Mechanized semantics

with applications to program proof and compiler verification

Xavier LEROY
INRIA Paris-Rocquencourt

Introduction

The semantics of a programming language describe *mathematically* the meaning of programs written in this language. An example of use of semantics is to define a programming language with much greater precision than standard language specifications written in English. (See for example the definition of Standard ML [35].) In turn, semantics enable us to formally verify some programs, proving that they satisfy their specifications. Finally, semantics are also necessary to establish the correctness of algorithms and implementations that operate over programs: interpreters, compilers, static analyzers (including type-checkers and bytecode verifiers), program provers, refactoring tools, etc.

Semantics for nontrivial programming languages can be quite large and complex, making traditional, on-paper proofs using these semantics increasingly painful and unreliable. Automatic theorem provers and especially interactive proof assistants have great potential to alleviate these problems and scale semantic-based techniques all the way to realistic programming languages and tools. Popular proof assistants that have been successfully used in this area include ACL2, Coq, HOL4, Isabelle/HOL, PVS and Twelf.

The purpose of this lecture is to introduce students to this booming field of mechanized semantics and its applications to program proof and formal verification of programming tools such as compilers. Using the prototypical IMP imperative language as a concrete example, we will:

- mechanize various forms of operational and denotational semantics for this language and prove their equivalence (section 1);
- introduce axiomatic semantics (Hoare logic) and show how to provide machine assistance for proving IMP programs using a verification condition generator (section 2);
- define a non-optimizing compiler from IMP to a virtual machine (a small subset of the Java virtual machine) and prove the correctness of this compiler via a semantic preservation argument (section 3);
- illustrate optimizing compilation through the development and proof of correctness of a dead code elimination pass (section 4).

We finish with examples of recent achievements and ongoing challenges in this area (section 5).

We use the Coq proof assistant to specify semantics and program transformations, and conduct all proofs. The best reference on Coq is Bertot and Castéran's book [13], but

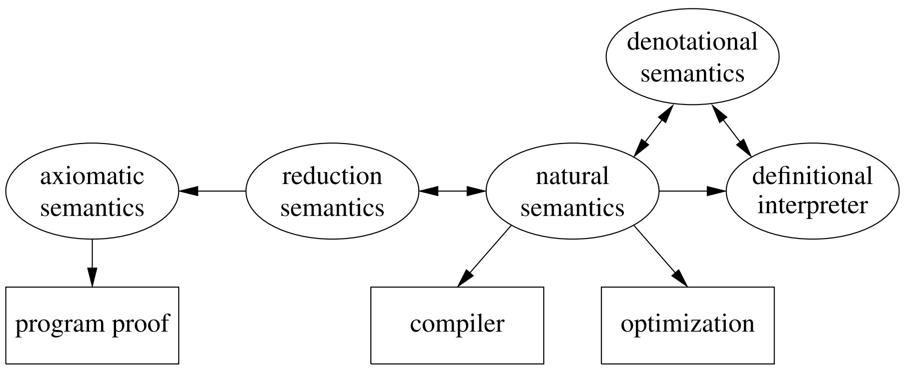

Figure 1. The various styles of semantics considered in this lecture and their uses. A double arrow denotes a semantic equivalence result. A single arrow from A to B means that semantics A is used to justify the correctness of B.

for the purposes of this lecture, Bertot's short tutorial [12] is largely sufficient. The Coq software and documentation is available as free software at `http://coq.inria.fr/`. By lack of time, we will not attempt to teach how to conduct interactive proofs in Coq (but see the two references above). However, we hope that by the end of this lecture, students will be familiar enough with Coq's specification language to be able to read the Coq development underlying this lecture, and to write Coq specifications for problems of their own interest.

The reference material for this lecture is the Coq development available at `http://gallium.inria.fr/~xleroy/courses/Marktoberdorf-2009/`. These preliminary notes recapitulate the definitions and main results using ordinary mathematical syntax, and provides bibliographical references. To help readers make the connection with the Coq development, the Coq names for the definitions and theorems are given as bracketed notes, [like this]. In the PDF version of these notes, available at the Web site above, these notes are hyperlinks pointing directly to the corresponding Coq definitions and theorems in the development.

1. The IMP language and its semantics

1.1. Syntax

The IMP language is a very simple imperative language with structured control.

Expressions: [expr]

$$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2$$

Boolean expressions (conditions): [bool_expr]

$$b ::= e_1 = e_2 \mid e_1 < e_2$$

Commands (statements): [cmd]

$$c ::= \texttt{skip} \mid x := e \mid c_1; c_2 \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}$$

The semantics of expressions is given in denotational style, as a function from states to integers or booleans [eval_expr] [eval_bool_expr]. States are mappings from variable names x to integers [state]

$$[\![x]\!]\ s = s(x) \qquad [\![n]\!]\ s = n$$
$$[\![e_1 + e_2]\!]\ s = [\![e_1]\!]\ s + [\![e_2]\!]\ s \qquad [\![e_1 - e_2]\!]\ s = [\![e_1]\!]\ s - [\![e_2]\!]\ s$$
$$[\![e_1 = e_2]\!]\ s = \begin{cases} \texttt{true} & \text{if } [\![e_1]\!]\ s = [\![e_2]\!]\ s; \\ \texttt{false} & \text{if } [\![e_1]\!]\ s \neq [\![e_2]\!]\ s; \end{cases}$$
$$[\![e_1 = e_2]\!]\ s = \begin{cases} \texttt{true} & \text{if } [\![e_1]\!]\ s < [\![e_2]\!]\ s; \\ \texttt{false} & \text{if } [\![e_1]\!]\ s \geq [\![e_2]\!]\ s; \end{cases}$$

1.2. Reduction semantics

A popular way to give semantics to languages such as IMP, where programs may not terminate, is reduction semantics, popularized by Plotkin under the name “structural operational semantics” [46], and also called “small-step semantics”. It builds on a reduction relation $(c, s) \to (c', s')$, meaning: in initial state s, the command c performs one elementary step of computation, resulting in modified state s' and residual computations c'. [red]

$$(x := e,\ s) \to (\texttt{skip},\ s[x \leftarrow [\![e]\!]\ s]) \quad \text{[red\_assign]}$$

$$\frac{(c_1, s) \to (c_1', s)}{((c_1; c_2),\ s) \to ((c_1'; c_2),\ s')} \ \text{[red\_seq\_left]} \qquad ((\texttt{skip}; c),\ s) \to (c, s) \ \text{[red\_seq\_skip]}$$

$$\frac{[\![b]\!]\ s = \texttt{true}}{((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2), s) \to (c_1, s)} \ \text{[red\_if\_true]}$$

$$\frac{[\![b]\!]\ s = \texttt{false}}{((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2), s) \to (c_2, s)} \ \text{[red\_if\_false]}$$

$$\frac{[\![b]\!]\ s = \texttt{true}}{((\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}), s) \to ((c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}), s)} \ \text{[red\_while\_true]}$$

$$\frac{[\![b]\!]\ s = \texttt{false}}{((\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}), s) \to (\texttt{skip}, s)} \ \text{[red\_while\_false]}$$

The behavior of a command c in an initial state s is obtained by forming sequences of reductions starting at c, s:

- Termination with final state s' ($c, s \Downarrow s'$): finite sequence of reductions to `skip`. [terminates]

$$(c, s) \xrightarrow{*} (\texttt{skip}, s')$$

- Divergence ($c, s \Uparrow$): infinite sequence of reductions. [diverges]

$$\forall (c', s'), (c, s) \xrightarrow{*} (c', s') \Rightarrow \exists c'', s'', (c', s') \to (c'', s'')$$

- Going wrong ($c, s \Downarrow \texttt{wrong}$): finite sequence of reductions to an irreducible state that is not `skip`. [goes_wrong]

$$(c, s) \to \cdots \to (c', s') \not\to \ \text{ with } c \neq \texttt{skip}$$

1.3. *Natural semantics*

An alternative to structured operational semantics is Kahn's natural semantics [24], also called big-step semantics. Instead of describing terminating executions as sequences of reductions, natural semantics aims at giving a direct axiomatization of executions using inference rules.

Execution relation $c, s \Rightarrow s'$ (from initial state s, the command c terminates with final state s'). [exec]

$$\texttt{skip}, \texttt{s} \Rightarrow \texttt{s} \quad \text{[exec\_skip]} \qquad\qquad x := e, s \Rightarrow s[x \leftarrow [\![e]\!]\ s] \quad \text{[exec\_assign]}$$

$$\frac{c_1, s \Rightarrow s_1 \qquad c_2, s_1 \Rightarrow s_2}{c_1; c_2, s \Rightarrow s_2} \ \text{[exec\_seq]} \qquad \frac{\begin{array}{c} c_1, s \Rightarrow s' \text{ if } [\![b]\!]\ s = \texttt{true} \\ c_2, s \Rightarrow s' \text{ if } [\![b]\!]\ s = \texttt{false} \end{array}}{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, s \Rightarrow s'} \ \text{[exec\_if]}$$

$$\frac{[\![b]\!]\ s = \texttt{false}}{\texttt{while } b \texttt{ do } c \texttt{ done}, s \Rightarrow s} \ \text{[exec\_while\_stop]}$$

$$\frac{[\![b]\!]\ s = \texttt{true} \qquad c, s \Rightarrow s_1 \qquad \texttt{while } b \texttt{ do } c \texttt{ done}, s_1 \Rightarrow s_2}{\texttt{while } b \texttt{ do } c \texttt{ done}, s \Rightarrow s_2} \ \text{[exec\_while\_loop]}$$

Equivalence results between reduction semantics and natural semantics for termination:

Theorem 1 [exec_terminates] *If* $c, s \Rightarrow s'$, *then* $(c, s) \xrightarrow{*} (\texttt{skip}, s')$.

Lemma 2 [red_preserves_exec] *If* $(c, s) \to (c', s')$ *and* $c', s' \Rightarrow s''$, *then* $c, s \Rightarrow s''$.

Theorem 3 [terminates_exec] *If* $(c, s) \xrightarrow{*} (\texttt{skip}, s')$, *then* $c, s \Rightarrow s'$.

As observed by Grall and Leroy [30], diverging executions can also be described in the style of natural semantics. Define the infinite execution relation $c, s \Rightarrow \infty$ (from initial state s, the command c diverges). [execinf]

$$\frac{c_1, s \Rightarrow \infty}{\overline{c_1; c_2, s \Rightarrow \infty}} \ \text{[execinf\_seq\_left]} \qquad \frac{c_1, s \Rightarrow s_1 \qquad c_2, s_1 \Rightarrow \infty}{\overline{c_1; c_2, s \Rightarrow \infty}} \ \text{[execinf\_seq\_right]}$$

$$\frac{\begin{array}{c} c_1, s \Rightarrow \infty \text{ if } [\![b]\!]\ s = \texttt{true} \\ c_2, s \Rightarrow \infty \text{ if } [\![b]\!]\ s = \texttt{false} \end{array}}{\overline{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, s \Rightarrow \infty}} \ \text{[execinf\_if]}$$

$$\dfrac{[\![b]\!]\ s = \texttt{true} \qquad c, s \Rightarrow \infty}{\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}, s \Rightarrow \infty}\ \text{[execinf\_while\_body]}$$

$$\dfrac{[\![b]\!]\ s = \texttt{true} \qquad c, s \Rightarrow s_1 \qquad \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}, s_1 \Rightarrow \infty}{\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}, s \Rightarrow \infty}\ \text{[execinf\_while\_loop]}$$

As denoted by the double horizontal bars, these rules must be interpreted *coinductively* as a greatest fixpoint [30, section 2]. Equivalently, the coinductive interpretation corresponds to conclusions of *infinite* derivation trees, while the inductive interpretation corresponds to *finite* derivation trees.

Equivalence results between reduction semantics and natural semantics for divergence:

Lemma 4 [execinf_red_step] *If $c, s \Rightarrow \infty$, there exists c' and s' such that $(c, s) \to (c', s')$ and $c', s' \Rightarrow \infty$.*

Theorem 5 [execinf_diverges] *If $c, s \Rightarrow \infty$, then $(c, s) \Uparrow$.*

Lemma 6 [red_deterministic] *If $c \to c_1$ and $c \to c_2$, then $c_1 = c_2$.*

Lemma 7 [diverges_star_inv] *If $(c, s) \Uparrow$ and $(c, s) \xrightarrow{*} (c', s')$, then $(c', s') \Uparrow$.*

Theorem 8 [diverges_execinf] *If $(c, s) \Uparrow$, then $c, s \Rightarrow \infty$.*

1.4. Definitional interpreter

Evaluation function $\mathcal{I}(n, c, s)$: evaluates c in initial state s at maximal recursion depth n; returns either $\lfloor s' \rfloor$ (termination with state s') or $\bot$ (insufficient recursion depth). [interp]

$$\begin{aligned}
\mathcal{I}(0, c, s) &= \bot \\
\mathcal{I}(n+1, \texttt{skip}, s) &= \lfloor s \rfloor \\
\mathcal{I}(n+1, x := e, s) &= \lfloor s[x \leftarrow [\![e]\!]\ s]\ \rfloor \\
\mathcal{I}(n+1, (c_1; c_2), s) &= \mathcal{I}(n, c_1, s) \rhd (\lambda s'.\, \mathcal{I}(n, c_2, s')) \\
\mathcal{I}(n+1, (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2), s) &= \mathcal{I}(n, c_1, s) \text{ if } [\![b]\!]\ s = \texttt{true} \\
\mathcal{I}(n+1, (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2), s) &= \mathcal{I}(n, c_2, s) \text{ if } [\![b]\!]\ s = \texttt{false} \\
\mathcal{I}(n+1, (\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}), s) &= \lfloor s \rfloor \text{ if } [\![b]\!]\ s = \texttt{false} \\
\mathcal{I}(n+1, (\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}), s) &= \mathcal{I}(n, c, s) \rhd (\lambda s'.\, \mathcal{I}(n, \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}, s')) \\
&\qquad \text{if } [\![b]\!]\ s = \texttt{true}
\end{aligned}$$

The “bind” operator $\rhd$ is defined by $\bot \rhd f = \bot$ and $\lfloor s \rfloor \rhd f = f(s)$.

Evaluation results are ordered by $\bot \leq \lfloor s \rfloor$ [res_le].

Lemma 9 [interp_mon] *(Monotonicity of $\mathcal{I}$.) If $n \leq m$, then $\mathcal{I}(n, c, s) \leq \mathcal{I}(m, c, s)$.*

Partial correctness of the definitional interpreter with respect to the big-step semantics:

Lemma 10 [interp_exec] *If $\mathcal{I}(n, c, s) = \lfloor s' \rfloor$, then $c, s \Rightarrow s'$.*

Lemma 11 [exec_interp] *If $c, s \Rightarrow s'$, there exists an n such that $\mathcal{I}(n, c, s) = \lfloor s' \rfloor$.*

Lemma 12 [execinf_interp] *If $c, s \Rightarrow \infty$, then $\mathcal{I}(n, c, s) = \bot$ for all n.*

1.5. Denotational semantics

A form of denotational semantics [38] can be obtained by "letting n goes to infinity" in the definitional interpreter.

Lemma 13 [interp_limit_dep] *For every c, there exists a function $[\![c]\!]$ from states to evaluation results such that $\forall s,\ \exists m,\ \forall n \geq m,\ \mathcal{I}(n, c, s) = [\![c]\!]\ s$.*

This denotation function satisfies the equations of denotational semantics:

$$\begin{aligned}
[\![\texttt{skip}]\!]\ s &= \lfloor s \rfloor \\
[\![x := e]\!]\ s &= \lfloor s[x \leftarrow [\![e]\!]\ s] \rfloor \\
[\![c_1; c_2]\!]\ s &= [\![c_1]\!]\ s \rhd (\lambda s'.\ [\![c_2]\!]\ s') \\
[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\ s &= [\![c_1]\!]\ s\ \text{if}\ [\![b]\!]\ s = \texttt{true} \\
[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\ s &= [\![c_2]\!]\ s\ \text{if}\ [\![b]\!]\ s = \texttt{false} \\
[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}]\!]\ s &= \lfloor s \rfloor\ \text{if}\ [\![b]\!]\ s = \texttt{false} \\
[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}]\!]\ s &= [\![c]\!]\ s \rhd (\lambda s'.\ [\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}]\!]\ s')\ \text{if}\ [\![b]\!]\ s = \texttt{true}
\end{aligned}$$

Moreover, $[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}]\!]$ is the smallest function from states to results that satisfies the last two equations.

Theorem 14 [denot_exec] [exec_denot] *$c, s \Rightarrow s'$ if and only if $[\![c]\!]\ s = \lfloor s' \rfloor$.*

Theorem 15 [denot_execinf] [execinf_denot] *$c, s \Rightarrow \infty$ if and only if $[\![c]\!]\ s = \bot$.*

1.6. Further reading

The material presented in this section is inspired by Nipkow [41] (in Isabelle/HOL, for the IMP language) and by Grall and Leroy [30] (in Coq, for the call-by-value λ-calculus).

We followed Plotkin's "SOS" presentation [46] of reduction semantics, characterized by structural inductive rules such as [red_seq_left]. An alternate presentation, based on reduction contexts, was introduced by Wright and Felleisen [51] and is very popular to reason about type systems [45].

Definitions and proofs by coinduction can be formalized in two ways: as greatest fixpoints in a set-theoretic presentation [1] or as infinite derivation trees in proof theory [13, chap. 13]. Grall and Leroy [30] connect the two approaches.

The definitional interpreter approach was identified by Reynolds in 1972. See [47] for a historical perspective.

The presentation of denotational semantics we followed avoids the complexity of Scott domains. Mechanizations of domain theory with applications to denotational semantics include Agerholm [2] (in HOL), Paulin [43] (in Coq) and Benton *et al.* [9] (in Coq).

2. Axiomatic semantics and program verification

Operational semantics as in section 1 focuses on describing actual executions of programs. In contrast, axiomatic semantics focuses on verifying logical assertions between the values of programs at various program points. It is the most popular approach to proving the correctness of imperative programs.

2.1. The rules of axiomatic semantics

Hoare triples: $\{\, P \,\}\ c\ \{\, Q \,\}$ (if precondition P holds, c does not go wrong, and if it terminates, the postcondition Q holds). P and Q are arbitrary predicates over states.

Some operations over predicates:

$$\begin{array}{rclrcl} P[x \leftarrow e] & \stackrel{\text{def}}{=} & \lambda s.\ P(s[x \leftarrow [\![e]\!]\ s]) & P \wedge Q & \stackrel{\text{def}}{=} & \lambda s.\ P(s) \wedge Q(s) \\ b\ \texttt{true} & \stackrel{\text{def}}{=} & \lambda s.\ [\![b]\!]\ s = \texttt{true} & P \vee Q & \stackrel{\text{def}}{=} & \lambda s.\ P(s) \vee Q(s) \\ b\ \texttt{false} & \stackrel{\text{def}}{=} & \lambda s.\ [\![b]\!]\ s = \texttt{false} & P \Longrightarrow Q & \stackrel{\text{def}}{=} & \forall s,\ P(s) \Longrightarrow Q(s) \end{array}$$

The axiomatic semantics, that is, the set of legal triples $\{\, P \,\}\ c\ \{\, Q \,\}$, is defined by the following inference rules: [triple]

$$\{\, P \,\}\ \texttt{skip}\ \{\, P \,\} \quad \text{[triple\_skip]} \qquad \{\, P[x \leftarrow e] \,\}\ x := e\ \{\, P \,\} \quad \text{[triple\_assign]}$$

$$\frac{\{\, P \,\}\ c_1\ \{\, Q \,\} \qquad \{\, Q \,\}\ c_2\ \{\, R \,\}}{\{\, P \,\}\ c_1; c_2\ \{\, R \,\}} \quad \text{[triple\_seq]}$$

$$\frac{\{\, b\ \texttt{true} \wedge P \,\}\ c_1\ \{\, Q \,\} \qquad \{\, b\ \texttt{false} \wedge P \,\}\ c_2\ \{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{\, Q \,\}} \quad \text{[triple\_if]}$$

$$\frac{\{\, b\ \texttt{true} \wedge P \,\}\ c\ \{\, P \,\}}{\{\, P \,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ \{\, b\ \texttt{false} \wedge P \,\}} \quad \text{[triple\_while]}$$

$$\frac{P \Longrightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad Q' \Longrightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}} \quad \text{[triple\_consequence]}$$

2.2. Soundness of the axiomatic semantics

Intuitively, a Hoare triple $\{\, P \,\}\ c\ \{\, Q \,\}$ is valid if for all initial states s such that $P\ s$ holds, either (c, s) diverges or it terminates in a state s' such that $Q\ s'$ holds. We capture the latter condition by the predicate (c, s) `finally` Q, defined coinductively as: [finally]

$$\frac{Q(s)}{(\texttt{skip}, s)\ \texttt{finally}\ Q}\ \text{[finally\_done]}$$

$$\frac{(c, s) \to (c', s') \qquad (c', s')\ \texttt{finally}\ Q}{(c, s)\ \texttt{finally}\ Q}\ \text{[finally\_step]}$$

The semantic interpretation $[\![\{\, P \,\}\ c\ \{\, Q \,\}]\!]$ of a triple is, then, the proposition

$$\forall s,\ P\ s \Longrightarrow (c, s)\ \texttt{finally}\ Q \qquad \text{[sem\_triple]}$$

Lemma 16 [finally_seq] *If* (c_1, s) $\texttt{finally}$ Q *and* $[\![\{\, Q \,\}\ c_2\ \{\, R \,\}]\!]$*, then* $((c_1; c_2), s)$ $\texttt{finally}$ R*.*

Lemma 17 [finally_while] *If* $[\![\{\, b\ \texttt{true} \wedge P \,\}\ c\ \{\, P \,\}]\!]$ *then* $[\![\{\, P \,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ \{\, b\ \texttt{false} \wedge P \,\}]\!]$*.*

Lemma 18 [finally_consequence] *If* (c, s) $\texttt{finally}$ Q *and* $Q \Longrightarrow Q'$*, then* (c, s) $\texttt{finally}$ Q'*.*

Theorem 19 [triple_correct] *If* $\{\, P \,\}\ c\ \{\, Q \,\}$ *can be derived by the rules of axiomatic semantics, then* $[\![\{\, P \,\}\ c\ \{\, Q \,\}]\!]$ *holds.*

2.3. Generation of verification conditions

In this section, we enrich the syntax of IMP commands with an annotation on `while` loops (to give the loop invariant) and an `assert`(P) command to let the user provide assertions. [acmd]

Annotated commands:

$c ::= \texttt{while}\ b\ \texttt{do}\ \{P\}\ c\ \texttt{done}$	loop with invariant
$\mid\ \texttt{assert}(P)$	explicit assertion
$\mid\ \ldots$	other commands as in IMP

Annotated commands can be viewed as regular commands by erasing the $\{P\}$ annotation on loops and turning `assert`(P) to `skip`. [erase]

The `wp` function computes the weakest (liberal) precondition for c given a postcondition Q: [wp]

$$\begin{aligned}
\texttt{wp}(\texttt{skip}, Q) &= Q \\
\texttt{wp}(x := e, Q) &= Q[x \leftarrow e] \\
\texttt{wp}((c_1; c_2), Q) &= \texttt{wp}(c_1, \texttt{wp}(c_2, Q)) \\
\texttt{wp}((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2), Q) &= (b\ \texttt{true} \wedge \texttt{wp}(c_1, Q)) \vee (b\ \texttt{false} \wedge \texttt{wp}(c_2, Q)) \\
\texttt{wp}((\texttt{while}\ b\ \texttt{do}\ \{P\}\ c\ \texttt{done}), Q) &= P \\
\texttt{wp}(\texttt{assert}(P), Q) &= P
\end{aligned}$$

With the same arguments, the vcg function computes a conjunction of implications that must hold for the triple $\{\, \mathtt{wp}(c, Q) \,\}\ c\ \{\, Q \,\}$ to hold. [vcg]

$$\begin{aligned}
\mathtt{vcg}(\mathtt{skip}, Q) &= T \\
\mathtt{vcg}(x := e, Q) &= T \\
\mathtt{vcg}((c_1; c_2), Q) &= \mathtt{vcg}(c_1, \mathtt{wp}(c_2, Q)) \wedge \mathtt{vcg}(c_2, Q) \\
\mathtt{vcg}((\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2), Q) &= \mathtt{vcg}(c_1, Q) \wedge \mathtt{vcg}(c_2, Q) \\
\mathtt{vcg}((\mathtt{while}\ b\ \mathtt{do}\ \{P\}\ c\ \mathtt{done}), Q) &= \mathtt{vcg}(c, P) \\
&\quad \wedge (b\ \mathtt{false} \wedge P \implies Q) \\
&\quad \wedge (b\ \mathtt{true} \wedge P \implies \mathtt{wp}(c, P)) \\
\mathtt{vcg}(\mathtt{assert}(P), Q) &= P \implies Q
\end{aligned}$$

Lemma 20 [vcg_correct] *If* $\mathtt{vcg}(c, Q)$ *holds, then* $\{\, \mathtt{wp}(c, Q) \,\}\ c\ \{\, Q \,\}$ *can be derived by the rules of axiomatic semantics.*

The derivation of a Hoare triple $\{\, P \,\}\ c\ \{\, Q \,\}$ can therefore be reduced to the computation of the following $\mathtt{vcgen}(P, c, Q)$ logical formula, and its proof. [vcgen]

$$\mathtt{vcgen}(P, c, Q) \ \stackrel{\text{def}}{=}\ (P \implies \mathtt{wp}(c, Q)) \wedge \mathtt{vcg}(c, Q)$$

Theorem 21 [vcgen_correct] *If* $\mathtt{vcgen}(P, c, Q)$ *holds, then* $\{\, P \,\}\ c\ \{\, Q \,\}$ *can be derived by the rules of axiomatic semantics.*

Example. Consider the following annotated IMP program c:

```
r := a; q := 0;
while b < r+1 do {I} r := r - b; q := q + 1 done
```

and the following precondition P, loop invariant I and postcondition Q:

$$\begin{aligned}
P &\ \stackrel{\text{def}}{=}\ \lambda s.\ s(\mathtt{a}) \geq 0 \wedge s(\mathtt{b}) > 0 \\
I &\ \stackrel{\text{def}}{=}\ \lambda s.\ s(\mathtt{r}) \geq 0 \wedge s(\mathtt{b}) > 0 \wedge s(\mathtt{a}) = s(\mathtt{b}) \times s(\mathtt{q}) + s(\mathtt{r}) \\
Q &\ \stackrel{\text{def}}{=}\ \lambda s.\ s(\mathtt{q}) = s(\mathtt{a}) / s(\mathtt{b})
\end{aligned}$$

To prove that $\{\, P \,\}\ c\ \{\, Q \,\}$, we apply theorem 21, then ask Coq to compute and simplify the formula $\mathtt{vcgen}(P, c, Q)$. We obtain the conjunction of three implications:

$$s(\mathtt{a}) \geq 0 \wedge s(\mathtt{b}) > 0 \implies s(\mathtt{a}) \geq 0 \wedge s(\mathtt{b}) > 0 \wedge s(\mathtt{a}) = s(\mathtt{b}) \times 0 + s(\mathtt{a})$$

$$\begin{aligned}
&\neg(s(\mathtt{b}) < s(\mathtt{r}) + 1) \wedge s(\mathtt{r}) \geq 0 \wedge s(\mathtt{b}) > 0 \wedge s(\mathtt{a}) = s(\mathtt{b}) \times s(\mathtt{q}) + s(\mathtt{r}) \\
&\qquad \implies s(\mathtt{q}) = s(\mathtt{a}) / s(\mathtt{b})
\end{aligned}$$

$$\begin{aligned}
&s(\mathtt{b}) < s(\mathtt{r}) + 1 \wedge s(\mathtt{r}) \geq 0 \wedge s(\mathtt{b}) > 0 \wedge s(\mathtt{a}) = s(\mathtt{b}) \times s(\mathtt{q}) + s(\mathtt{r}) \\
&\qquad \implies s(\mathtt{r}) - s(\mathtt{b}) \geq 0 \wedge s(\mathtt{b}) > 0 \wedge s(\mathtt{a}) = s(\mathtt{b}) \times (s(\mathtt{q}) + 1) + (s(\mathtt{r}) - s(\mathtt{b}))
\end{aligned}$$

which are easy to prove by purely arithmetic reasoning.

2.4. *Further reading*

The material in this section follows Nipkow [41] (in HOL) and Bertot [11] (in Coq).

Separation logic [42,48] extends axiomatic semantics with a notion of local reasoning: assertions carry a *domain* (in our case, a set of variable; in pointer programs, a set of store locations) and the logic enforces that nothing outside the domain of the triple changes during execution. Examples of mechanized separation logics include Marti *et al.* [33] in Coq, Tuch *et al.* [50] in Isabelle/HOL, Appel and Blazy [5] in Coq, and Myreen and Gordon [40] in HOL4.

3. Compilation to a virtual machine

3.1. *The IMP virtual machine*

Instruction set: [instruction] [code]

$$
\begin{array}{lll}
i ::= & \texttt{const}(n) & \text{push } n \text{ on stack} \\
\mid & \texttt{var}(x) & \text{push value of } x \\
\mid & \texttt{setvar}(x) & \text{pop value and assign it to } x \\
\mid & \texttt{add} & \text{pop two values, push their sum} \\
\mid & \texttt{sub} & \text{pop two values, push their difference} \\
\mid & \texttt{branch}(\delta) & \text{unconditional jump} \\
\mid & \texttt{bne}(\delta) & \text{pop two values, jump if } \neq \\
\mid & \texttt{bge}(\delta) & \text{pop two values, jump if } \geq \\
\mid & \texttt{halt} & \text{end of program}
\end{array}
$$

In branch instructions, δ is an offset relative to the next instruction.

A state of the machine is composed of: [machine_state]

- A fixed code C (a list of instructions).
- A variable program counter pc (an integer position in C).
- A variable stack σ (a list of integers).
- A variable state s (mapping variables to integers).

The dynamic semantics of the machine is given by the following one-step transition relation [transition]. $C(pc)$ is the instruction at position pc in C, if any.

$$
\begin{array}{ll}
C \vdash (pc, \sigma, s) \rightarrow (pc+1, n.\sigma, s) & \text{if } C(pc) = \texttt{const}(n) \\
C \vdash (pc, \sigma, s) \rightarrow (pc+1, s.(x).\sigma, s) & \text{if } C(pc) = \texttt{var}(n) \\
C \vdash (pc, n.\sigma, s) \rightarrow (pc+1, \sigma, s[x \leftarrow n]) & \text{if } C(pc) = \texttt{setvar}(x) \\
C \vdash (pc, n_2.n_1.\sigma, s) \rightarrow (pc+1, (n_1+n_2).\sigma, s) & \text{if } C(pc) = \texttt{add} \\
C \vdash (pc, n_2.n_1.\sigma, s) \rightarrow (pc+1, (n_1-n_2).\sigma, s) & \text{if } C(pc) = \texttt{sub} \\
C \vdash (pc, \sigma, s) \rightarrow (pc+1+\delta, \sigma, s) & \text{if } C(pc) = \texttt{branch}(\delta) \\
C \vdash (pc, n_2.n_1.\sigma, s) \rightarrow (pc+1+\delta, \sigma, s) & \text{if } C(pc) = \texttt{bne}(\delta) \text{ and } n_1 \neq n_2 \\
C \vdash (pc, n_2.n_1.\sigma, s) \rightarrow (pc+1, \sigma, s) & \text{if } C(pc) = \texttt{bne}(\delta) \text{ and } n_1 = n_2 \\
C \vdash (pc, n_2.n_1.\sigma, s) \rightarrow (pc+1+\delta, \sigma, s) & \text{if } C(pc) = \texttt{bge}(\delta) \text{ and } n_1 \geq n_2 \\
C \vdash (pc, n_2.n_1.\sigma, s) \rightarrow (pc+1, \sigma, s) & \text{if } C(pc) = \texttt{bge}(\delta) \text{ and } n_1 < n_2
\end{array}
$$

As in section 1.2, the observable behavior of a machine program is defined by sequences of transitions:

- Termination $C \vdash (pc, \sigma, s) \Downarrow s'$ if $C \vdash (pc, \sigma, s) \xrightarrow{*} (pc', \sigma', s')$ and $C(pc') = \mathtt{halt}$.
- Divergence $C \vdash (pc, \sigma, s) \Uparrow$ if the machine makes infinitely many transitions from (pc, σ, s).
- Going wrong, otherwise.

3.2. The compilation scheme

The code $\mathtt{comp}(e)$ for an expression evaluates e and pushes its value on top of the stack. [compile_expr]

$$\begin{aligned}
\mathtt{comp}(x) &= \mathtt{var}(x)\\
\mathtt{comp}(n) &= \mathtt{const}(n)\\
\mathtt{comp}(e_1 + e_2) &= \mathtt{comp}(e_1); \mathtt{comp}(e_2); \mathtt{add}\\
\mathtt{comp}(e_1 - e_2) &= \mathtt{comp}(e_1); \mathtt{comp}(e_2); \mathtt{sub}
\end{aligned}$$

The code $\mathtt{comp}(b, \delta)$ for a boolean expression falls through if b is true, and branches to offset δ if b is false. [compile_bool_expr]

$$\begin{aligned}
\mathtt{comp}(e_1 = e_2,\ \delta) &= \mathtt{comp}(e_1); \mathtt{comp}(e_2); \mathtt{bne}(\delta)\\
\mathtt{comp}(e_1 < e_2,\ \delta) &= \mathtt{comp}(e_1); \mathtt{comp}(e_2); \mathtt{bge}(\delta)
\end{aligned}$$

The code $\mathtt{comp}(c)$ for a command c updates the state according to the semantics of c, while leaving the stack unchanged. [compile_cmd]

$$\begin{aligned}
\mathtt{comp}(\mathtt{skip}) &= \varepsilon\\
\mathtt{comp}(x := e) &= \mathtt{comp}(e); \mathtt{setvar}(x)\\
\mathtt{comp}(c_1; c_2) &= \mathtt{comp}(c_1); \mathtt{comp}(c_2)\\
\mathtt{comp}(\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2) &= \mathtt{comp}(b, |C_1| + 1); C_1; \mathtt{branch}(|C_2|); C_2\\
&\qquad \text{where } C_1 = \mathtt{comp}(c_1) \text{ and } C_2 = \mathtt{comp}(c_2)\\
\mathtt{comp}(\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done}) &= B; C; \mathtt{branch}(-(|B| + |C| + 1))\\
&\qquad \text{where } C = \mathtt{comp}(c) \text{ and } B = \mathtt{comp}(b, |C| + 1)
\end{aligned}$$

$|C|$ is the length of a list of instructions C. The mysterious offsets in branch instructions are depicted in figure 2.

Finally, the compilation of a program c is $\mathtt{compile}(c) = \mathtt{comp}(c); \mathtt{halt}$. [compile_program]

Combining the compilation scheme with the semantics of the virtual machine, we obtain a new way to execute a program c in initial state s: start the machine in code

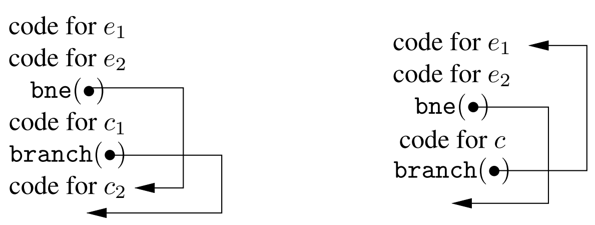

Figure 2. Shape of generated code for `if` $e_1 = e_2$ `then` c_1 `else` c_2 (left) and `while` $e_1 = e_2$ `do` c `done` (right)

$\texttt{comp}(c)$ and state $(0, \varepsilon, s)$ (program counter at first instruction of $\texttt{comp}(c)$; empty stack; state s), and observe its behavior. Does this behavior agree with the behavior of c predicted by the semantics of section 1?

3.3. Notions of semantic preservation

Consider two programs P_1 and P_2, possibly in different languages. (For example, P_1 is an IMP command and P_2 a sequence of VM instructions.) Under which conditions can we say that P_2 preserves the semantics of P_1?

- Bisimulation (equivalence):
  $P_1 \Downarrow s \Leftrightarrow P_2 \Downarrow s \quad \wedge \quad P_1 \Uparrow \; \Leftrightarrow P_2 \Uparrow \quad \wedge \quad P_1 \Downarrow \texttt{wrong} \Leftrightarrow P_2 \Downarrow \texttt{wrong}$
- Backward simulation (refinement):
  $P_2 \Downarrow s \Rightarrow P_1 \Downarrow s \quad \wedge \quad P_2 \Uparrow \; \Rightarrow P_1 \Uparrow \quad \wedge \quad P_2 \Downarrow \texttt{wrong} \Rightarrow P_1 \Downarrow \texttt{wrong}$
- Backward simulation for correct source programs:
  if $\neg(P_1 \Downarrow \texttt{wrong})$ then $\neg(P_2 \Downarrow \texttt{wrong}) \;\wedge\; P_2 \Downarrow s \Rightarrow P_1 \Downarrow s \;\wedge\; P_2 \Uparrow \; \Rightarrow P_1 \Uparrow$
- Forward simulation for correct source programs:
  if $\neg(P_1 \Downarrow \texttt{wrong})$ then $P_1 \Downarrow s \Rightarrow P_2 \Downarrow s \quad \wedge \quad P_1 \Uparrow \; \Rightarrow P_2 \Uparrow$

Lemma 22 *(Simulation and determinism.) If P_2 has deterministic semantics, then "forward simulation for correct programs" imply "backward simulation for correct programs".*

3.4. Semantic preservation for the compiler

To show semantic preservation between an IMP program and its compiled code, we prove a "forward simulation for correct programs" result. It is convenient to reason over the big-step operational semantics of the source program, since its compositional nature is a good match for the compositional nature of the compilation scheme. Therefore, we split the proof in two cases: (1) the source program terminates, and (2) it diverges.

Lemma 23 [compile_expr_correct] *For all instruction sequences C_1, C_2, stacks σ and states s,*

$$C_1; \texttt{comp}(e); C_2 \vdash (|C_1|, \sigma, s) \xrightarrow{*} (|C_1| + |\texttt{comp}(e)|, [\![e]\!]\ s.\sigma, s)$$

Lemma 24 [compile_bool_expr_correct] *For all instruction sequences C_1, C_2, stacks σ and states s,*

$$C_1; \texttt{comp}(b,\ \delta); C_2 \vdash (|C_1|, \sigma, s) \xrightarrow{*} (pc, \sigma, s)$$

with $pc = |C_1| + |\texttt{comp}(b)|$ *if* $[\![b]\!]\ s = \texttt{true}$ *and* $pc = |C_1| + |\texttt{comp}(b)| + \delta$ *otherwise.*

Theorem 25 [compile_cmd_correct_terminating] *Assume* $c, s \Rightarrow s'$. *Then, for all instruction sequences* C_1, C_2 *and stacks* σ,

$$C_1; \texttt{comp}(c); C_2 \vdash (|C_1|, \sigma, s) \xrightarrow{*} (|C_1| + |\texttt{comp}(c)|, \sigma, s')$$

The proof for the diverging case uses the following special-purpose coinduction principle.

Lemma 26 *Let* X *be a set of (machine code, machine state) pairs such that*

$$\forall (C, S) \in X,\ \exists S' \in X,\ C \vdash S \xrightarrow{+} S'.$$

Then, for all $(C, S) \in X$, *we have* $C \vdash S \Uparrow$ *(there exists an infinite sequence of transitions starting from* S*).*

The following theorem follows from the coinduction principle above applied to the set

$$X = \{(C_1; \texttt{comp}(c); C_2, (|C_1|, \sigma, s)) \mid c, s \Rightarrow \infty\}.$$

Theorem 27 [compile_cmd_correct_diverging] *Assume* $c, s \Rightarrow \infty$. *Then, for all instruction sequences* C_1, C_2 *and stacks* σ,

$$C_1; \texttt{comp}(c); C_2 \vdash (|C_1|, \sigma, s) \Uparrow$$

3.5. Further reading

The virtual machine used in this section matches a small subset of the Java Virtual Machine [32]. Other examples of mechanized verification of nonoptimizing compilers producing virtual machine code include Bertot [10] (for the IMP language), Klein and Nipkow [27] (for a subset of Java), and Grall and Leroy [30] (for call-by-value λ-calculus). The latter two show forward simulation results; Bertot shows both forward and backward simulation, and concludes that backward simulation is considerably more difficult to prove. Other examples of difficult backward simulation arguments (not mechanized) can be found in [22], for call-by-name and call-by-value λ-calculus.

Theorem 23 (correctness of compilation of arithmetic expression to stack machine code) is historically important: it is the oldest published compiler correctness proof (McCarthy and Painter [34], in 1967) and the oldest mechanized compiler correctness proof (Milner and Weyhrauch, [36], in 1972). Since then, a great many correctness proofs for compilers and compilation passes have been published, some of them being mechanized: Dave's bibliography [19] lists 99 references up to 2002.

4. An example of optimizing program transformation: dead code elimination

The purpose of dead code elimination is to remove assignments $x := e$ (turning them into `skip` instructions) such that the value of x is not used in the remainder of the program.

Example. Consider:

```
x := 1;  y := y + 1;  x := 2
```

The assignment `x := 1` can always be eliminated since `x` is not referenced before being redefined by `x := 2`.

To detect the fact that the value of a variable is not used later, we need a static analysis known as *liveness analysis*.

4.1. Liveness analysis

A variable is *dead* at a program point if its value is not used later in the execution of the program: either the variable is never mentioned again, or it is always redefined before further use. A variable is *live* if it is not dead.

Given a set a of variables live "after" a command c, the function $\texttt{live}(c, a)$ over-approximates the set of variables live "before" the command [live]. It proceeds by a form of reverse execution of c, conservatively assuming that conditional branches can go both ways. FV computes the set of variables referenced in an expression [fv_expr] [fv_bool_expr].

$$
\begin{aligned}
\texttt{live}(\texttt{skip}, a) &= a \\
\texttt{live}(x := e,\ a) &= \begin{cases} (a \setminus \{x\}) \cup FV(e) & \text{if } x \in a; \\ a & \text{if } x \notin a. \end{cases} \\
\texttt{live}((c_1, c_2),\ a) &= \texttt{live}(c_1, \texttt{live}(c_2, a)) \\
\texttt{live}((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2),\ a) &= FV(b) \cup \texttt{live}(c_1, a) \cup \texttt{live}(c_2, a) \\
\texttt{live}((\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}),\ a) &= \texttt{fix}(\lambda X.\ a \cup FV(b) \cup \texttt{live}(c, X))
\end{aligned}
$$

If F is a function from sets of variables to sets of variables, $\texttt{fix}(F)$ is supposed to compute a *post-fixpoint* of F, that is, a set X such that $F(X) \subseteq X$. Typically, F is iterated n times, starting from the empty set, until we reach an n such that $F^{n+1}(\emptyset) \subseteq F^n(\emptyset)$. Ensuring termination of such an iteration is, in general, a difficult problem. (See section 4.4 for discussion.) To keep things simple, we bound arbitrarily to N the number of iterations, and return a default overapproximation if a post-fixpoint cannot be found within N iterations: [fixpoint]

$$
\texttt{fix}(F, \mathit{default}) = \begin{cases} F^n(\emptyset) & \text{if } \exists n \leq N,\ F^{n+1}(\emptyset) \subseteq F^n(\emptyset); \\ \mathit{default} & \text{otherwise} \end{cases}
$$

Here, a suitable default is $a \cup FV(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done})$, the set of variables live "after" the loop or referenced within the loop.

$$\begin{aligned}\texttt{live}((\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}),\ a) = \texttt{fix}(\lambda X.\ a \cup FV(b) \cup \texttt{live}(c, X),\\ a \cup FV(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}))\end{aligned}$$

Lemma 28 [live_while_charact] *Let* $a' = \texttt{live}(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done},\ a)$. *Then:*

$$FV(b) \subseteq a' \qquad a \subseteq a' \qquad \texttt{live}(c, a') \subseteq a'$$

4.2. Dead code elimination

The program transformation that eliminates dead code is, then: [dce]

$$\begin{aligned}
\texttt{dce}(\texttt{skip}, a) &= \texttt{skip}\\
\texttt{dce}(x := e,\ a) &= \begin{cases} x := e & \text{if } x \in a;\\ \texttt{skip} & \text{if } x \notin a.\end{cases}\\
\texttt{dce}((c_1, c_2),\ a) &= \texttt{dce}(c_1, \texttt{live}(c_2, a)); \texttt{dce}(c_2, a)\\
\texttt{dce}((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2),\ a) &= \texttt{if}\ b\ \texttt{then}\ \texttt{dce}(c_1, a)\ \texttt{else}\ \texttt{dce}(c_2, a)\\
\texttt{dce}((\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}),\ a) &= \texttt{while}\ b\ \texttt{do}\ \texttt{dce}(c, a)\ \texttt{done}
\end{aligned}$$

Example. Consider again the "Euclidean division" program c:

```
r := a; q := 0;  while b < r+1 do r := r - b; q := q + 1 done
```

If q is not live "after" (q $\notin a$), it is not live throughout this program either. Therefore, $\texttt{dce}(c, a)$ produces

```
r := a; skip;    while b < r+1 do r := r - b; skip        done
```

The useless computations of q have been eliminated entirely, in a process similar to program slicing.

4.3. Correctness of the transformation

We show a "forward simulation for correct programs" property:

- If $c, s \Downarrow s'$, then $\texttt{dce}(c, a), s \Downarrow s''$ for some s'' related to s'.
- If $c, s \Uparrow$, then $\texttt{dce}(c, a), s \Uparrow$.

However, the program $\texttt{dce}(c, a)$ performs fewer assignments than c, therefore the final states can differ on the values of dead variables. We define agreement between two states s, s' with respect to a set of live variables a: [agree]

$$s \approx s' : a \ \stackrel{\text{def}}{=}\ \forall x \in a,\ s(x) = s'(x)$$

Lemma 29 [eval_expr_agree] [eval_bool_expr_agree] *Assume* $s \approx s' : a$. *If* $FV(e) \subseteq a$, *then* $[\![e]\!]\ s = [\![e]\!]\ s'$. *If* $FV(b) \subseteq a$, *then* $[\![b]\!]\ s = [\![b]\!]\ s'$.

Lemma 30 [agree_update_live] *(Assignment to a live variable.) If $s \approx s' : a \setminus \{x\}$, then $s[x \leftarrow v] \approx s'[x \leftarrow v] : a$.*

Lemma 31 [agree_update_dead] *(Assignment to a dead variable.) If $s \approx s' : a$ and $x \notin a$, then $s[x \leftarrow v] \approx s' : a$.*

Theorem 32 [dce_correct_terminating] *If $c, s \Rightarrow s'$ and $s \approx s_1 : \texttt{live}(c, a)$, then there exists s'_1 such that $\texttt{dce}(c, a), s_1 \Rightarrow s'_1$ and $s' \approx s'_1 : a$.*

Theorem 33 [dce_correct_diverging] *If $c, s \Rightarrow \infty$ and $s \approx s_1 : \texttt{live}(c, a)$, then $\texttt{dce}(c, a), s_1 \Rightarrow \infty$.*

4.4. Further reading

Dozens of compiler optimizations are known, each targeting a particular class of inefficiencies. See Appel [3] for an introduction to optimization, and Muchnick [39] for a catalogue of classic optimizations.

The results of liveness analysis can be exploited to perform register allocation (a crucial optimization performance-wise), following Chaitin's approach [17] [3, chap. 11]: coloring of an interference graph. A mechanized proof of correctness for graph coloring-based register allocation, extending the proof given in this section, is described by Leroy [29,28].

Liveness analysis is an instance of a more general class of static analyses called *dataflow analyses* [3, chap. 17], themselves being a special case of abstract interpretation. Bertot *et al.* [14] and Leroy [28] prove, in Coq, the correctness of several optimizations based on dataflow analyses, such as constant propagation and common subexpression elimination. Cachera *et al.* [16] present a reusable Coq framework for dataflow analyses.

Dataflow analyses are generally carried on an unstructured representation of the program called the control-flow graph. Dataflow equations are set up between the nodes of this graph, then solved by one global fixpoint iteration, often based on Kildall's worklist algorithm [25]. This is more efficient than the approach we described (computing a local fixpoint for each loop), which can be exponential in the nesting degree of loops. Kildall's worklist algorithm has been mechanically verified many times [14,18,27].

The effective computation of fixpoints is a central issue in static analysis. Theorems such as Knaster-Tarski's show the existence of fixpoints in many cases, and can be mechanized [44,15], but fail to provide effective algorithms. Noetherian recursion can be used if the domain of the analysis is well founded (no infinite chains) [13, chap. 15], but this property is difficult to ensure in practice [16]. The shortcut we took in this section (bounding arbitrarily the number of iterations) is inelegant but a reasonable engineering compromise.

5. State of the art and perspectives

Some recent achievements using mechanized semantics (in reverse chronological order):

- The verification of the seL4 secure micro-kernel `http://nicta.com.au/research/projects/l4.verified/` [26].

- The CompCert verified compiler: a realistic, moderately-optimizing compiler for a large subset of the C language down to PowerPC and ARM assembly code. http://compcert.inria.fr/ [29].
- The Verisoft project http://www.verisoft.de/, which aims at the end-to-end formal verification of a complete embedded system, from hardware to application.
- Formal specifications of the Java / Java Card virtual machines and mechanized verifications of the Java bytecode verifier: Ninja [27], Jakarta [7], Bicolano http://mobius.inria.fr/twiki/bin/view/Bicolano, and the Kestrel Institute project http://www.kestrel.edu/home/projects/java/.
- Formal verification of the ARM6 processor micro-architecture against the ARM instruction set specification [21]
- The "foundational" approach to Proof-Carrying Code [4].
- The CLI stack: a formally verified microprocessor and compiler from an assembly-level language http://www.cs.utexas.edu/~moore/best-ideas/piton/index.html [37].

Some active research topics in this area:

- Combining static analysis and program proof. Static analysis can be viewed as the automatic generation of logical assertions, enabling the results of static analysis to be verified *a posteriori* using a program logic, and facilitating the annotation of existing code with logical assertions.
- Proof-preserving compilation. Given a source program annotated with assertions and a proof in axiomatic semantics, can we produce machine code annotated with the corresponding assertions and the corresponding proof? [8,31].
- A major obstacle to the mechanization of rich language semantics and advanced type systems is the handling of bound variables and the fact that terms containing binders are equal modulo α-conversion of bound variables. The POPLmark challenge explores this issue [6] http://plclub.org/mmm/.
- Shared-memory concurrency raises major semantic difficulties, ranging from formalizing the "weakly-consistent" memory models implemented by today's multicore processors [49] to mechanizing program logics appropriate for proving concurrent programs correct [20,23].
- Progressing towards fully-verified development and verification environments for high-assurance software. Beyond verifying compilers and other code generation tools, we'd like to gain formal assurance in the correctness of program verification tools such as static analyzers and program provers.

References

[1] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logics and the Foundations of Mathematics*, pages 739–782. North-Holland, 1997.

[2] Sten Agerholm. Domain theory in HOL. In *Higher Order Logic Theorem Proving and its Applications, Workshop HUG '93*, volume 780 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 1994.

[3] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[4] Andrew W. Appel. Foundational proof-carrying code. In *Logic in Computer Science 2001*, pages 247–258. IEEE Computer Society Press, 2001.

[5] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.

[6] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[7] G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa. Tool-Assisted Specification and Verification of the JavaCard Platform. In *Proceedings of AMAST'02*, volume 2422 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2002.

[8] Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate translation for optimizing compilers. In *Static Analysis, 13th Int. Symp., SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2006.

[9] Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in Coq. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*. Springer, 2009. To appear.

[10] Yves Bertot. A certified compiler for an imperative language. Research report RR-3488, INRIA, 1998.

[11] Yves Bertot. Theorem proving support in programming language semantics. Research Report RR-6242, INRIA, 2007.

[12] Yves Bertot. Coq in a hurry. Tutorial available at http://cel.archives-ouvertes.fr/inria-00001173, October 2008.

[13] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer, 2004.

[14] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs, Workshop TYPES 2004*, volume 3839 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2006.

[15] Yves Bertot and Vladimir Komendantsky. Fixed point semantics and partial recursion in Coq. In *10th int. conf. on Principles and Practice of Declarative Programming (PPDP 2008)*, pages 89–96. ACM Press, 2008.

[16] David Cachera, Thomas P. Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

[17] Gregory J. Chaitin. Register allocation and spilling via graph coloring. In *Symposium on Compiler Construction*, volume 17(6) of *SIGPLAN Notices*, pages 98–105. ACM Press, 1982.

[18] Solange Coupet-Grimal and William Delobel. A uniform and certified approach for two static analyses. In *Types for Proofs and Programs, Workshop TYPES 2004*, volume 3839 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2006.

[19] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003.

[20] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007.

[21] Anthony C. J. Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40. Springer, 2003.

[22] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtimes within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.

[23] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2008.

[24] Gilles Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–257. Elsevier, 1988.

[25] Gary A. Kildall. A unified approach to global program optimization. In *1st symposium Principles of Programming Languages*, pages 194–206. ACM Press, 1973.

[26] Gerwin Klein. Operating system verification — an overview. *Sadhana*, 34(1):27–69, 2009.

[27] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[28] Xavier Leroy. A formally verified compiler back-end. arXiv:0902.2137 [cs]. Submitted, July 2008.

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

[30] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[31] Guodong Li, Scott Owens, and Konrad Slind. Structure of a proof-producing compiler for a subset of higher order logic. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2007.

[32] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1999. Second edition.

[33] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In *Formal Methods and Software Engineering, 8th Int. Conf. ICFEM 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. Springer, 2006.

[34] John McCarthy and James Painter. Correctness of a compiler for arithmetical expressions. In *Mathematical Aspects of Computer Science*, volume 19 of *Proc. of Symposia in Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.

[35] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (revised)*. The MIT Press, 1997.

[36] R[obin] Milner and R[ichard] Weyhrauch. Proving compiler correctness in a mechanized logic. In Bernard Meltzer and Donald Michie, editors, *Proc. 7th Annual Machine Intelligence Workshop*, volume 7 of *Machine Intelligence*, pages 51–72. Edinburgh University Press, 1972.

[37] J. S. Moore. *Piton: a mechanically verified assembly-language*. Kluwer, 1996.

[38] Peter D. Mosses. Denotational semantics. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B*, pages 577–631. The MIT Press/Elsevier, 1990.

[39] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[40] Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 568–582. Springer, 2007.

[41] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics. *Formal Aspects of Computing*, 10(2):171–186, 1998.

[42] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th Int. Workshop, CSL 2001*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[43] Christine Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors, *From Semantics to Computer Science: Essays in Honor of Gilles Kahn*. Cambridge University Press, to appear.

[44] Lawrence C. Paulson. Set theory for verification. II: Induction and recursion. *Journal of Automated Reasoning*, 15(2):167–215, 1995.

[45] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[46] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[47] John Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[48] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society Press, 2002.

[49] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *36th symposium Principles of Programming Languages*, pages 379–391. ACM Press, 2009.

[50] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *34th symposium Principles of Programming Languages*, pages 97–108. ACM Press, 2007.

[51] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.