induction Step)*

3 Relativized Predicate Transformers

In this section we recall relative correctness and relativized predicate transformers, which were introduced and discussed at length in [10], focusing on what's important for our purposes.

We consider imperative programs π intended to compute on a certain non-empty set of states Σ. For the moment, the details of program execution are not of interest; we are only interested in the final outcomes of computations. We thus assume that each program π is furnished with a relation $R(\pi) \subseteq \Sigma \times (\Sigma \cup \Omega)$, where Ω is a non-empty set of *failure (or irregular) outcomes*[1] disjoint from Σ. Typically, Ω contains error states like "DivByZero" and "StackOverflow" and a special symbol ∞ representing divergence.

We use the following conventions for the naming of variables: Σ is ranged over by s, Ω by ω, and $\Sigma \cup \Omega$ by σ. Intuitively, $(s, s') \in R(\pi)$ records that s' is a possible regular result of π from initial state s, $(s, o) \in R(\pi)$ means that error state $o \in \Omega \setminus \{\infty\}$ can be reached from s, and $(s, \infty) \in R(\pi)$ that π may diverge from s, i.e., run forever. $R(\pi)$ can be thought to be derived from an operational or denotational semantics. An example is discussed in Sect. 4.

[1] We use the more neutral word 'outcome' instead of 'result' because some people object to the idea that divergence is a result of a program.

Relative correctness. When evaluating partial correctness assertions all irregular outcomes of programs are accepted; in contrast in total correctness assertions all irregular outcomes are taken as disproof. Relative correctness is built around the idea of *parameterizing* assertions w.r.t. the set of accepted outcomes. The irregular outcomes that are not accepted are taken as disproof.

Assume given a set $A \subseteq \Omega$ of accepted outcomes; this set may contain divergence as in partial correctness. For a given precondition ϕ and postcondition ψ we call program π *relatively correct w.r.t.* ϕ, ψ *and* A if each π-computation starting in a state satisfying ϕ terminates regularly in a state satisfying ψ or has an accepted outcome in A (e.g. π may diverge if $\infty \in A$). More formally:

$$\langle\phi\rangle\pi\langle\psi\rangle_A \quad \text{iff} \quad \forall s, \sigma : s \in \phi \wedge (s, \sigma) \in R(\pi) \ \Rightarrow \ \sigma \in \psi \cup A \ .$$

The classical notions of partial and total correctness are special cases: partial correctness amounts to $\langle\phi\rangle\pi\langle\psi\rangle_\Omega$ and total correctness to $\langle\phi\rangle\pi\langle\psi\rangle_\emptyset$.

Weakest relative preconditions. Relative correctness gives rise to a corresponding predicate transformer semantics of programs. The *weakest relative precondition* of π w.r.t. ψ and A is the set of regular states from which all π-computations either terminate regularly in a state satisfying ψ or have an outcome in A:

$$\mathsf{wrp}_A(\pi)(\psi) \ = \ \{s \in \Sigma \mid \forall\sigma : \ (s, \sigma) \in R(\pi) \ \Rightarrow \ \sigma \in \psi \cup A\} \ .$$

Note that $\mathsf{wrp}_A(\pi) \in PTrans$. Dijkstra's wlp and wp transformers [3] are just the border cases of wrp: $\mathsf{wrp}_\Omega = \mathsf{wlp}$ and $\mathsf{wrp}_\emptyset = \mathsf{wp}$. There is a fundamental difference between wlp and wp regarding the fixpoint definition of repetitive and recursive construct which generalizes as follows to the wrp_A transformers: if $\infty \in A$ we must refer to greatest fixpoints, otherwise to least ones.

The following equivalence generalizes the well-known characterization of partial and total correctness in terms of wlp and wp:

$$\phi \subseteq \mathsf{wrp}_A(\pi)(\psi) \quad \iff \quad \langle\phi\rangle\pi\langle\psi\rangle_A \ .$$

Preserving relative correctness. A natural way to approach translation correctness is to focus on *properties* that transfer from source to target programs. Suppose, for instance, that π is a source program and π' is its translation. We say that the translation *preserves relative correctness w.r.t.* A if

$$\forall\phi, \psi : \ \langle\phi\rangle\pi\langle\psi\rangle_A \ \Rightarrow \ \langle\phi\rangle\pi'\langle\psi\rangle_A \ , \tag{1}$$

i.e., if all relative correctness assertions transfer from π to π'. It is straightforward to show that (1) is equivalent to the refinement inequality $\mathsf{wrp}_A(\pi) \leq \mathsf{wrp}_A(\pi')$. Refinement between predicate transformers can be established by algebraic calculations. We can thus take advantage from such algebraic calculations in semantic compiler proofs. The remaining part of this section is devoted to providing suitable notations that enable this in the scenario studied in this paper.

Concrete predicate transformers. Suppose given three basic sets of syntactic objects: a set *Var* of *variables* x, a set *Expr* of *expressions* e, and a set *BExpr* of *Boolean expressions* b. We assume interpretation functions for expressions and Boolean expressions $\mathcal{E}(e) : \Sigma \to (Val \cup \Omega)$ and $\mathcal{B}(b) : \Sigma \to (\mathbb{B} \cup \Omega)$; here *Val* is the value set of variables and the set $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$ represents the truth values. For the remainder of this paper, states are valuations of variables, i.e. $\Sigma = (Var \to Val)$. Intuitively, results $\mathcal{E}(e)(s), \mathcal{B}(b)(s) \in \Omega$ represent failures (including divergence) arising during evaluation of (Boolean) expressions.

In order to deal with partially defined expressions we assume special types of basic predicates: $\mathsf{def}(e) \stackrel{\text{def}}{=} \{s \mid \mathcal{E}(e)(s) \in Val\}$ and $\mathsf{in}_A(e) \stackrel{\text{def}}{=} \{s \mid \mathcal{E}(e)(s) \in A\}$ for expressions e and $A \subseteq \Omega$. Analogously, we have the predicates $\mathsf{def}(b)$, $\mathsf{in}_A(b)$ and also $b = \mathsf{tt} \stackrel{\text{def}}{=} \{s \mid \mathcal{B}(b)(s) = \mathsf{tt}\}$ and $b = \mathsf{ff}$ for Boolean expressions b.

We consider an assignment $x := e$. The expression e is evaluated in some given state, and if evaluation delivers a regular result it is assigned to x. But evaluation of e might also fail with an outcome ω. It depends on whether $\omega \in A$ or not if we consider this acceptable. Hence, the weakest relative precondition of this assignment w.r.t. A and postcondition ψ is given by

$$(x :=_A e)(\psi) \stackrel{\text{def}}{=} \mathsf{in}_A(e) \cup (\mathsf{def}(e) \cap \psi[e/x]) \ .$$

Another example is a conditional with branches P and Q guarded by b, where the PTs P and Q are wrp_A-transformers. Obviously the weakest relative precondition w.r.t. A and ψ of this construct is $P(\psi)$ resp. $Q(\psi)$ if b evaluates to tt resp. ff. Since evaluation of b can also fail, the weakest relative precondition PT w.r.t. A and postcondition ψ is given by

$$(P \lhd b/A \rhd Q)(\psi) \stackrel{\text{def}}{=} \mathsf{in}_A(b) \cup (b = \mathsf{tt} \cap P(\psi)) \cup (b = \mathsf{ff} \cap Q(\psi)) \ .$$

4 An Abstract Assembler Language

Syntax. The language defined in this section is intended to capture the essence of flat, unstructured assembler code. In this, our main interest is a realistic treatment of control structures. Therefore, labels l taken from a set *Lab* are used to mark the destination of jump instructions as common in assembler languages. In order to keeps things manageable, the language works on a state space with named variables and we provide instructions embodying entire (Boolean) expressions: $\mathsf{asg}(x, e)$ and $\mathsf{cj}(b, l)$. Such instructions should be thought to be 'macros' representing a sequence of more concrete assembler instructions. A language of this kind might be used as a stepping stone on the way down to actual binary machine code.

The set *Instr* consists of *instructions* of the following form.

- $\mathsf{asg}(x, e)$: an assignment instruction,
- $\mathsf{cj}(b, l)$: a conditional jump (on false) to label l,
- $\mathsf{jsr}(l)$: a subroutine jump to label l, and

– ret: a return jump.

We write $\mathsf{goto}(l)$ for $\mathsf{cj}(\mathtt{false}, l)$. It represents an unconditional jump.

An *assembler (or machine) program* m is a finite sequence consisting of instructions and labels where we assume *unique labeling*, formally

$$m \in MP \stackrel{\text{def}}{=} \{m \in (Instr \cup Lab)^* \mid \forall i, j : m_i = m_j \in Lab \;\Rightarrow\; i = j\} \ .$$

Concatenation of programs is denoted by an infix dot "$\cdot$". A program m is called *closed* if every label that has an applied occurrence in m also has a defining occurrence. The set of closed programs is denoted by *CMP*. Here is an example of a closed program computing the factorial of x leaving the result in variable y:

$$\mathsf{asg}(\mathsf{y}, 1) \cdot \mathsf{Loop} \cdot \mathsf{cj}(\mathsf{x} \neq 0, \mathsf{End}) \cdot \mathsf{asg}(\mathsf{y}, \mathsf{x} * \mathsf{y}) \cdot \mathsf{asg}(\mathsf{x}, \mathsf{x} - 1) \cdot \mathsf{goto}(\mathsf{Loop}) \cdot \mathsf{End}$$

Basic operational semantics. A processor executing a machine program will typically use an instruction pointer that points to the next instruction to be executed at any given moment. For reasoning about assembler code, however, it is more convenient to represent the current control point in a more symbolic manner: we partition the executed program m into two parts u, v such that $m = u \cdot v$ and that the next instruction to be executed is just the first instruction of v. Progress of execution can nicely be expressed by partitioning the same code sequence differently. *PMP* (partitioned machine programs) denotes the set of pairs $\langle u, v \rangle$ such that $u \cdot v \in CMP$.

Similarly, we prefer to work with a symbolic representation of the stack of return addresses; such a stack is necessary to execute jump-subroutine and return instructions. The idea is to use a stack of partitioned code sequences (modeled by a member of PMP^*) instead of a stack of addresses.

The basic semantics of the abstract assembler language is an operational semantics built around the ideas just described. It works on configurations of the form $\langle u, v, a, s \rangle$, where $\langle u, v \rangle \in PMP$ models the current control point ($u \cdot v$ is the executed program), $a \in PMP^*$ is the symbolic representation of the return stack, and $s \in \Sigma$ is the current state. Thus,

$$\Gamma \stackrel{\text{def}}{=} \{\langle u, v, a, s \rangle \mid \langle u, v \rangle \in PMP \wedge a \in PMP^* \wedge s \in \Sigma\}$$

is the set of *regular configurations*. In order to treat error situations, we use the members of Ω as *irregular configurations*. Table 1 defines the transition relation $\rightarrow \subseteq \Gamma \times (\Gamma \cup \Omega)$ of an abstract machine executing assembler programs.

Let us consider the rules in more detail. [Asg1] applies if expression e evaluates without error to a value in the current state s: the machine changes the value of x accordingly – the new state is $s[x \mapsto \mathcal{E}(e)(s)]$ – and transfers control to the subsequent instruction by moving $\mathsf{asg}(x, e)$ to the end of the u-component. [Asg2] is used if evaluation of e fails in the current state: the failure value $\mathcal{E}(e)(s)$ is just propagated. [Cj1] describes that a conditional jump $\mathsf{cj}(b, l)$ is not taken if b evaluates to tt in the current state: control is simply transferred to the subsequent instruction. If b evaluates to ff, rule [Cj2] applies and the control is transferred to

[Asg1] $$\frac{\mathcal{E}(e)(s) \in \Sigma}{\langle u, \mathsf{asg}(x,e)\cdot v, a, s\rangle \to \langle u \cdot \mathsf{asg}(x,e), v, a, s[x \mapsto \mathcal{E}(e)(s)]\rangle}$$

[Asg2] $$\frac{\mathcal{E}(e)(s) \in \Omega}{\langle u, \mathsf{asg}(x,e)\cdot v, a, s\rangle \to \mathcal{E}(e)(s)}$$

[Cj1] $$\frac{\mathcal{B}(b)(s) = \mathsf{tt}}{\langle u, \mathsf{cj}(b,l)\cdot v, a, s\rangle \to \langle u \cdot \mathsf{cj}(b,l), v, a, s\rangle}$$

[Cj2] $$\frac{\mathcal{B}(b)(s) = \mathsf{ff}\ ,\ u\cdot \mathsf{cj}(b,l)\cdot v = x \cdot l \cdot y}{\langle u, \mathsf{cj}(b,l)\cdot v, a, s\rangle \to \langle x, l \cdot y, a, s\rangle}$$

[Cj3] $$\frac{\mathcal{B}(b)(s) \in \Omega}{\langle u, \mathsf{cj}(b,l)\cdot v, a, s\rangle \to \mathcal{B}(b)(s)}$$

[Jsr1] $$\frac{u \cdot \mathsf{jsr}(l)\cdot v = x \cdot l \cdot y}{\langle u, \mathsf{jsr}(l)\cdot v, a, s\rangle \to \langle x, l\cdot y, a\cdot\langle u\cdot \mathsf{jsr}(l), v\rangle, s\rangle}$$

[Jsr2] $\langle u, \mathsf{jsr}(l)\cdot v, a, s\rangle \to$ "$\mathsf{StackOverflow}$"

[Ret1] $\langle u, \mathsf{ret}\cdot v, a\cdot\langle x, y\rangle, s\rangle \to \langle x, y, a, s\rangle$

[Ret2] $\langle u, \mathsf{ret}\cdot v, \varepsilon, s\rangle \to$ "$\mathsf{EmptyStack}$"

[Label] $\langle u, l\cdot v, a, s\rangle \to \langle u\cdot l, v, a, s\rangle$

Table 1. Operational semantics of the assembler language

label l, the position of which is determined by the premise $u \cdot \mathsf{cj}(b,l)\cdot v = x\cdot l\cdot y$. [CJ3] propagates errors resulting from evaluation of b. [Jsr1] is concerned with a subroutine jump to label l. Similarly to rule [Cj2], control is transferred to label l. Additionally, the machine stores the return address by pushing $\langle u\cdot \mathsf{jsr}(l), v\rangle$ onto the symbolically modeled return stack a. If execution subsequently reaches a ret instruction, execution of $\langle u\cdot \mathsf{jsr}(l), v\rangle$ is resumed as specified by [Ret1]. A processor with finite memory will not always be able to stack a return address when executing a jsr instruction. We model this by rule [Jsr2] that allows the machine to report "StackOverflow" spontaneously. Of course, in an actual processor the choice between regular stacking and overflow will be mutually exclusive and not just non-deterministic as in our model. This could be modeled by furnishing [Jsr2] by a premise *StackFull* and [Jsr1] by a premise $\neg$*StackFull*, where *StackFull* is a (complicated) condition depending on the current state of the machine. Finally, [Ret2] reports an error if a ret instruction is executed on an empty return stack, and [Label] allows to skip labels.

The evaluation of m in state s starts in the initial configuration $\langle\varepsilon, m, \varepsilon, s\rangle$, i.e. with the first instruction of m and with an empty stack. Execution terminates regularly if a configuration of the form $\langle u, \varepsilon, a, s'\rangle$ is reached; other possible outcomes are reachable error configurations ω, and ∞, if there is an infinite sequence of transitions from $\langle\varepsilon, m, \varepsilon, s\rangle$. Based on this intuition, we could now define a relational semantics $R(m)$ for a given program $m \in \mathit{CMP}$ following the lines of the definition below. $R(m)$ would give rise to a family of predicate transformers $\mathsf{wrp}_A(m)$. Up to this point $\mathsf{wrp}_A(m)$ would be known only with reference to the operational semantics. In order to allow a reasoning on a more abstract level we would like to derive sufficiently strong laws about $\mathsf{wrp}_A(m)$ from the operational semantics first; afterwards we would use just these laws in our reasoning without referring directly to the operational semantics.

Unfortunately, this approach fails for $\mathsf{wrp}_A(m)$: only very weak laws can be established. The main problem is that the behavior of jump and jump-subroutine instructions cannot adequately be described without having context information available. We, therefore, work with a semantics of machine programs that takes the sequential context as well as the stack context into account.

For $\langle u, v\rangle \in PMP$ and $a \in PMP^*$ we define

$$\begin{aligned} R(u,v,a) \stackrel{\text{def}}{=} \; & \{(s,s') \mid \exists u',a' : \langle u,v,a,s\rangle \rightarrow^* \langle u',\varepsilon,a',s'\rangle\} \\ & \cup \{(s,\omega) \mid \langle u,v,a,s\rangle \rightarrow^* \omega\} \\ & \cup \{(s,\infty) \mid \langle u,v,a,s\rangle \rightarrow^\infty\} \;, \end{aligned}$$

where $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$, and $\rightarrow^\infty$ the existence of an infinite path. This definition induces a family of predicate transformers $\mathsf{wrp}_A(u,v,a)$ and it is this family that we are using in our reasoning. We can define $R(m)$ and $\mathsf{wrp}_A(m)$ by $R(m) = R(\varepsilon,m,\varepsilon)$ and $\mathsf{wrp}_A(m) = \mathsf{wrp}_A(\varepsilon,m,\varepsilon)$.

The laws in Table 2 can now be proved from the operational semantics. Technically these laws are just derived properties but they can also be read as axioms about the total behavior of a machine. Law [Asg-wrp], e.g., tells us about a machine started in a situation where it executes an asg-instruction first: its total behavior can safely be assumed to be composed of the respective assignment to x and the total behavior of a machine started just after the assignment instruction. The other laws have a similar interpretation. Together the laws allow a kind of symbolic execution of assembler programs. But we do not have to refer to low-level concepts like execution sequences; instead we can use more abstract properties, e.g., that $\geq$ is an ordering.

All these laws can be strengthened to equalities. We state them as inequalities in order to stress that just one direction is needed in the following. Refinement allows to use safe approximations on the right hand side instead of fully accurate descriptions. This allows to reason safely about instructions whose effect is either difficult to capture or not fully specified by the manufacturer of the processor [9]. If, for example, [Jsr1] and [Jsr2] are furnished with a condition *StackFull* as discussed above, the refinement inequality stated in [Jsr-wrp] becomes proper, because jsr would definitely lead to the acceptable error "StackOverflow" if *StackFull* holds. Therefore, the PT on the left hand side would succeed for all states satisfying *StackFull* irrespective of the post-condition, while the right hand side may fail.

Note that the premise "StackOverflow" $\in A$ of the law [Jsr-wrp] is essential. If "StackOverflow" is considered unacceptable ("StackOverflow" $\notin A$), we have $\mathsf{wrp}_A(u, \mathsf{jsr}(l)\cdot v, a) = \bot$ as a consequence of [Jsr2]. This means that jsr cannot be used to implement any non-trivial statement. If the more precise operational model with a *StackFull* predicate is used, $\mathsf{wrp}_A(u, \mathsf{jsr}(l)\cdot v, a)$ is better than $\bot$ but

[Asg-wrp]	$\mathsf{wrp}_A(u, \mathsf{asg}(x,e) \cdot v, a) \geq (x :=_A e) \,;\, \mathsf{wrp}_A(u \cdot \mathsf{asg}(x,e), v, a)$
[Cj-wrp]	$\mathsf{wrp}_A(u, \mathsf{cj}(b,l) \cdot v, a) \geq \mathsf{wrp}_A(u \cdot \mathsf{cj}(b,l), v, a) \lhd b/A \rhd \mathsf{wrp}_A(x, l \cdot y, a)$, if $u \cdot \mathsf{cj}(b,l) \cdot v = x \cdot l \cdot y$
[Goto-wrp]	$\mathsf{wrp}_A(u, \mathsf{goto}(l) \cdot v, a) \geq \mathsf{wrp}_A(x, l \cdot y, a)$, if $u \cdot \mathsf{goto}(l) \cdot v = x \cdot l \cdot y$
[Jsr-wrp]	$\mathsf{wrp}_A(u, \mathsf{jsr}(l) \cdot v, a) \geq \mathsf{wrp}_A(x, l \cdot y, a \cdot \langle u \cdot \mathsf{jsr}(l), v\rangle)$, if $u \cdot \mathsf{jsr}(l) \cdot v = x \cdot l \cdot y$ and "$\mathsf{StackOverflow}$" $\in A$
[Ret-wrp]	$\mathsf{wrp}_A(u, \mathsf{ret} \cdot v, a \cdot \langle x, y\rangle) \geq \mathsf{wrp}_A(x, y, a)$
[Label-wrp]	$\mathsf{wrp}_A(u, l \cdot v, a) \geq \mathsf{wrp}_A(u \cdot l, v, a)$
[Term-wrp]	$\mathsf{wrp}_A(u, \varepsilon, a) \geq \mathit{Id}$

Table 2. wrp-laws for the assembler language

any non-trivial approximation will involve the *StackFull* predicate. This would force us to keep track of the storage requirements when we head for a verified compilation. As the recursion depth of programs is in general not computable, we could not justify translation of arbitrary recursive procedures.

5 A Simple High-Level Language

As a prototypic instance of a high-level language we consider a While-language with parameterless, nested procedures. Such a language is adequate for studying the control-flow aspects of translation of ALGOL-like programming languages.

Syntax. We define the set of programs, *Prog*, by the following grammar. In order to distinguish programs clearly from the corresponding semantic predicate transformers from Sect. 3 we use an abstract kind of syntax.

$$\pi ::= \mathsf{assign}(x,e) \mid \mathsf{seq}(\pi_1, \pi_2) \mid \mathsf{if}(b, \pi_1, \pi_2) \mid \mathsf{while}(b, \pi) \mid \mathsf{call}(p) \mid \mathsf{blk}(p, \pi_p, \pi_b)$$

In this grammar, x ranges over the variables in Var, b and e over *BExpr* and *Expr*, and p over a set *ProcName* of procedure identifiers.

$\mathsf{blk}(p, \pi_p, \pi_b)$ is a block in which a (possibly recursive) local procedure p with body π_p is declared. π_b is the body of the block; it might call p as well as more globally defined procedures. The semantics below ensures static scoping and so the translation of the next section has to guarantee static scoping as well. Note that nesting of procedure declarations and even re-declaration is allowed. Our exposition generalizes straightforwardly to blocks in which a system of mutually recursive procedures can be declared instead of just a single procedure. We refrained from treating this more general case only, as it burdens the notation a bit without bringing more insight. The intuitive semantics of the other syntactic operators should be clear from their name.

Semantics. Now we furnish the While-language with a predicate transformer semantics. Due to lack of space, we cannot follow the lines from the last section; instead we *postulate* the resulting predicate transformer semantics directly. Nevertheless the oncoming definitions should be read as *laws* derived from a more concrete semantics. In [10] we justified such definitions briefly for a language without procedures.

In order to give a compositional semantics, we refer as usual to *environments* $\eta \in Env \stackrel{\text{def}}{=} (ProcName \to PTrans)$, mapping procedure identifiers to the weakest relative precondition transformer of their body. The environment is taken by wrp as an additional argument written as a superscript.

$$\begin{aligned}
\mathsf{wrp}^\eta_A(\mathsf{assign}(x, e)) &= (x :=_A e) \\
\mathsf{wrp}^\eta_A(\mathsf{seq}(\pi_1, \pi_2)) &= \mathsf{wrp}^\eta_A(\pi_1) \ ; \mathsf{wrp}^\eta_A(\pi_2) \\
\mathsf{wrp}^\eta_A(\mathsf{if}(b, \pi_1, \pi_2)) &= \mathsf{wrp}^\eta_A(\pi_1) \lhd b/A \rhd \mathsf{wrp}^\eta_A(\pi_2) \\
\mathsf{wrp}^\eta_A(\mathsf{while}(b, \pi)) &= \lambda\mathcal{W} \\
\mathsf{wrp}^\eta_A(\mathsf{call}(p)) &= \eta(p) \\
\mathsf{wrp}^\eta_A(\mathsf{blk}(p, \pi_p, \pi_b)) &= \mathsf{wrp}^{\eta[p \mapsto \lambda\mathcal{P}]}_A(\pi_b)
\end{aligned}$$

In the clauses for while and blk, $\lambda = \nu$ if $\infty \in A$, and $\lambda = \mu$ otherwise, i.e. we have to take the greatest fixpoint if divergence is accepted (like in partial correctness semantics) and the least fixpoint otherwise (see [10]).

Let us discuss briefly each of the clauses in turn. The assignment law takes advantage from the assignment combinator defined in Sect. 3. The weakest precondition of a sequential composition is the weakest precondition of the first statement establishing the weakest precondition of the second statement. A conditional's weakest precondition depends on the validity of the guard. Operationally a loop is unrolled as long as the guard holds, hence the weakest precondition PT of a loop is a fixpoint of the well known semantical function $\mathcal{W} : PTrans \to PTrans$, where $\mathcal{W}(X) = (\pi; X) \lhd b/A \rhd Id$. Application of the environment in question captures the call-case. A block's weakest precondition in some given environment is the weakest precondition of the body in a varied environment that contains a new binding for the local procedure declared in that block. The weakest precondition of that procedure is a fixpoint of the function $\mathcal{P} : PTrans \to PTrans$, where $\mathcal{P}(X) = \mathsf{wrp}^{\eta[p \mapsto X]}_A(\pi_p)$.

Complete programs are interpreted in the environment $\perp_{Env}$ that bind all procedures to the $\perp$ predicate transformer, because otherwise the call of an undeclared procedure would miraculously have a non-trivial meaning. Hence, when comparing a complete program π to its translation, we refer to $\mathsf{wrp}^{\perp_{Env}}_A(\pi)$.

6 Specification of Compilation

In Table 3 we inductively define a compiling relation $\mathcal{C} \subseteq Prog \times MP \times Dict$. Here $Dict = (ProcName \stackrel{\text{fin}}{\to} Lab)$ is the set of *dictionaries* that intuitively map procedure names to labels where code for the corresponding body can be found.

$$\text{[Assign]}\quad \mathcal{C}(\mathsf{assign}(x,e), \mathsf{asg}(x,e), \delta)$$

$$\text{[Seq]}\quad \frac{\mathcal{C}(\pi_1, m_1, \delta),\ \mathcal{C}(\pi_2, m_2, \delta)}{\mathcal{C}(\mathsf{seq}(\pi_1,\pi_2), m_1 \cdot m_2, \delta)}$$

$$\text{[If]}\quad \frac{\mathcal{C}(\pi_1, m_1, \delta),\ \mathcal{C}(\pi_2, m_2, \delta)}{\mathcal{C}(\mathsf{if}(b,\pi_1,\pi_2), \mathsf{cj}(b,l_1) \cdot m_1 \cdot \mathsf{goto}(l_2) \cdot l_1 \cdot m_2 \cdot l_2, \delta)}$$

$$\text{[While]}\quad \frac{\mathcal{C}(\pi, m, \delta)}{\mathcal{C}(\mathsf{while}(b,\pi), l_0 \cdot \mathsf{cj}(b,l_1) \cdot m \cdot \mathsf{goto}(l_0) \cdot l_1, \delta)}$$

$$\text{[Call]}\quad \frac{p \in \mathrm{dom}(\delta)}{\mathcal{C}(\mathsf{call}(p), \mathsf{jsr}(\delta(p)), \delta)}$$

$$\text{[Blk]}\quad \frac{\mathcal{C}(\pi_p, m_p, \delta[p \mapsto l_p]),\ \mathcal{C}(\pi_b, m_b, \delta[p \mapsto l_p])}{\mathcal{C}(\mathsf{blk}(p,\pi_p,\pi_b), \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b, \delta)}$$

Table 3. Compiling relation

We have $\mathcal{C}(\pi, m, \delta)$ if machine program m is a possible compiling result of source program π assuming that dictionary δ assigns appropriate labels to free procedure names. The program

$$\mathsf{seq}(\mathsf{assign}(\mathsf{y},1), \mathsf{while}(\mathsf{x} > 0, \mathsf{seq}(\mathsf{assign}(\mathsf{y}, \mathsf{x} * \mathsf{y}), \mathsf{assign}(\mathsf{x}, \mathsf{x} - 1)))) \ ,$$

for instance, may be compiled to the assembler program computing the factorial function in Sect. 4 irrespective of the dictionary δ.

Note that the typing constraint $m \in \mathit{MP}$ guarantees that target programs are labeled uniquely. An advantage of a relational specification over a functional compiling-function is that certain aspects, like choice of labels here, can be left open for a later design stage of the compiler.

7 Correctness of Compilation

This section is concerned with proving correctness of the translation specified in the previous section. As discussed in the introduction, the translation cannot be correct in the sense of preservation of total correctness (PTC), as our assembler language might report "StackOverflow" on executing a jsr instruction and thus regularly terminating source programs might be compiled to target programs that do not terminate regularly. Nevertheless source programs that do not diverge are never compiled to diverging target programs. But PTC identifies divergence and runtime-errors and, therefore, it cannot treat this scenario appropriately. A main purpose of this paper is to show how the greater selectivity of wrp_A-based reasoning allows a more adequate treatment by appropriate choice of A. We treat "StackOverflow" as an acceptable outcome but ∞ as an unacceptable one. This gives rise to a relativized version of PTC. We comment on the proof for relativized versions of PPC in the conclusion.

Theorem 2. *Suppose* $\infty \notin A$ *and* "$\mathsf{StackOverflow}$" $\in A$. *Then for all* π, m*:*

$$\mathcal{C}(\pi, m, \emptyset) \;\Rightarrow\; \mathsf{wrp}_A(m) \geq \mathsf{wrp}_A^{\perp_{Env}}(\pi) \enspace .$$

Thus, if a program π is compiled to an assembler program m in an empty dictionary, relative correctness is preserved. Note that the premise of the compiling rule [Call] guarantees, that non-closed programs cannot be compiled with an empty dictionary.

When we try to prove Theorem 2 by a structural induction we encounter two problems. Firstly, when machine programs are put together to implement composed programs, like in the [Seq] or [If] rule, the induction hypothesis cannot directly be applied because it is concerned with code for the components in isolation while, in the composed code, the code runs in the context of other code. Our approach to deal with this problem is to establish a stronger claim that involves a universal quantification over all contexts. More specifically, we show $\mathsf{wrp}_A(u, m \cdot v, a) \geq \mathsf{wrp}_A^{\eta}(\pi) \;;\; \mathsf{wrp}_A(u \cdot m, v, a)$ for all surrounding code sequences u, v and stack contexts a. Note how the sequential composition with $\mathsf{wrp}_A(u \cdot m, v, a)$ on the right hand side beautifully expresses that m transfers control to the subsequent code and that the stack is left unchanged.

Secondly, when considering the call-case, some knowledge about the bindings in the dictionary δ is needed. To solve this problem we use the following predicate.

$$\begin{aligned}\mathsf{fit}(\eta, \delta, u) \;\stackrel{\text{def}}{\Longleftrightarrow}\; & \forall q \in \mathrm{dom}(\delta) : \exists x, y : \\ & \quad x \cdot \delta(q) \cdot y = u \;\wedge \\ & \quad \forall e, f, g :\; \mathsf{wrp}_A(x, \delta(q) \cdot y, g \cdot \langle e, f\rangle) \geq \eta(q) \,;\, \mathsf{wrp}_A(e, f, g) \enspace .\end{aligned}$$

It expresses that the bindings in δ together with the assembler code u 'fit' to the bindings in the semantic environment η. The first conjunct says that the context provides a corresponding label for each procedure q bound by δ; the second conjunct tells us that the code following this label implements q's binding in η and proceeds with the code on top of the return stack. This is just what is needed in the call-case of the induction. The code generated for blocks has to ensure that this property remains valid for newly declared procedures.

Putting the pieces together we are going to prove the following.

Lemma 3. *Suppose* $\infty \notin A$ *and* "$\mathsf{StackOverflow}$" $\in A$. *For all* $\pi, m, u, v, a, \eta, \delta$*:*

$$\mathcal{C}(\pi, m, \delta) \wedge \mathsf{fit}(\eta, \delta, u \cdot m \cdot v) \;\Rightarrow\; \mathsf{wrp}_A(u, m \cdot v, a) \geq \mathsf{wrp}_A^{\eta}(\pi) \,;\, \mathsf{wrp}_A(u \cdot m, v, a) \enspace .$$

Theorem 2 follows by the instantiation $u = v = \varepsilon$, $a = \varepsilon$, $\eta = \perp_{Env}$, $\delta = \emptyset$ using the [Term-wrp] law and the fact that $\mathsf{wrp}_A(m) = \mathsf{wrp}_A(\varepsilon, m, \varepsilon)$.

8 Proof of Lemma 3

The proof is by structural induction on π. So consider some arbitrarily chosen $\pi, m, u, v, a, \eta, \delta$ such that $\mathcal{C}(\pi, m, \delta)$ and $\mathsf{fit}(\eta, \delta, u \cdot m \cdot v)$, and assume that for all component programs the claim of Lemma 3 holds. As usual, we proceed by a case analysis on the structure of π. In each case we perform a kind of 'symbolic execution' of the corresponding assembler code using the wrp-laws from Sect. 4. The assumptions about fit will solve the call-case elegantly, the while- and blk-case moreover involve some fixpoint reasoning.

Due to lack of space we can discuss here only the cases concerned with procedures: call and blk.

Case a.) $\pi = \mathsf{call}(p)$. By the [Call] rule, $m = \mathsf{jsr}(\delta(p))$ and $p \in \mathrm{dom}(\delta)$. As a consequence of $\mathsf{fit}(\eta, \delta, u \cdot m \cdot v)$ there exist x and y such that $x \cdot \delta(p) \cdot y = u \cdot \mathsf{jsr}(\delta(p)) \cdot v$. Now,

$$\begin{array}{ll} & \mathsf{wrp}_A(u, \mathsf{jsr}(\delta(p)) \cdot v, a) \\ \geq & \quad \{\text{Law [Jsr-wrp], ``}\mathsf{StackOverflow}\text{''} \in A\text{, existence of } x \text{ and } y\} \\ & \mathsf{wrp}_A(x, \delta(p) \cdot y, a \cdot \langle u \cdot \mathsf{jsr}(\delta(p)), v\rangle) \\ \geq & \quad \{\text{Second conjunct of } \mathsf{fit}(\eta, \delta, u \cdot m \cdot v)\} \\ & \eta(p) \;;\; \mathsf{wrp}_A(u \cdot \mathsf{jsr}(\delta(p)), v, a) \\ = & \quad \{\text{Definition of call semantics}\} \\ & \mathsf{wrp}_A^\eta(\pi) \;;\; \mathsf{wrp}_A(u \cdot \mathsf{jsr}(\delta(p)), v, a) \enspace . \end{array}$$

Case b.) $\pi = \mathsf{blk}(p, \pi_p, \pi_b)$. By the [Blk] rule, there are assembler programs m_p, m_b and labels l_p, l_b such that $m = \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b$ and $\mathcal{C}(\pi_p, m_p, \delta[p \mapsto l_p])$ and $C(\pi_b, m_b, \delta[p \mapsto l_p])$ hold.

We would like to calculate as follows:

$$\begin{array}{ll} & \mathsf{wrp}_A(u, \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b \cdot v, a) \\ \geq & \quad \{\text{Laws [Goto-wrp] and [Label-wrp]}\} \\ & \mathsf{wrp}_A(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b, m_b \cdot v, a) \\ \geq & \quad \{\text{Induction hypothesis: } \mathcal{C}(\pi_b, m_b, \delta[p \mapsto l_p]) \text{ holds}\} \\ & \mathsf{wrp}_A^{\eta[p \mapsto \mu\mathcal{P}]}(\pi_b) \;;\; \mathsf{wrp}_A(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b, v, a) \\ = & \quad \{\text{Definition of block semantics}\} \\ & \mathsf{wrp}_A^\eta(\mathsf{blk}(p, \pi_p, \pi_b)) \;;\; \mathsf{wrp}_A(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b, v, a) \enspace . \end{array}$$

In order to apply the induction hypothesis in the second step, however, we have to check $\mathsf{fit}(\eta[p \mapsto \mu\mathcal{P}], \delta[p \mapsto l_p], u \cdot m \cdot v)$, i.e. that for all $q \in \mathrm{dom}(\delta[p \mapsto l_p])$

$$\begin{array}{l} \exists x, y : \\ \quad x \cdot \delta[p \mapsto l_p](q) \cdot y = u \cdot m \cdot v \;\; \wedge \\ \quad \forall e, f, g : \;\mathsf{wrp}_A(x, \delta[p \mapsto l_p](q) \cdot y, g \cdot \langle e, f\rangle) \;\geq\; \eta[p \mapsto \mu\mathcal{P}](q) \;;\; \mathsf{wrp}_A(e, f, g) \enspace . \end{array} \tag{2}$$

So suppose given $q \in \mathrm{dom}(\delta[p \mapsto l_p])$. If $q \neq p$, (2) reduces to

$$\begin{array}{l} \exists x, y : \; x \cdot \delta(q) \cdot y = u \cdot m \cdot v \;\; \wedge \\ \qquad \forall e, f, g : \;\mathsf{wrp}_A(x, \delta(q) \cdot y, g \cdot \langle e, f\rangle) \;\geq\; \eta(q) \;;\; \mathsf{wrp}_A(e, f, g) \enspace , \end{array}$$

which follows directly from $\mathsf{fit}(\eta, \delta, u \cdot m \cdot v)$. For $q = p$, on the other hand, we must prove

$$\begin{aligned}\exists x, y:\ & x \cdot l_p \cdot y = u \cdot m \cdot v\ \wedge \\ & \forall e, f, g:\ \mathsf{wrp}_A(x, l_p \cdot y, g \cdot \langle e, f\rangle) \geq \mu\mathcal{P}\ ;\ \mathsf{wrp}_A(e, f, g)\ .\end{aligned}$$

Choosing $x = u \cdot \mathsf{goto}(l_b)$ and $y = m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b \cdot v$ makes the first conjunct true. The second conjunct is established by a fixpoint induction for $\mu\mathcal{P}$:

Admissibility is straightforward and the base case follows easily from the fact that $\perp\ ;\ \mathsf{wrp}_A(e, f, g) = \perp$. For the induction step assume that X is given such that for all e, f, g

$$\mathsf{wrp}_A(x, l_p \cdot y, g \cdot \langle e, f\rangle) \geq X\ ;\ \mathsf{wrp}_A(e, f, g)\ . \tag{3}$$

Now, $\mathsf{fit}(\eta[p \mapsto X], \delta[p \mapsto l_p], u \cdot m \cdot v)$ holds: for $q \neq p$ we can argue as above and for $q = p$ this follows from (3). Thus, by using the induction hypothesis of the structural induction applied to π_p we can calculate as follows for arbitrarily given e, f, g:

$$\begin{aligned}& \mathsf{wrp}_A(x, l_p \cdot y, g \cdot \langle e, f\rangle) \\ \geq\ & \quad \{\text{Law [Label-wrp] and unfolding of } y\} \\ & \mathsf{wrp}_A(x \cdot l_p, m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b \cdot v, g \cdot \langle e, f\rangle) \\ \geq\ & \quad \{\text{Induction hypothesis applied to } \pi_p\} \\ & \mathsf{wrp}_A^{\eta[p \mapsto X]}(\pi_p)\ ;\ \mathsf{wrp}_A(x \cdot l_p \cdot m_p, \mathsf{ret} \cdot l_b \cdot m_b \cdot v, g \cdot \langle e, f\rangle) \\ \geq\ & \quad \{\text{Definition of } \mathcal{P} \text{ and law [Ret-wrp]}\} \\ & \mathcal{P}(X)\ ;\ \mathsf{wrp}_A(e, f, g)\ .\end{aligned}$$

This completes the fixpoint induction. □

9 Conclusion

Two interweaved aspects motivated us to write the present paper. First of all we wanted to prove correct translation of a language with procedures to abstract assembler code; not just somehow or other but in an elegant and comprehensible manner. Algebraic calculations with predicate transformers turned out to be an adequate means for languages without procedures (see, e.g., [9]), so we decided to apply this technique in the extended scenario, too. The second stimulus is due to [10], where we proposed to employ wrp-semantics in compiler proofs. Real processors are always limited by their finite memory and a realistic notion of translation correctness must be prepared to deal with errors resulting from this limitation. We hope that the current paper demonstrates convincingly that wrp-based reasoning can cope with finite machines without burdening the verification.

The target language studied in this paper provides an adequate level of abstraction for further refinement down to actual binary machine code. The instructions may be considered as 'macros' for instructions of a more concrete assembler or machine language. Labels facilitate this, as they allow to describe destination of jumps independently from the length of code. An interesting aspect of our proof is that it shows how to handle the transition from tree-structured source programs to 'flat' target code. For this purpose we established a stronger claim that involves a universal quantification over syntactic target program contexts. This should be contrasted to the use of a tree-structured assembler language in [11] where translation correctness for a While-language without procedures is investigated. The proof in [11] does not immediately generalize to flat code.

Future work includes studying the relativized version of preservation of partial correctness ($\infty \in A$). In this case, semantics of recursive constructs is given by greatest rather than least fixpoints. As a consequence, fixpoint reasoning based on the fixpoints in the source language does not seem to work. We intend to use a fixpoint characterization of the target language's semantics instead. We also are working on concretizing from the abstract assembler language towards a realistic processor.

References

1. E. Börger and I. Durdanović. Correctness of compiling Occam to transputer code. *The Computer Journal*, 39(1), 1996.
2. L. M. Chirica and D. F. Martin. Towards compiler implementation correctness proofs. *ACM TOPLAS*, 8(2):185–214, April 1986.
3. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
4. J. D. Guttman, J. D. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.
5. C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorenson, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987.
6. C. A. R. Hoare, H. Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.
7. H. Langmaack. Software engineering for certification of systems: specification, implementation, and compiler correctness (in German). *Informationstechnik und Technische Informatik*, 39(3):41–47, 1997.
8. J. S. Moore. *Piton, A Mechanically Verified Assembly-Level Language*. Kluwer Academic Publishers, 1996.
9. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, LNCS 1283. Springer-Verlag, 1997.
10. M. Müller-Olm and A. Wolf. On excusable and inexcusable failures: towards an adequate notion of translation correctness. In *FM '99*, LNCS 1709, pp. 1107–1127. Springer-Verlag, 1999.
11. H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.
12. T. S. Norvell. Machine code programs are predicates too. In *6th Refinement Workshop*, Workshops in Computing. Springer-Verlag and British Computer Society, 1994.
13. E. Pofahl. Methods used for inspecting safety relevant software. In *High Integrity Programmable Electronics*, pages 13–14. Dagstuhl-Sem.-Rep. 107, 1995.