Axiomatic Semantics for Compiler Verification

Steven Schäfer Sigurd Schneider Gert Smolka
Saarland University, Germany
{schaefer,smolka}@ps.uni-saarland.de, sigurd.schneider@cs.uni-saarland.de

Abstract

Based on constructive type theory, we study two idealized imperative languages GC and IC and verify the correctness of a compiler from GC to IC. GC is a guarded command language with underspecified execution order defined with an axiomatic semantics. IC is a deterministic low-level language with linear sequential composition and lexically scoped gotos defined with a small-step semantics. We characterize IC with an axiomatic semantics and prove that the compiler from GC to IC preserves specifications. The axiomatic semantics we consider model total correctness and map programs to continuous predicate transformers. We define the axiomatic semantics of GC and IC with elementary inductive predicates and show that the predicate transformer described by a program can be obtained compositionally by recursion on the syntax of the program using a fixed point operator for loops and continuations. We also show that two IC programs are contextually equivalent if and only if their predicate transformers are equivalent.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Pre- and post-conditions; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational Semantics; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Mechanical Theorem Proving

Keywords Weakest preconditions, compiler correctness, formal verification

1. Introduction

In this paper we study a new form of axiomatic semantics for imperative languages designed for compiler verification. Based on constructive type theory, the semantics of a language is formulated as an inductive predicate providing total correctness judgements $\langle\sigma\rangle\, s\, \langle Q\rangle$. A judgement $\langle\sigma\rangle\, s\, \langle Q\rangle$ says that every execution of the program s on the initial state σ terminates with a state satisfying the postcondition Q. Given the basic judgements, Hoare-style total correctness statements

$$\langle P\rangle\, s\, \langle Q\rangle \;:=\; \forall\sigma.\, P\sigma \to \langle\sigma\rangle\, s\, \langle Q\rangle$$

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CPP 2016, January 20 - 22, 2016, Saint Petersburg, FL, USA.
Copyright © 2016 ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2854065.2854083

relating a program with a specification consisting of a pre- and a postcondition can be defined. We can also define a function

$$\mathrm{wp}\, s\, Q\, \sigma \;:=\; \langle\sigma\rangle\, s\, \langle Q\rangle$$

mapping a program s to a predicate transformer mapping a postcondition Q to the weakest precondition P such that $\langle P\rangle\, s\, \langle Q\rangle$. For the languages we consider, we show that programs are mapped to continuous predicate transformers. We also show that the wp function can be computed by structural recursion on programs using a fixed point operator for predicate transformers. We refer to the resulting denotational semantics as WP semantics.

The first language we study is a Dijkstra-style guarded command language [4] we call GC. GC is operationally underspecified in that it leaves open which guarded command is executed in case several guards are satisfied. This form of underspecification is easily dealt with by an axiomatic semantics, which speaks about predicates on states rather than single states. The rules of our axiomatic semantics for GC are reminiscent of the rules of a conventional big-step semantics $\langle\sigma\rangle\, s\, \langle\tau\rangle$, with the essential difference that we use postconditions rather than final states. This way the relevant behavior of an underspecified program can be described with a single judgement, which is not possible with a big-step semantics. The rules of our axiomatic semantics do not involve invariants or termination functions.

The second language we study is a low-level language with linear sequential composition and lexically scoped gotos we call IC. We see the declaration of a target for a goto as the definition of an argumentless, possibly recursive procedure to be used as a continuation. We define the semantics of IC with an operational small-step semantics realizing continuations with program substitution. In addition, we give an axiomatic semantics for IC and show that it agrees with the small-step semantics. The WP semantics complementing the axiomatic semantics turns out to be useful for proofs establishing properties of IC.

We give a compiler from GC to IC and verify its correctness. The compiler linearizes sequential compositions and realizes loops with continuations. Based on the axiomatic semantics of GC and IC, we express the correctness of the compiler as preservation of specifications. That is, given a GC program s, the compiler must yield an IC program t such that $\langle\sigma\rangle\, t\, \langle Q\rangle$ whenever $\langle\sigma\rangle\, s\, \langle Q\rangle$ (or, equivalently, $\langle P\rangle\, t\, \langle Q\rangle$ whenever $\langle P\rangle\, s\, \langle Q\rangle$).

For IC, we study program equivalence. As one would expect, program equivalence can be expressed with both the small-step and the axiomatic semantics. We show that program equivalence is the coarsest termination-preserving congruence on programs.

For the languages we consider, we show that the predicate transformers described by programs are monotonic. This provides for fixed points and enables a form of denotational semantics we call WP semantics. We show that the predicate transformers described by programs are continuous, which makes it possible to obtain the fixed points for loops and continuations with ω-iteration instead of intersection of all prefixed points. The WP semantics of IC

facilitates the verification of the compiler and the study of program equivalence.

We see the main contribution of the paper in the study of a new style of axiomatic semantics for imperative programming languages based on constructive type theory. The semantics is presented as an inductive predicate providing total correctness judgements $\langle\sigma\rangle\, s\, \langle Q\rangle$. Loops and continuations are accommodated with unfolding. In a second step, the axiomatic semantics is refined to a denotational semantics mapping programs to continuous predicate transformers (referred to as WP semantics). As one would expect, the denotation of a program can be obtained with structural recursion and ω-iterative fixed points.

Axiomatic semantics adapts well to languages with underspecified execution order (e.g., C or OCaml). Treating compiler correctness for such languages with a nondeterministic small-step semantics seems complex and tedious. At the example of GC we see that compiler correctness can be expressed elegantly and without overspecification using axiomatic semantics. The correctness statement for the compiler leaves the compiler unconstrained for states on which a program may diverge.

Another contribution of the paper is the definition and study of the low-level language IC. IC is an idealized version of an intermediate language IL used in previous work on compiler verification [15].

The entire development presented in this paper is formalized in Coq with Ssreflect. We assume functional extensionality so that we can realize program substitution for IC using the library Autosubst. The Coq development can be found online at the following URL: `http://www.ps.uni-saarland.de/extras/ascv`.

The paper is organized as follows. First we present GC and its axiomatic semantics. We then develop the WP semantics of GC. Next we present IC with its small-step semantics, axiomatic semantics, and WP semantics. We then verify a compiler from GC to IC. Following this, we study program equivalence in IC and show the WP semantics of GC and IC yield continuous predicate transformers. Finally, we discuss the Coq development and related work.

2. GC (Guarded Commands)

Dijkstra's language of guarded commands [4] is an imperative language with underspecified execution order. We introduce an abstract guarded command language GC whose states are taken from an abstract type Σ. Assignments are replaced with *actions*, which are abstract functions from states to states. *Guards* are modeled as boolean predicates on states. The syntax of GC is as follows:

$$\begin{array}{rll} \sigma, \tau : \Sigma & & \text{states} \\ a : \Sigma \to \Sigma & & \text{actions} \\ b : \Sigma \to \mathbb{B} & & \text{guards} \\ s, t ::= \mathsf{skip} \mid a \mid s; t \mid \mathsf{if}\ G \mid \mathsf{do}\ G & & \text{programs} \\ G ::= b_1 \Rightarrow s_1 \parallel \ldots \parallel b_n \Rightarrow s_n & (n \geq 0) & \end{array}$$

Conditionals **if** G and *loops* **do** G work on a *guarded command set* G, which is realized as a non-empty list of *guarded commands* $b \Rightarrow s$. The term set is justified since the order of the guarded commands does not matter semantically. We write $\varnothing$ for the empty guarded command set.

We describe the execution of conditionals and loops informally. The execution of a conditional **if** G selects a guarded command in G whose guard is satisfied and executes the program of the command. If no guard in G is satisfied, execution is aborted. The execution of a loop **do** G repeatedly executes guarded commands from G whose guard is satisfied. Execution of the loop terminates once all guards are dissatisfied. In case the guards of several commands are satisfied, any of the commands may be chosen for execution. We say that the execution order of GC is underspecified.

$$\frac{Q\sigma}{\langle\sigma\rangle\, \mathsf{skip}\, \langle Q\rangle} \qquad \frac{Q(a\sigma)}{\langle\sigma\rangle\, a\, \langle Q\rangle} \qquad \frac{\langle\sigma\rangle\, s\, \langle P\rangle \qquad \langle P\rangle\, t\, \langle Q\rangle}{\langle\sigma\rangle\, s; t\, \langle Q\rangle}$$

$$\frac{\widehat{G}\sigma \qquad \forall (b \Rightarrow s) \in G.\ b\sigma \to \langle\sigma\rangle\, s\, \langle Q\rangle}{\langle\sigma\rangle\, \mathsf{if}\ G\, \langle Q\rangle}$$

$$\frac{\langle\sigma\rangle\, \mathsf{if}\ G\, \langle P\rangle \qquad \langle P\rangle\, \mathsf{do}\ G\, \langle Q\rangle}{\langle\sigma\rangle\, \mathsf{do}\ G\, \langle Q\rangle} \qquad \frac{\neg(\widehat{G}\sigma) \qquad Q\sigma}{\langle\sigma\rangle\, \mathsf{do}\ G\, \langle Q\rangle}$$

$$\langle P\rangle\, s\, \langle Q\rangle := \forall\sigma.\ P\sigma \to \langle\sigma\rangle\, s\, \langle Q\rangle$$
$$\widehat{G} := \lambda\sigma.\ \exists (b \Rightarrow s) \in G.\ b\sigma$$

Figure 1. Axiomatic semantics of GC

Example 1 (Greatest Common Divisor). *The following program computes the gcd of two positive integers x and y.*

$$\textbf{\textit{do}}\ x > y \Rightarrow x := x - y \parallel y > x \Rightarrow y := y - x$$

We specify the axiomatic semantics of GC with an inductive predicate providing total correctness judgments $\langle\sigma\rangle\, s\, \langle Q\rangle$. Informally, a judgement $\langle\sigma\rangle\, s\, \langle Q\rangle$ says that every execution of the program s on the initial state σ terminates with a state satisfying the postcondition Q. Postconditions are unary predicates on states.

The definition of the inductive predicate $\langle\sigma\rangle\, s\, \langle Q\rangle$ is shown in Figure 1. Figure 1 also defines judgements of the form $\langle P\rangle\, s\, \langle Q\rangle$, which assert that every execution of the program s on a state satisfying the precondition P terminates with a state satisfying the postcondition Q.

The rule for sequential compositions $s; t$ uses a predicate P that serves as postcondition for s and as precondition for t. We refer to P as *interpolant*. The rule for loops also uses an interpolant. The structure of the rules for the axiomatic semantics is similar to the structure of the rules for a big-step semantics, where interpolants and postconditions appear as states.

The rule for a conditional **if** G is applicable only if the initial state σ satisfies at least one guard in G. Moreover, every command in G whose guard is satisfied by σ must satisfy the postcondition.

The axiomatic semantics is monotone in the postcondition.

Fact 2 (Monotonicity). *If $\langle\sigma\rangle\, s\, \langle P\rangle$ and $P \subseteq Q$, then $\langle\sigma\rangle\, s\, \langle Q\rangle$.*

As expected, the absurd post-condition cannot be derived:

Lemma 3. *If $\langle\sigma\rangle\, s\, \langle(\lambda\sigma.\bot)\rangle \equiv \lambda\sigma.\bot$.*

3. WP Semantics of GC

We define a function wp:

$$\mathrm{wp}\, s\, Q\, \sigma \ :=\ \langle\sigma\rangle\, s\, \langle Q\rangle$$

Given a program s, wp s is a function that maps postconditions to *weakest preconditions*. We call functions of this type *predicate transformers*.

Fact 4. wp s *is a monotonic predicate transformer.*

Proof. Follows with Fact 2. □

Since the type $\Sigma \to \mathbf{P}$ of predicates is a complete lattice, we know by the Knaster-Tarski theorem (see, e.g., Winskel [18]) that every monotonic predicate transformer has a least fixed point. This provides for a denotational semantics for GC where the predicate

transformer for a program is obtained by structural recursion on the program and the weakest precondition of a loop is obtained as a fixed point. We will refer to this form of denotational semantics as *WP semantics*.

For the following definition and facts we assume some arbitrary type X.

Definition 5. *We define a function* $\text{fix} : ((X \to \mathbf{P}) \to X \to \mathbf{P}) \to X \to \mathbf{P}$ *as follows:*

$$\text{fix}\ F\ x\ :=\ \forall P.\ FP \subseteq P \to Px$$

We refer to fix as *fixed point operator*. Note that $\text{fix}\ F$ is defined as the intersection of all prefixed points of F.

Fact 6 (Fixed Point Induction). *If* $FP \subseteq P$*, then* $\text{fix}\ F \subseteq P$.

Fact 7 (Least Fixed Point). *Let* F *be monotonic. Then* $\text{fix}\ F$ *is a least fixed point of* F.

We state a theorem providing the equivalences for the structurally recursive definition of the WP semantics for IC. The WP semantics is similar to Dijkstra's original semantics of GC [4].

Theorem 8. *The following equivalences hold for GC.*

$$\begin{aligned}
\text{wp}\ \textbf{\textit{skip}}\, Q &\equiv Q \\
\text{wp}\, a\, Q &\equiv \lambda\sigma.\, Q(a\sigma) \\
\text{wp}\, (s;t)\, Q &\equiv \text{wp}\, s\, (\text{wp}\, t\, Q) \\
\text{wp}\, (\textbf{\textit{if}}\ G)\, Q &\equiv \lambda\sigma.\, \widehat{G}\sigma \wedge \forall (b \Rightarrow s) \in G.\, b\sigma \to \text{wp}\, s\, Q\, \sigma \\
\text{wp}\, (\textbf{\textit{do}}\ G)\, Q &\equiv \text{fix}\ (\lambda P\sigma.\, \text{if}\ \widehat{G}\sigma\ \text{then}\ \text{wp}\, (\textbf{\textit{if}}\ G)\, P\, \sigma\ \text{else}\ Q\sigma)
\end{aligned}$$

Proof. Define a recursive function WP based on the claimed equivalences and show that it is monotonic in the postcondition. For the claim it suffices to show that $\langle\sigma\rangle\, s\, \langle Q\rangle$ and $\text{WP}\, s\, Q\, \sigma$ are equivalent. One direction follows by induction on $\langle\sigma\rangle\, s\, \langle Q\rangle$. The other direction follows by induction on s and in the case of a loop by fixed point induction. □

Realizing the WP semantics with Coq's structural recursion is not difficult. See the accompanying Coq development for details.

4. IC (Imperative Continuations)

The second language we study is a low-level language with linear sequential composition and lexically scoped gotos we call IC. We see the declaration of a target for a goto as the definition of an argumentless, possibly recursive procedure to be used as a continuation. IC is an idealized version of an intermediate language IL used in previous work on compiler verification [15]. We define the syntax of IC as follows:

$$\begin{aligned}
&f, g\ :\ \mathcal{L} && \text{labels} \\
&s, t\ ::=\ a; s \mid \textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ t \mid \textbf{def}\ f = s\ \textbf{in}\ t \mid f && \text{programs}
\end{aligned}$$

Actions a and guards b are as in GC. Figure 2 defines an operational small-step semantics for IC. The semantics is formulated as an inductive predicate providing judgements $(s, \sigma) \rhd (t, \tau)$ describing single execution steps.

Things are arranged such that terminal configurations are pairs (f, σ) consisting of a label and a state. We see such configurations as calls to external continuations.

Local continuations are introduced with programs of the form $\textbf{def}\ f = s\ \textbf{in}\ t$. The label f acts as a local variable in s and t. Informally, execution of a program $\textbf{def}\ f = s\ \textbf{in}\ t$ binds the label f to the program s and proceeds with the execution of the program t. The small-step semantics realizes this idea by reducing the program $\textbf{def}\ f = s\ \textbf{in}\ t$ to its unfolding $t^f_{\textbf{def}\ f=s\ \textbf{in}\ s}$. The unfolding provides for recursive and lexically scoped continuations.

$$(a; s, \sigma) \rhd (s, a\sigma) \qquad (\textbf{def}\ f = s\ \textbf{in}\ t, \sigma) \rhd (t^f_{\textbf{def}\ f=s\ \textbf{in}\ s}, \sigma)$$

$$\frac{b\sigma}{(\textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ t, \sigma) \rhd (s, \sigma)}$$

$$\frac{\neg b\sigma}{(\textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ t, \sigma) \rhd (t, \sigma)}$$

t^f_u is obtained from t by capture-free substitution of f with u

Figure 2. Small-step semantics of IC

$$\frac{\langle a\sigma\rangle\, s\, \langle\mathcal{Q}\rangle}{\langle\sigma\rangle\, a; s\, \langle\mathcal{Q}\rangle} \qquad \frac{\mathcal{Q}f\sigma}{\langle\sigma\rangle\, f\, \langle\mathcal{Q}\rangle}$$

$$\frac{b\sigma \qquad \langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle}{\langle\sigma\rangle\, \textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ t\, \langle\mathcal{Q}\rangle} \qquad \frac{\neg b\sigma \qquad \langle\sigma\rangle\, t\, \langle\mathcal{Q}\rangle}{\langle\sigma\rangle\, \textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ t\, \langle\mathcal{Q}\rangle}$$

$$\frac{\langle\sigma\rangle\, t\, \langle\mathcal{Q}[f \mapsto P]\rangle \qquad \langle P\rangle\, \textbf{def}\ f = s\ \textbf{in}\ s\, \langle\mathcal{Q}\rangle}{\langle\sigma\rangle\, \textbf{def}\ f = s\ \textbf{in}\ t\, \langle\mathcal{Q}\rangle}$$

$$\langle P\rangle\, s\, \langle Q\rangle := \forall\sigma.\, P\sigma \to \langle\sigma\rangle\, s\, \langle Q\rangle$$
$$\mathcal{Q}[f \mapsto P] := \lambda g.\, \text{if}\ g = f\ \text{then}\ P\ \text{else}\ \mathcal{Q}g$$

Figure 3. Axiomatic semantics of IC

We write $(s, \sigma) \rhd^* (t, \tau)$ for multiple steps of execution. Formally, the predicate $\rhd^*$ is obtained as the reflexive transitive closure of $\rhd$.

Example 9 (Greatest Common Divisor in IC). *The following program computes the greatest common divisor of* x *and* y *and terminates with a call to the external continuation* ret.

$$\begin{aligned}
&\textbf{\textit{def}}\ f = \ \textbf{\textit{if}}\ x > y\ \textbf{\textit{then}}\ x := x - y; f \\
&\qquad\qquad \textbf{\textit{else if}}\ y > x\ \textbf{\textit{then}}\ y := y - x; f\ \textbf{\textit{else}}\ \text{ret} \\
&\textbf{\textit{in}}\ f
\end{aligned}$$

4.1 Axiomatic Semantics of IC

We specify the axiomatic semantics of IC with an inductive predicate providing total correctness judgments $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle$. The judgements employ *generalized postconditions* $\mathcal{Q}$, which are predicates $\mathcal{L} \to \Sigma \to \mathbf{P}$ on labels and states. We see $\mathcal{Q}f$ as a precondition for the external continuation f that must be satisfied if f is called. Informally, a judgment $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle$ says that the execution of the program s on the initial state σ terminates with a call to an external continuation f such that $\mathcal{Q}f$ holds for the final state.

The definition of the inductive predicate $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle$ is shown in Figure 3. The most interesting rule is the rule for the definition of local continuations, which employs an interpolant serving as precondition for the defined continuation.

We will write $[f \mapsto P]$ or just $f \mapsto P$ for the generalized postcondition $(\lambda g\sigma.\, \bot)[f \mapsto P]$. Note that this postcondition requires a program to terminate with a call to the external continuation f.

Fact 10 (Monotonicity). *If* $\langle\sigma\rangle\, s\, \langle\mathcal{P}\rangle$ *and* $\mathcal{P} \subseteq \mathcal{Q}$*, then* $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle$.

4.2 WP Semantics of IC

We define *weakest preconditions* for IC as follows:

$$\text{wp}\, s\, \mathcal{Q}\, \sigma\ :=\ \langle\sigma\rangle\, s\, \langle Q\rangle$$

By Fact 10 we know that the function wp maps programs to monotonic predicate transformers. We state a theorem providing the equivalences for the structurally recursive definition of the WP semantics for IC.

Theorem 11. *The following equivalences hold for IC.*

$$\begin{aligned}
\mathrm{wp}\, f\, \mathcal{Q} &\equiv \mathcal{Q} f \\
\mathrm{wp}\,(a; s)\, \mathcal{Q} &\equiv \lambda\sigma.\ \mathrm{wp}\, s\, \mathcal{Q}\, (a\sigma) \\
\mathrm{wp}\,(\textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ s\ \textbf{\textit{else}}\ t)\, \mathcal{Q} &\equiv \lambda\sigma.\ \text{if } b\sigma \text{ then } \mathrm{wp}\, s\, \mathcal{Q}\, \sigma \text{ else } \mathrm{wp}\, t\, \mathcal{Q}\, \sigma \\
\mathrm{wp}\,(\textbf{\textit{def}}\ f = s\ \textbf{\textit{in}}\ t)\, \mathcal{Q} &\equiv \mathrm{wp}\, t\, \mathcal{Q}[f \mapsto \mathrm{fix}\, F] \\
\text{where } F P &:= \mathrm{wp}\, s\, \mathcal{Q}[f \mapsto P]
\end{aligned}$$

Proof. Similar to the proof of Theorem 8. □

5. Agreement of Operational and Axiomatic Semantics of IC

In this section, we show that the axiomatic semantics of IC characterizes the small-step operational semantics (see Theorem 18):

$$\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle \leftrightarrow \exists\tau f.\ (s,\sigma) \rhd^* (f,\tau) \wedge \mathcal{Q} f \tau$$

The small-step semantics for IC uses syntactic substitutions. In the axiomatic semantics we can express substitution using the post-condition.

Lemma 12. $\mathrm{wp}\, t_s^f\, \mathcal{Q} \equiv \mathrm{wp}\, t\, \mathcal{Q}[f \mapsto \mathrm{wp}(s, \mathcal{Q})]$

Proof. By induction on s using the WP semantics of IC. □

Lemma 13. $\mathrm{wp}(\textbf{\textit{def}}\ f = s\ \textbf{\textit{in}}\ t)\, \mathcal{Q} \equiv \mathrm{wp}\, t\, \mathcal{Q}[f \mapsto \mathrm{wp}(s, \mathcal{Q})]$ *if* f *not free in* s.

Proof. We have $\mathrm{wp}\, s\, \mathcal{Q}[f \mapsto P] \equiv \mathrm{wp}\, s\, \mathcal{Q}$ since f is not free in s. The result follows by monotonicity of wp. □

The axiomatic semantics of IC is compatible with small-step reduction.

Lemma 14. *If* $(s,\sigma) \rhd (t,\tau)$*, then* $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle \leftrightarrow \langle\tau\rangle\, t\, \langle\mathcal{Q}\rangle$.

Proof. By case analysis on the step relation using the WP semantics of IC. For local definitions we use Lemma 12. All other cases are immediate. □

Lemma 15. *If* $(s,\sigma) \rhd^* (f,\tau)$*, then* $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle \leftrightarrow \mathcal{Q} f \tau$.

We define termination as follows:

$$(s,\sigma)\Downarrow \ :=\ \exists\tau f.\ (s,\sigma) \rhd^* (f,\tau)$$

In order to show that $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle$ implies termination of s, we generalize in a way reminiscent of logical relations. According to Lemma 12, a post-condition can be read as a semantic substitution. Conversely, if $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle$ holds, then s terminates under all substitutions which are compatible with $\mathcal{Q}$.

A substitution is a function θ mapping continuations to programs. We write θs for the program s with all free variables substituted according to θ.

Lemma 16. *Let* θ *be a function mapping continuations to programs. If* $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle$ *and* $\forall f\tau.\ \mathcal{Q} f \tau \to (\theta f, \sigma)\Downarrow$ *then* $(\theta s, \sigma)\Downarrow$.

Proof. By induction on the derivation of $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle$. □

Lemma 17. $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle \to (s,\sigma)\Downarrow$

We can now show that the axiomatic semantics of IC coincides with the small-step semantics.

Theorem 18. $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle \leftrightarrow \exists\tau f.\ (s,\sigma) \rhd^* (f,\tau) \wedge \mathcal{Q} f \tau$

$$\begin{aligned}
\mathcal{C}\, s &:= \mathcal{T}\, \mathsf{ret}\, s \\
\mathcal{T}\, u\, \mathsf{skip} &:= u \\
\mathcal{T}\, u\, a &:= a; u \\
\mathcal{T}\, u\, (s;t) &:= \mathcal{T}\, (\mathcal{T}\, u\, t)\, s \\
\mathcal{T}\, u\, (\mathsf{if}\ \varnothing) &:= \mathsf{ret} \\
\mathcal{T}\, u\, (\mathsf{if}\ b \Rightarrow s \parallel G) &:= \mathcal{F}\, (\mathcal{T} u)\, (\mathcal{T}\, u\, s)\, G \\
\mathcal{T}\, u\, (\mathsf{do}\ G) &:= \mathsf{def}\ f = \mathcal{F}\, (\mathcal{T} f)\, u\, G\ \mathsf{in}\ f \\
&\qquad \text{(where } f \text{ is fresh)} \\
\mathcal{F}\, C\, v\, \varnothing &:= v \\
\mathcal{F}\, C\, v\, (b \Rightarrow s \parallel G) &:= \mathsf{if}\ b\ \mathsf{then}\ C\, s\ \mathsf{else}\ \mathcal{F}\, C\, v\, G
\end{aligned}$$

Figure 4. Compiler from GC to IC

Proof. In the direction "$\to$", we use Lemma 17 to obtain (f,τ) such that $(s,\sigma) \rhd^* (f,\tau)$ from $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle$. By Lemma 15 we obtain $\mathcal{Q} f \tau$. The direction "$\leftarrow$" is a special case of Lemma 15. □

Corollary 19. $\langle\sigma\rangle\, s\, \langle f \mapsto (= \tau)\rangle \leftrightarrow (s,\sigma) \rhd^* (f,\tau)$

Corollary 20. $(s,\sigma)\Downarrow \leftrightarrow \langle\sigma\rangle\, s\, \langle\lambda f.\top\rangle$

We obtain that IC is deterministic in the following sense.

Theorem 21. $\langle\sigma\rangle\, s\, \langle\mathcal{Q}\rangle \leftrightarrow \exists\tau f.\ \langle\sigma\rangle\, s\, \langle f \mapsto (= \tau)\rangle \wedge \mathcal{Q} f \tau$

Proof. By Theorem 18 and Corollary 19. □

Theorem 22 (Distributivity). $\mathrm{wp}\, s\, (\mathcal{P} \cap \mathcal{Q}) \equiv \mathrm{wp}\, s\, \mathcal{P} \cap \mathrm{wp}\, s\, \mathcal{Q}$

Proof. The direction "$\subseteq$" follows from Fact 10. The direction "$\supseteq$" follows from Theorem 18 and Lemma 15. □

6. Compiling GC to IC

In this section, we verify a compiler $\mathcal{C}$ from GC to IC. We distinguish an external label ret. The compiler arranges the target program such that a call to ret indicates successful termination. The compiler correctness statement we prove is preservation of specifications:

$$\langle\sigma\rangle\, s\, \langle Q\rangle \to \langle\sigma\rangle\, \mathcal{C} s\, \langle \mathsf{ret} \mapsto Q\rangle$$

Together with Theorem 18, this yields the following correctness statement, which connects the axiomatic semantics of GC to the small-step semantics of IC:

$$\langle\sigma\rangle\, s\, \langle Q\rangle \to \exists\tau.\ (\mathcal{C} s, \sigma) \rhd^* (\mathsf{ret}, \tau) \wedge Q\tau$$

The compiler is defined in Figure 4. The main transformations are the sequentialization of guarded command sets and the translation of loops to recursive functions. The compiler exploits underspecification of GC and the fact that the compiled program only needs to be correct for states σ under which, intuitively speaking, s always terminates.

6.1 Sequencing Guarded Command Sets

The compiler realizes guarded command sets with nested IC conditionals.

Example 23 (Translation of Conditionals). *The program*

$$(\textbf{\textit{if}}\ b_1 \Rightarrow s_1 \parallel b_2 \Rightarrow s_2 \parallel b_3 \Rightarrow s_3); u$$

is translated to

$$\textbf{\textit{if}}\ b_2\ \textbf{\textit{then}}\ s_2; u\ \textbf{\textit{else if}}\ b_3\ \textbf{\textit{then}}\ s_3; u\ \textbf{\textit{else}}\ s_1; u$$

Underspecification in GC allows to test the guards b_2, b_3 in any order. The guard b_1 does not need to be tested, because if the GC programs terminates, then one of the guards b_1, b_2, b_3 is satisfied.

The function $\mathcal{F}$ maps a guarded command set to nested IC conditionals. $\mathcal{F}$ takes three arguments: A function C mapping GC programs to IC programs, an IC program v, and the guarded command set to be translated. $\mathcal{F}$ translates each guarded command to the corresponding conditional, and places the program u in the final else case. The program $\mathcal{F}\,C\,v\,(b_1 \Rightarrow s_1 \parallel \ldots \parallel b_n \Rightarrow s_n)$ tests the guards b_i in order, and continues execution with the first program s_i for which b_i is satisfied. If no guard b_i is satisfied, execution continues with v.

Lemma 24. *Let G be a guarded command set, C a function mapping GC into IC, and v an IC program. The following inference rules are admissible for the semantics of IC.*

$$\frac{\widehat{G}\sigma \qquad \forall (b \Rightarrow s) \in G.\, b\sigma \to \langle\sigma\rangle\, C\, s\, \langle Q\rangle}{\langle\sigma\rangle\, \mathcal{F}\, C\, v\, G\, \langle Q\rangle}$$

$$\frac{\neg\widehat{G}\sigma \qquad \langle\sigma\rangle\, v\, \langle\mathcal{Q}\rangle}{\langle\sigma\rangle\, \mathcal{F}\, C\, v\, G\, \langle\mathcal{Q}\rangle}$$

Proof. By induction on G. □

The compiler uses $\mathcal{F}$ to translate conditionals $\textsf{if}\ b \Rightarrow s \parallel G$ to nested conditionals in IC by placing s in the final else case. The compiler is allowed to translate the empty conditional $\textsf{if}\ \varnothing$ to any program, because, intuitively speaking, the empty conditional is stuck.

6.2 Linearizing GC

In addition to sequencing guarded command sets, the compiler translates loops to tail-recursive continuations. We call this translation *linearization*, because full sequentialization $s; t$ as it occurs in GC is translated to the linear sequenzialization $a; s$ as it is available in IC.

Example 25 (Translation of Loops). *The program*

$$(\textbf{\textit{do}}\ b_1 \Rightarrow s_1 \parallel b_2 \Rightarrow s_2); u$$

is translated to

$$\textbf{\textit{def}}\ f = \textbf{\textit{if}}\ b_1\ \textbf{\textit{then}}\ s_1; f\ \textbf{\textit{else if}}\ b_2\ \textbf{\textit{then}}\ s_2; f\ \textbf{\textit{else}}\ u\ \textbf{\textit{in}}\ f$$

Underspecification in GC allows the guards b_1, b_2 to be tested in any order. The final else case is reached if none of the guards b_1, b_2 is satisfied. In this case, the loop terminates and execution continues with u.

Linearization is implemented by a function $\mathcal{T}\,u\,s$, which takes a GC program s and an IC program u to be used as a continuation. Intuitively, the program $\mathcal{T}\,u\,s$ is the sequentialization of s and u: First s is executed, and then execution continues with u. The correctness statement for $\mathcal{T}$ reflects the sequentialization aspect: Given a specification $\langle\sigma\rangle\, s\, \langle P\rangle$ for the source program, and a specification $\langle P\rangle\, u\, \langle\mathcal{Q}\rangle$ for the continuation, the target satisfies the specification $\langle\sigma\rangle\, \mathcal{T}\, u\, s\, \langle\mathcal{Q}\rangle$. P acts as interpolant, similarly as in the rules for sequentialization in the axiomatic semantics of GC.

Theorem 26. *Let $\langle\sigma\rangle\, s\, \langle P\rangle$ and $\langle P\rangle\, u\, \langle\mathcal{Q}\rangle$, then $\langle\sigma\rangle\, \mathcal{T}\, u\, s\, \langle\mathcal{Q}\rangle$.*

Proof. By induction on the derivation of $\langle\sigma\rangle\, s\, \langle P\rangle$. We use WP semantics of IC and consider the cases for guarded commands and loops.

- In the case of a conditional, note that the semantics of GC ensures that the guarded command set cannot be empty. For the compilation of $\mathcal{T}\,u\,(\textsf{if}\ b \Rightarrow s \parallel G)$ we have to show $\langle\sigma\rangle\, \mathcal{F}\,(\mathcal{T} u)\,(\mathcal{T}\, u\, s)\, G\, \langle P\rangle$. We distinguish two cases. If $\widehat{G}\sigma$ holds, we apply the corresponding rule from Lemma 24. The second premise of the rule follows from the inductive hypothesis. If $\widehat{G}\sigma$ does not hold, then no guard in G is satisfied. The semantics of GC ensures that $b\sigma$ holds. We apply the corresponding rule from Lemma 24, and discharge its second premise with the inductive hypothesis for s.
- In the case of $\mathcal{T}\,u\,(\textsf{do}\ G)$, we have

  $$\langle\sigma\rangle\, \mathcal{T}\, u\, (\textsf{do}\ G)\, \langle\mathcal{Q}\rangle = \langle\sigma\rangle\, \mathcal{F}\,(\mathcal{T} f)\, u\, G\, \langle\mathcal{Q}'\rangle$$
  $$\text{where } \mathcal{Q}' := \mathcal{Q}[f \mapsto \mathrm{wp}(\mathcal{T}\, u\, (\textsf{do}\ G))\, \mathcal{Q}]$$

  Depending on $\widehat{G}(\sigma)$, we apply the corresponding rule from Lemma 24. If $\widehat{G}(\sigma)$ holds, we can conclude with the induction hypothesis. If $\neg\widehat{G}(\sigma)$ holds, we use the monotonicity property of the WP semantics to show $\langle\sigma\rangle\, u\, \langle\mathcal{Q}\rangle \subseteq \langle\sigma\rangle\, u\, \langle\mathcal{Q}'\rangle$. □

From Theorem 26 we obtain the correctness of the complete compiler.

Corollary 27. $\langle\sigma\rangle\, s\, \langle Q\rangle \to \langle\sigma\rangle\, \mathcal{C}\, s\, \langle \mathsf{ret} \mapsto Q\rangle$

6.3 Avoiding Exponential Blowup

We now have a formally verified compiler. However, the translation of guarded command sets may cause exponential blowup. Consider the duplication of the continuation u in Example 23.

We avoid duplicating u with an abstraction in the compilation of conditionals.

$$\mathcal{T}\,u\,(\textsf{if}\ b \Rightarrow s \parallel G) =$$
$$\quad \textsf{def}\ f = u\ \textsf{in}\ \mathcal{F}\,(\mathcal{T}\, f)\,(\mathcal{T}\, f\, s)\, G \qquad (f \text{ fresh})$$

The proof of Theorem 26 now requires Lemma 13 to be applied in the case for conditionals, but remains otherwise unchanged.

$\mathcal{T}$ now avoids exponential blowup, but may introduce more continuations than necessary. In particular, we do not want a new definition if u is already a call to a continuation. We define an auxiliary function **let**, which introduces definitions only if necessary.

$$\textsf{let}\ f = s\ \textsf{in}\ t = \begin{cases} \textsf{def}\ g = s\ \textsf{in}\ t^f_g,\ (g \text{ fresh}) & \text{if } |s| > 1 \\ t^f_s & \text{otherwise} \end{cases}$$

Semantically, **let** behaves like a non-recursive definition.

Lemma 28. $\mathrm{wp}(\textbf{\textit{let}}\ f = s\ \textbf{\textit{in}}\ t)\, \mathcal{Q} = \mathrm{wp}\, t\, \mathcal{Q}[f \mapsto \mathrm{wp}(s, \mathcal{Q})]$

Proof. By Lemma 12 and Lemma 13. □

We modify the compiler to handle conditionals according to the following equation. The modification still avoids exponential blowup and only introduces new labels if necessary.

$$\mathcal{T}\,u\,(\textsf{if}\ b \Rightarrow s \parallel G) =$$
$$\quad \textsf{let}\ f = u\ \textsf{in}\ \mathcal{F}\,(\mathcal{T} f)\,(\mathcal{T}\, f\, s)\, G \qquad (f \text{ fresh})$$

The proof of Theorem 26 now relies on Lemma 28, instead of Lemma 13, but remains otherwise unchanged.

7. IC Program Equivalence

We consider program equivalence on IC. As usual, program equivalence is obtained from program preorder:

$$s \cong t \leftrightarrow s \preceq t \wedge t \preceq s$$

The different semantic characterizations of IC suggest different notions of program preorder. From an operational semantics we can define *contextual approximation* [11] as the most-permissive notion of program preorder. A context C is a program with a hole, and

contextual approximation $s \preceq t$ demands that no context in which s terminates can distinguish s from t.

$$s \preceq t \;:=\; \forall C\sigma.\, (C[s],\sigma)\Downarrow \to (C[t],\sigma)\Downarrow$$

Preservation of specifications provides an alternative preorder.

$$s \lesssim t \;:=\; \forall \mathcal{Q}.\ \mathrm{wp}\, s\, \mathcal{Q} \subseteq \mathrm{wp}\, t\, \mathcal{Q}$$

We show that contextual approximation and specification preservation coincide under the assumption that certain guards are available.

Formally, contexts are given by grammar

$$\begin{aligned} C \;::=\;& a; C \mid \mathsf{if}\ b\ \mathsf{then}\ C\ \mathsf{else}\ t \mid \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ C \mid \\ & \mathsf{def}\ f = C\ \mathsf{in}\ s \mid \mathsf{def}\ f = s\ \mathsf{in}\ C \mid [\cdot] \end{aligned}$$

Instantiation of contexts is written as $C[s]$ and denotes the program consisting of the context C with the hole replaced by s.

For a relation R on IC programs we define:

$$\begin{aligned} R \textit{ compatible} &:= \forall stC.\ R\,s\,t \to R\,C[s]\,C[t] \\ R \textit{ consistent} &:= \forall st\sigma.\ R\,s\,t \to (s,\sigma)\Downarrow \to (t,\sigma)\Downarrow \end{aligned}$$

An *approximation* is a compatible and consistent relation.

Fact 29. *Contextual approximation is the coarsest approximation.*

We show that specification preservation is an approximation.

Lemma 30. *Specification preservation is consistent.*

Proof. By Corollary 20 termination is a specification. □

Lemma 31. *Specification preservation is compatible.*

Proof. We show $s \lesssim t \to C[s] \lesssim C[t]$ by induction on C. The only case which does not follow immediately from the definition is the case for contexts of the form $\mathsf{def}\ f = C\ \mathsf{in}\ t$. By the inductive hypothesis we may assume that $s \lesssim s'$ and it remains to show that $\mathsf{def}\ f = s\ \mathsf{in}\ t \lesssim \mathsf{def}\ f = s'\ \mathsf{in}\ t$. Unfolding the definition of wp we obtain

$$\begin{aligned} \mathrm{wp}(\mathsf{def}\ f = s\ \mathsf{in}\ t)\ \mathcal{Q} &\equiv \mathrm{wp}\, t\, \mathcal{Q}[f \mapsto \mathrm{fix}(Fs)] \\ &\quad \text{where } FuP = \mathrm{wp}\, u\, \mathcal{Q}[f \mapsto P] \end{aligned}$$

By monotonicity of wp it suffices to show that $\mathrm{fix}(Fs) \subseteq \mathrm{fix}(Fs')$. This follows by induction:

$$\begin{aligned} Fs\,(\mathrm{fix}(Fs')) &= \mathrm{wp}\, s\, \mathcal{Q}[f \mapsto \mathrm{fix}(Fs')] \\ &\subseteq \mathrm{wp}\, s'\, \mathcal{Q}[f \mapsto \mathrm{fix}(Fs')] \\ &\equiv \mathrm{fix}(Fs') \end{aligned}$$ □

Lemma 32. $s \lesssim t \to s \preceq t$

Proof. Follows with Fact 29 from Lemma 30 and 31. □

The proof of the converse direction requires enough guards to distinguish individual states. Guards are boolean predicates on state. There are boolean predicates that distinguish states if the type of states Σ has decidable equality. This is a rather mild assumption, since states are typically finite maps from locations to values.

Lemma 33. *If Σ has decidable equality, then $s \preceq t \to s \lesssim t$.*

Proof. By Theorem 21 it suffices to show that singleton specifications are preserved. Assume that the judgement $\langle\sigma\rangle\, s\, \langle f \mapsto (= \tau)\rangle$ holds. Since specifications are total, s terminates under σ. By consistence, we can assume that t terminates and in particular that the judgement $\langle\sigma\rangle\, t\, \langle f' \mapsto (= \tau')\rangle$ holds for some f', τ'. It suffices to show that t terminates with the same continuation and state as s, i.e., that $f = f'$ and $\tau = \tau'$. We can show both equalities at the same time by crafting a context in which t diverges. Consider the context

$$C = \mathsf{def}\ f' = \big(\mathsf{if}\ (= \tau')\ \mathsf{then}\ f'\ \mathsf{else}\ \mathsf{ret}\big)\ \mathsf{in}\ [\cdot]$$

For an arbitrary program u in context C we use Lemma 15 to show

$$\begin{aligned} &\langle\sigma\rangle\, C[u]\, \langle\mathcal{Q}\rangle \to \langle\sigma\rangle\, u\, \langle\mathcal{Q}[f' \mapsto (\neq \tau')]\rangle \\ &\qquad \to ((u,\sigma) \rhd^* (f'',\tau'') \to f' \neq f'' \vee \tau' \neq \tau'') \end{aligned}$$

We decide whether $f' = f$ and $\tau' = \tau$. If this is not the case, then s terminates in C, and by contextual approximation the same is true for t. From the reasoning above, we obtain $f' \neq f' \vee \tau' \neq \tau'$, a contradiction. □

Given the equivalence between the operational and axiomatic semantics we obtain several equivalent presentations of contextual approximations for IC.

Theorem 34. *If Σ has decidable equality the following statements are equivalent for all IC programs s, t:*

(1) $s \preceq t$
(2) $s \lesssim t$
(3) $\forall P\mathcal{Q}.\ \langle P\rangle\, s\, \langle\mathcal{Q}\rangle \to \langle P\rangle\, t\, \langle\mathcal{Q}\rangle$
(4) $\forall\sigma f\tau.\ (s,\sigma) \rhd^* (f,\tau) \to (t,\sigma) \rhd^* (f,\tau)$

Proof. The equivalence of (1) and (2) follows from Lemma 32 and Lemma 33. By definition $\langle P\rangle\, s\, \langle\mathcal{Q}\rangle = P \subseteq \mathrm{wp}\, s\, \mathcal{Q}$ and thus (2) is equivalent to (3). By Corollary 19, proposition (4) is equivalent to preservation of singleton specifications, which, by Theorem 21, is equivalent to (2). □

8. Continuity

The WP semantics of GC and IC resembles a denotational semantics. In denotational semantics, fixed points are computed as least upper bounds over finite approximations. In this section, we show that the WP semantics assigns continuous predicate transformers to GC and IC programs. The result allows us to compute fixed points by ω-iteration. Formally, we obtain this result as a special case of a fixed point theorem [18]. In particular, we show that syntactic fixed points are least upper bounds with respect to preservation of specifications. This yields a complete syntactic proof method for showing inequalities involving recursive definitions.

A predicate transformer F is continuous if it distributes over suprema of ω-chains.

Definition 35 (Chain). *A family $(D_n)_{n\in\mathbb{N}}$ is a chain if $D_n \leq D_{n+1}$ for all $n \in \mathbb{N}$.*

Definition 36 (Continuous). *A predicate transformer F is continous, if for all chains $(D_n)_{n\in\mathbb{N}}$ we have*

$$F(\bigcup_{n\in\mathbb{N}} D_n) \equiv \bigcup_{n\in\mathbb{N}} F\, D_n$$

Fact 37. *Continuity implies monotonicity.*

8.1 Continuity for IC

We show that the WP semantics for IC yields continuous predicate transformers. The proof follows from the fact that the language is deterministic and does not need the assumption that the family D_n forms a chain. In this sense, continuity is weaker than determinism.

Lemma 38. *Let D be a class of environments and s be an IC program. Then* wp s *is continuous:*

$$\mathrm{wp}\, s\, (\bigcup_{\mathcal{Q}\in D} \mathcal{Q}) \;\equiv\; \bigcup_{\mathcal{Q}\in D} \mathrm{wp}\, s\, \mathcal{Q}$$

Proof. By Theorem 21. □

The restriction to chains is necessary in non-deterministic languages. For GC, wp does not distribute over binary unions, i.e.,

$\mathrm{wp}\, s\, (Q_1 \cup Q_2) \not\subseteq \mathrm{wp}\, s\, Q_1 \cup \mathrm{wp}\, s\, Q_2$, since the post-condition must include all final states of s.

In Section 3 we defined a fixed-point operator in terms of an intersection over all prefixed points. A general fixed-point theorem [18] allows us to characterize fixed points of continuous predicates in terms of ω-iteration.

Fact 39. *Let $F : (X \to \mathbf{P}) \to X \to \mathbf{P}$ be continuous. Then*

$$\mathrm{fix}\, F \equiv \bigcup_{n\in\mathbb{N}} F^n \bot$$

where F^n denotes the n-fold application of F.

We use a chain of approximations to $\mathrm{wp}(\textsf{def}\ f = s\ \textsf{in}\ s, \mathcal{Q})$ to give semantics to recursive definitions in IC via iteration.

Lemma 40. *For fixed f, s, and $\mathcal{Q}$ define*

$$P_0 := \bot$$
$$P_{n+1} := \mathrm{wp}(s, \mathcal{Q}[f \mapsto P_n])$$

We have

$$\mathrm{wp}(\textbf{\textit{def}}\ f = s\ \textbf{\textit{in}}\ t)\, \mathcal{Q} = \mathrm{wp}\, t\, \mathcal{Q}[f \mapsto \bigcup_{n\in\mathbb{N}} P_n]$$

Proof. By Fact 39 and Lemma 38. □

The chain of approximations to $\mathrm{wp}(\textsf{def}\ f = s\ \textsf{in}\ s)\, \mathcal{Q}$ can be built by iteration of syntactic substitution, and consequently the WP semantics of recursive definitions can be characterized syntactically.

Lemma 41. *For fixed f, s, and $\mathcal{Q}$ define*

$$s^0 = \textbf{\textit{def}}\ f = f\ \textbf{\textit{in}}\ f$$
$$s^{n+1} = s^f_{s^n}$$

Define P_n as in Lemma 40. We have

$$\mathrm{wp}\, s^n\, \mathcal{Q} \equiv P_n$$

Proof. By induction on n with Lemma 12. □

Theorem 42. $\mathrm{wp}(\textbf{\textit{def}}\ f = s\ \textbf{\textit{in}}\ t)\, \mathcal{Q} \equiv \bigcup_{n\in\mathbb{N}} \mathrm{wp}\, t^f_{s^n}\, \mathcal{Q}$

Proof. By Lemma 40, 38, 41, and 12:

$$\begin{aligned}\mathrm{wp}(\textsf{def}\ f = s\ \textsf{in}\ t)\, \mathcal{Q} &\equiv \mathrm{wp}\, t\, \mathcal{Q}[f \mapsto \bigcup_{n\in\mathbb{N}} P_n] \\ &\equiv \bigcup_{n\in\mathbb{N}} \mathrm{wp}\, t\, \mathcal{Q}[f \mapsto P_n] \\ &\equiv \bigcup_{n\in\mathbb{N}} \mathrm{wp}\, t\, \mathcal{Q}[f \mapsto \mathrm{wp}\, s^n\, \mathcal{Q}] \\ &\equiv \bigcup_{n\in\mathbb{N}} \mathrm{wp}\, t^f_{s^n}\, \mathcal{Q}\end{aligned}$$ □

As a corollary of Theorem 42, we obtain a proof rule for approximation in the case of local definitions.

Corollary 43. $(\forall n.\ t^f_{s^n} \lesssim u) \leftrightarrow (\textbf{\textit{def}}\ f = s\ \textbf{\textit{in}}\ t \lesssim u)$

8.2 Continuity for GC

Our proof of continuity for the WP semantics of GC uses a general fact about least fixed points of continuous functions.

Lemma 44. *Let $F : (X \to \mathbf{P}) \to (X \to \mathbf{P}) \to X \to \mathbf{P}$ be a binary predicate transformer and let F be continuous in both arguments. Then the predicate transformer $\mathrm{fix} \circ F$ is continuous.*

Proof. Let $(D_n)_{n\in\mathbb{N}}$ be a chain of predicates. We have to show that

$$\mathrm{fix}(F\, (\bigcup_{n\in\mathbb{N}} D_n)) \equiv \bigcup_{n\in\mathbb{N}} \mathrm{fix}(F\, D_n)$$

The $\subseteq$-inclusion follows by induction using the fact that $(D_n)_n$ is a chain.

$$\begin{aligned}F\, (\bigcup_{n\in\mathbb{N}} D_n)\, (\bigcup_{n\in\mathbb{N}} \mathrm{fix}(F\, D_n)) &\subseteq \bigcup_{n\in\mathbb{N}} F\, D_n\, (\mathrm{fix}(F\, D_n)) \\ &\equiv \bigcup_{n\in\mathbb{N}} \mathrm{fix}(F\, D_n)\end{aligned}$$

The $\supseteq$-inclusion follows by another induction. □

Lemma 45. *The predicate transformer* wp *s is continuous for all GC programs s.*

Proof. The direction "$\supseteq$" is equivalent to monotonicity. In the reverse direction we proceed by induction on s.

For conditional statements **if** G, we have a $Q_i \in D$ for each guard in G which is satisfied in the current state. Since (D_n) is a chain and G is finite, there is an index n such that $Q_i \subseteq D_n$. The result follows by monotonicity of wp.

The case for loops is an instance of Lemma 44. □

Dijkstra defines the semantics of loops with the help of a recursive function H.

$$\begin{aligned}H\, G\, Q\, k\ :=\ &\text{if } k = 0 \text{ then } Q \cap \neg\widehat{G} \\ &\text{else } \mathrm{wp}\, (\textsf{if}\ G)\, (H\, G\, Q\, (k-1)) \cup H\, G\, Q\, 0\end{aligned}$$

We formulate a predicate transformer F

$$F\, G\, Q\ :=\ \mathrm{wp}\, (\textsf{if}\ G)\, Q \cup (Q \cap \neg\widehat{G})$$

and show that H can be expressed by iteration of F.

Lemma 46. $H\, G\, Q\, k \equiv F^{k+1} \bot$

Proof. By induction on k using Lemma 3 and Fact 2. □

Finally, we show that our semantics admits Dijkstra's definition [4] of the weakest precondition for loops.

Theorem 47. $\mathrm{wp}\, (\textbf{\textit{do}}\ G)\, Q\ \equiv\ \exists k.\ H\, G\, Q\, k$

Proof. By Lemma 46 and Fact 39. □

9. Formal Development

All results in the paper have been formalized in the proof assistant Coq. The development is available online at the following URL: `http://www.ps.uni-saarland.de/extras/ascv`. In this section we discuss the technical differences between the paper presentation and its Coq formalization.

Binders We use the de Bruijn [3] representation of binders and parallel substitutions in the formalization, instead of named terms modulo α-equivalence. The Autosubst library [13] is used to generate the capture avoiding substitution operation for IC and to automate proofs of substitution lemmas.

Recall that in de Bruijn representation we replace named variables by references to their binders. These references are implemented with indices: natural numbers where the number n refers to the n-th enclosing binder, counting from 0.

Substitutions are functions from indices to terms. Instantiation of a term s under a substitution θ, written $s[\theta]$, replaces all free variables in s according to θ.

We generalize all lemmas about single variable substitutions to parallel substitutions. For example, Lemma 12 takes the form

$$\mathrm{wp}\, t[\theta]\, \mathcal{Q}\ \equiv \mathrm{wp}\, t\, (\mathrm{wp}\, \theta\, \mathcal{Q})$$
$$\text{where } \mathrm{wp}\, \theta\, \mathcal{Q}\, f := \mathrm{wp}(\theta f)\, \mathcal{Q}$$

Formally, the proof is by induction on the structure of the instantiation operation. This means that we first show the lemma for

renamings $\xi : \mathbb{N} \to \mathbb{N}$ and then use this to show the statement for all substitutions.

Freshness assumptions are encoded differently in de Bruijn representation. A binder always introduces a fresh variable by definition, but changes the indices corresponding to all other variables in its scope. For instance, in the compilation of loop statements we have a freshness assumption for the continuation f.

$$\mathcal{T}\, u\, (\mathsf{do}\ G) = \mathsf{def}\ f = \mathcal{F}\, (\mathcal{T} f)\, u\, G\ \mathsf{in}\ f$$
(where f is fresh)

In the de Bruijn version, the freshness assumption is encoded by adjusting the indices in u.

$$\mathcal{T}\, u\, (\mathsf{do}\ G) = \mathsf{def}\ \mathcal{F}\, (\mathcal{T} 0)\, u[+1]\, G\ \mathsf{in}\ 0$$

Where $(+1)$ is the substitution which skips an additional binder, i.e., increases all free variables by 1.

Fixed-point Operator The definition of fix depends on the impredicativity of the universe of propositions. Alternatively, we could have used Fact 39 to define fixed-points only for continuous predicate transformers. This definition may be preferable in predicative type theories, but complicates the proof of Theorem 8 and 11. Given the usefulness of the WP semantics to results such as Lemma 12, we prefer to work with a more flexible fixed point operator.

WP Semantics The WP semantics for GC and IC is a denotional semantics formulated by recursion on syntax. In the paper, the WP semantics is presented as a number of equivalences, which we show admissible in Theorem 8 and 11. In the Coq development, we realize a type-theoretic function wp according to these equivalences and exploit convertibility in the proofs.

Technical Characteristics The formalization uses Ssreflect [7] (version 1.5) for its compact and consistent tactic language and extensive library. We assume the axiom of functional extensionality, since it is used in Autosubst.

The development is parameterized over the types of states, actions, and guards. We do not assume that all functions on states are available as actions, or that all boolean functions are available as guards.

The complete development consists of around 300 lines of specification and 600 lines of proofs.

10. Related Work

Contextual Equivalence Contextual equivalence for the untyped λ-calculus was introduced as "extensional equivalence" by Morris [11]. Morris shows that his extensional equivalence is the coarsest congruence that preserves termination.

Showing that two programs coterminate in all contexts directly is not convenient. Alternative characterizations of contextual equivalence that provide viable proof methods have hence received considerable attention; see, for example, the book by Harper [8] and the work by Pitts [12].

Axiomatic Semantics Axiomatic semantics describe the meaning of programs in terms of a program logic. The axiomatic style of semantics is attributed to Floyd [6] and Hoare [9]. Verification-oriented axiomatic semantics support reasoning about total and partial correctness. The axiomatic semantics we consider in this paper account for total correctness only.

Dijkstra gives an axiomatic semantics to GC in terms of predicate transformers for weakest preconditions [4], which are formulated with ω-iteration in the case of loops, and shows continuity of the transformers [5].

Winskel [18] gives axiomatic semantics for IMP, and shows that the big-step semantics is sound for the axiomatic semantics. The other direction does not hold, since Winskel's axiomatic semantics accounts for partial correctness.

The way we connect operational and axiomatic semantics bears a close resemblance to Charguéraud's [2] work on characteristic formulas for program verification. Schirmer [14] develops a library for program verification based on axiomatic semantics in Isabelle/HOL.

Fixed-points The facts about fixed-point theorems we state in this paper can be found, for example, in Winskel [18].

Simulation-based Compiler Correctness CompCert [10] is a realistic, verified compiler. The languages in CompCert are first-order. The correctness statement in CompCert is formulated in terms of a coinductive simulation, which distinguishes diverging behaviors according to the sequence of occurring system calls. The axiomatic methods in this paper only treat the terminating case, and do not consider system calls.

Leroy [10] relies on the coincidence of forward and backward simulation for the correctness proofs in CompCert. Sevcík [16] reports in the context of CompCertTSO that a particular optimization could not be verified with standard forward simulations because of unobservable non-determinism. The axiomatic methods in this paper deal with unobservable non-determinism in GC.

Axiomatic Methods for Compiler Correctness Wang [17] uses Hoare logic to verify cross-language linking for a compiler. The semantics is an operational big-step semantics relating input and output states. The function application rule requires a certain specification, which is expressed as a Hoare triple, to hold. Specifications in Hoare logic are then used to interface between modules from different languages. The correctness statement of the compiler ensures that valid Hoare triples are preserved by compilation.

Imperative Continuations The language IC is an idealization of the intermediate language IL/I [15]. IC treats state abstractly. Like IL/I, IC does not permit arbitrary jumps but uses lexically scoped gotos. Since IC lacks mutual recursion, control flow in IC programs is reducible [1]. In contrast to IL/I, IC does not contain a `return` construct to terminate execution, but treats return uniformly as just another continuation.

11. Conclusion and Future Work

We present an axiomatic semantics for Dijkstra's GC language [4] with correctness judgments $\langle \sigma \rangle\, s\, \langle Q \rangle$. In contrast to big-step semantics, the post-condition Q accounts for all—possibly nondeterministic—behaviors of s and σ. Unlike verification-oriented axiomatic semantics, we formulate the rule for loops with an interpolant instead of an invariant. Hoare logic, which uses an invariant for loops, supports partial and total correctness. Our formulation with an interpolant has total correctness built in. As a direction for future work, we would like to investigate whether the coinductive interpretation of the rules defining the axiomatic semantics provides judgments for partial correctness.

Dijkstra's original semantic specification of GC [4] is a recursive function computing weakest preconditions. We define weakest preconditions directly in terms of the axiomatic semantics. Similar to Dijkstra, we give a characterization of the weakest preconditions by recursion on the syntax of GC. In the case of loops, we use a fixed-point operator, while Dijkstra uses ω-iteration. We show that the predicate transformers computing weakest preconditions for GC and IC programs are continuous. This ensures that the fixed-points that occur in our definitions can be characterized in terms of ω-iteration. Building on this result, we formally show our axiomatic semantics admits Dijkstra's formulation of the loop semantics.

We consider a low-level language IC with a substitution-based small-step semantics. We give an axiomatic semantics for IC and

show that it coincides with the small-step semantics. In future work, we want to give an environment-based small-step semantics with closures to IC. An interesting question is whether the axiomatic semantic facilitates proving that the closure semantics coincides with the substitution semantics. We are also interested in extending IC with mutual recursion. Mutual recursion closes the gap to imperative languages that are formulated with an external table of mutually recursive definitions. Examples of such languages can be found, for example, in Winskel [18].

We verified a compiler from GC to IC. We formulate compiler correctness in terms of the axiomatic semantics as preservation of specifications. The axiomatic semantics makes it easy to account for the semantic underspecification of GC. The correctness proof is a straight-forward induction on the derivation of the axiomatic GC semantics. The WP semantics is useful for the proof that the axiomatic semantics coincides with the small-step semantics, and for the proof that two IC programs are contextually equivalent if and only if their weakest preconditions are equivalent. It is an open question whether an axiomatic semantics can be used to account for different diverging behaviors, which are distinguished by I/O, for example.

References

[1] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[2] Arthur Charguéraud. Program verification through characteristic formulae. *ACM Sigplan Notices*, 45(9):321–332, 2010.

[3] Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[4] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[5] Edsger W Dijkstra. A discipline of programming. *Prentice-Hall Series in Automatic Computation, Englewood Cliffs: Prentice-Hall, 1976*, 1, 1976.

[6] Robert W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967. American Mathematical Society.

[7] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the Coq system. 2008.

[8] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[10] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[11] James Hiram Morris Jr. *Lambda-calculus models of programming languages.* PhD thesis, Massachusetts Institute of Technology, 1969.

[12] Andrew M. Pitts. Operational semantics and program equivalence. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer, 2000.

[13] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In *Interactive Theorem Proving*, pages 359–374. Springer, 2015.

[14] Norbert Schirmer et al. *Verification of sequential imperative programs in Isabelle-HOL.* PhD thesis, Technical University Munich, 2006.

[15] Sigurd Schneider, Gert Smolka, and Sebastian Hack. A linear first-order functional intermediate language for verified compilers. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 2015.

[16] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompcertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[17] Peng Wang, Santiago Cuellar, and Adam Chlipala. Compiler verification meets cross-language linking via data abstraction. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 675–690. ACM, 2014.

[18] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.