Mechanized semantics

with applications to program proof and compiler verification

Xavier LEROY
INRIA Paris-Rocquencourt

Abstract. The goal of this lecture is to show how modern theorem provers—in this case, the Coq proof assistant—can be used to mechanize the specification of programming languages and their semantics, and to reason over individual programs and over generic program transformations, as typically found in compilers. The topics covered include: operational semantics (small-step, big-step, definitional interpreters); a simple form of denotational semantics; axiomatic semantics and Hoare logic; generation of verification conditions, with application to program proof; compilation to virtual machine code and its proof of correctness; an example of an optimizing program transformation (dead code elimination) and its proof of correctness.

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 [39].) 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 (sections 1 and 2);

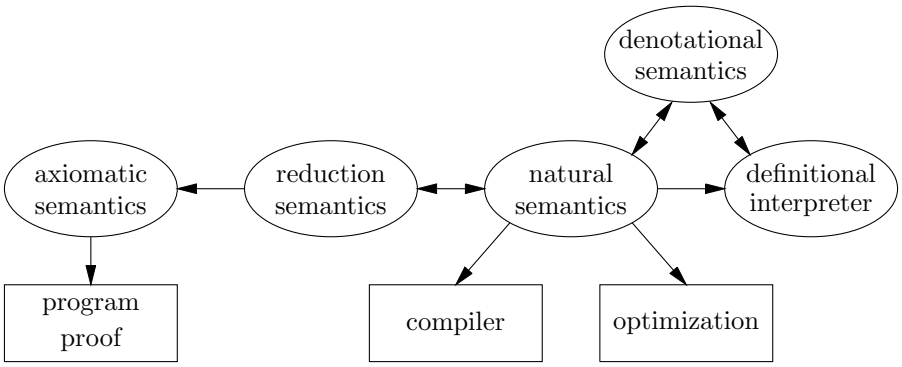

Figure 1. The various styles of semantics considered in this lecture and their uses. A double arrow denotes a semantic equivalence result. A single arrow from A to B means that semantics A is used to justify the correctness of B.

- introduce axiomatic semantics (Hoare logic) and show how to provide machine assistance for proving IMP programs using a verification condition generator (section 3);
- 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 4);
- illustrate optimizing compilation through the development and proof of correctness of a dead code elimination pass (section 5).

We finish with examples of recent achievements and ongoing challenges in this area (section 6).

We use the Coq proof assistant to specify semantics and program transformations, and conduct all proofs. An excellent text to learn Coq is *Software Foundations* by Pierce *et al* [50], but for the purposes of this lecture, Bertot's short tutorial [11] 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/VTSA-2013/`. These notes explain and 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 the present document, available at the Web site above, these notes are hyperlinks pointing directly to the corresponding Coq definitions and theorems in the development.

1. Symbolic expressions

1.1. Syntax

As a warm-up exercise, we start by formalizing the syntax and semantics of a simple language of expressions comprising variables (x, y, ...), integer constants n, and two operators $+$ and $-$.

Expressions: [expr]

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

The Coq representation of expressions is as a *inductive type*, similar to an ML or Haskell datatype.

```
Definition ident := nat.
Inductive expr : Type :=
  | Evar: ident -> expr
  | Econst: Z -> expr
  | Eadd: expr -> expr -> expr
  | Esub: expr -> expr -> expr.
```

`nat` and `Z` are predefined types for natural numbers and integers, respectively. Each case of the inductive type is a function that constructs terms of type `expr`. For instance, `Evar` applied to the name of a variable produces the representation of the corresponding expression; and `Eadd` applied to the representations of two subexpressions e_1 and e_2 returns the representation of the expression $e_1 + e_2$. Moreover, all terms of type `expr` are finitely generated by repeated applications of the 4 constructor functions; this enables definitions by pattern matching and reasoning by case analysis and induction.

1.2. Denotational semantics

The simplest and perhaps most natural way to specify the semantics of this language is as a function $[\![e]\!]\ s$ that associates an integer value to the expression e in the state s. States associate values to variables.

$$[\![x]\!]\ s = s(x) \qquad [\![n]\!]\ s = n$$

$$[\![e_1 + e_2]\!]\ s = [\![e_1]\!]\ s + [\![e_2]\!]\ s \qquad [\![e_1 - e_2]\!]\ s = [\![e_1]\!]\ s - [\![e_2]\!]\ s$$

In Coq, this denotational semantics is presented as a recursive function [eval_expr].

```
Definition state := ident -> Z.
Fixpoint eval_expr (s: state) (e: expr) {struct e} : Z :=
  match e with
  | Evar x => s x
  | Econst n => n
  | Eadd e1 e2 => eval_expr s e1 + eval_expr s e2
  | Esub e1 e2 => eval_expr s e1 - eval_expr s e2
  end.
```

`Fixpoint` marks a recursive function definition. The `struct e` annotation states that it is structurally recursive on its `e` parameter, and therefore guaranteed to terminate. The `match...with` construct represents pattern-matching on the shape of the expression `e`.

1.3. Using the denotational semantics

The `eval_expr` function can be used as an interpreter, to evaluate expressions in a known environment. For example:

```
Eval compute in (
  let x : ident := 0 in
  let s : state := fun y => if eq_ident y x then 12 else 0 in
  eval_expr s (Eadd (Evar x) (Econst 1))).
```

Coq prints "`13 : Z`". For additional performance, efficient executable Caml code can also be generated automatically from the Coq definition of `eval_expr` using the extraction mechanism of Coq.

Another use of `eval_expr` is to reason symbolically over expressions in arbitrary states. Consider the following claim:

```
Remark expr_add_pos:
  forall s x,
  s x >= 0 -> eval_expr s (Eadd (Evar x) (Econst 1)) > 0.
```

Using the `simpl` tactic of Coq, the goal reduces to a purely arithmetic statement:

```
  forall s x, s x >= 0 -> s x + 1 > 0.
```

which can be proved by standard arithmetic (the `omega` tactic).

Finally, the denotation function `eval_expr` can also be used to prove "meta" properties of the semantics. For example, we can easily show that the denotation of an expression is insensitive to values of variables not mentioned in the expression.

```
Lemma eval_expr_domain:
  forall s1 s2 e,
  (forall x, is_free x e -> s1 x = s2 x) ->
  eval_expr s1 e = eval_expr s2 e.
```

The proof is a simple induction on the structure of `e`. The predicate `is_free`, stating whether a variable occurs in an expression, is itself defined as a recursive function:

```
Fixpoint is_free (x: ident) (e: expr) {struct e} : Prop :=
  match e with
  | Evar y => x = y
  | Econst n => False
  | Eadd e1 e2 => is_free x e1 \/ is_free x e2
  | Esub e1 e2 => is_free x e1 \/ is_free x e2
  end.
```

As the `Prop` annotation indicates, the result of this function is not a data type but a logical formula.

1.4. Variants

The denotational semantics we gave above interprets the `+` and `-` operators as arithmetic over mathematical integer. We can easily interpret them differently, for instance as signed, modulo 2^{32} arithmetic (as in Java):

```
Fixpoint eval_expr (s: state) (e: expr) {struct e} : Z :=
  match e with
  | Evar x => s x
  | Econst n => normalize n
  | Eadd e1 e2 => normalize (eval_expr s e1 + eval_expr s e2)
  | Esub e1 e2 => normalize (eval_expr s e1 - eval_expr s e2)
  end.
```

Here, `normalize` n is n reduced modulo 2^{32} to the interval $[-2^{31}, 2^{31})$.

We can also account for undefined expressions. In practical programming languages, the value of an expression can be undefined for several reasons: if it mentions a variable that was not previously defined; in case of overflow during an arithmetic operation; in case of an integer division by 0; etc. A simple way to account for undefinedness is to use the `option` type, as defined in Coq's standard library. This is a two-constructor inductive type with `None` meaning "undefined" and `Some` n meaning "defined and having value n".

```
Definition state := ident -> option Z.
Fixpoint eval_expr (s: state) (e: expr) {struct e} : option Z :=
  match e with
  | Evar x => s x
  | Econst n => Some n
  | Eadd e1 e2 =>
      match eval_expr s e1, eval_expr s e2 with
      | Some n1, Some n2 => Some (n1 + n2)
      | _, _ => None
      end
  | Esub e1 e2 =>
      match eval_expr s e1, eval_expr s e2 with
      | Some n1, Some n2 => Some (n1 - n2)
      | _, _ => None
      end
  end.
```

1.5. Summary

The approach we followed in this section—denotational semantics represented as a Coq recursive function—is natural and convenient, but limited by a fundamental aspect of Coq: all functions must be terminating, so that they are defined everywhere by construction. The termination guarantee can come either by the fact that they are structurally recursive (recursive calls are only done on strict sub-terms of the argument, as in the case of `eval_expr`), or by Noetherian recursion on a well-founded ordering. Consequently, the approach followed in this section cannot be used to give semantics to languages featuring general loops or

general recursion. As we now illustrate with the IMP language, we need to move away from functional presentations of the semantics (where a function computes a result given a state and a term) and adopt relational presentations instead (where a ternary predicate relates a state, a term, and a result).

2. The IMP language and its semantics

2.1. Syntax

The IMP language is a very simple imperative language with structured control. Syntactically, it extends the language of expressions from section 1 with boolean expressions (conditions) and commands (statements):

Expressions: [expr]

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

Boolean expressions: [bool_expr]

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

Commands: [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 boolean expressions is given in the denotational style of section 1, as a function from states to booleans [eval_bool_expr].

$$[\![e_1 = e_2]\!]\ s = \begin{cases} \texttt{true} & \text{if } [\![e_1]\!]\ s = [\![e_2]\!]\ s; \\ \texttt{false} & \text{otherwise.} \end{cases}$$

$$[\![e_1 < e_2]\!]\ s = \begin{cases} \texttt{true} & \text{if } [\![e_1]\!]\ s < [\![e_2]\!]\ s; \\ \texttt{false} & \text{otherwise.} \end{cases}$$

2.2. Reduction semantics

A standard 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" [51], 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]

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

$$\frac{(c_1, s) \to (c_1', s)}{((c_1; c_2),\ s) \to ((c_1'; c_2),\ s')} \quad \text{[red_seq_left]} \qquad ((\texttt{skip}; c),\ s) \to (c, s) \quad \text{[red_seq_skip]}$$

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

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

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

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

The Coq translation of such a definition by inference rules is called an inductive predicate. Such predicates build on the same inductive definition mechanisms that we already use to represent abstract syntax trees, but the resulting logical object is a proposition (sort `Prop`) instead of a data type (sort `Type`).

The general recipe for translating inference rules to an inductive predicate is as follows. First, write each axiom and rule as a proper logical formula, using implications and universal quantification over free variables. For example, the rule `red_seq_left` becomes

```
forall c1 c2 s c1' s',
    red (c1, s) (c1', s') ->
    red (Cseq c1 c2, s) (Cseq c1' c2, s')
```

Second, give a name to each rule. (These names are called "constructors", by analogy with data type constructors.) Last, wrap these named rules in an inductive predicate definition like the following.

```
Inductive red: (cmd * state) -> (cmd * state) -> Prop :=
  | red_assign: forall x e s,
      red (Cassign x e, s) (Cskip, update s x (eval_expr s e))
  | red_seq_left: forall c1 c2 s c1' s',
      red (c1, s) (c1', s') ->
      red (Cseq c1 c2, s) (Cseq c1' c2, s')
  | red_seq_skip: forall c s,
      red (Cseq Cskip c, s) (c, s)
  | red_if_true: forall s b c1 c2,
      eval_bool_expr s b = true ->
      red (Cifthenelse b c1 c2, s) (c1, s)
  | red_if_false: forall s b c1 c2,
      eval_bool_expr s b = false ->
      red (Cifthenelse b c1 c2, s) (c2, s)
  | red_while_true: forall s b c,
      eval_bool_expr s b = true ->
      red (Cwhile b c, s) (Cseq c (Cwhile b c), s)
  | red_while_false: forall b c s,
      eval_bool_expr s b = false ->
      red (Cwhile b c, s) (Cskip, s).
```

Each constructor of the definition is a theorem that lets us conclude `red` (c, s) (c', s') when the corresponding premises hold. Moreover, the proposition `red` (c, s) (c', s') holds only if it was derived by applying these theorems a finite number of times (smallest fixpoint). This provides us with powerful reasoning principles: by case analysis on the last rule used, and by induction on a derivation. Consider for example the determinism of the reduction relation:

```
Lemma red_deterministic:
  forall cs cs1, red cs cs1 -> forall cs2, red cs cs2 -> cs1 = cs2.
```

It is easily proved by induction on a derivation of `red cs cs1` and a case analysis on the last rule used to conclude `red cs cs2`.

From the one-step reduction relation, we can define the 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', written $(c, s) \Downarrow s'$: finite sequence of reductions to `skip`. [terminates]

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

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

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

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

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

2.3. Natural semantics

An alternative to structured operational semantics is Kahn's natural semantics [27], 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.

To build intuitions for natural semantics, consider a terminating reduction sequence for the command $c; c'$.

$$((c; c'),\ s \rightarrow ((c_1; c'),\ s_1) \rightarrow \cdots \rightarrow ((\texttt{skip}; c'),\ s_2) \rightarrow (c',\ s_2) \rightarrow \cdots \rightarrow (\texttt{skip},\ s_3)$$

It contains a terminating reduction sequence for c, of the form $(c, s) \xrightarrow{*} (\texttt{skip}, s_2)$, followed by another terminating sequence for (c', s_2).

The idea of natural semantics is to write inference rules that follow this structure and define a predicate $c, s \Rightarrow s'$, meaning "in initial state s, the command c terminates with final state s' ". [exec]

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

$$\frac{c_1, s \Rightarrow s_1 \qquad c_2, s_1 \Rightarrow s_2}{c_1; c_2, s \Rightarrow s_2} \quad \text{[exec\_seq]}$$

$$\frac{c_1, s \Rightarrow s' \text{ if } [\![b]\!]\ s = \texttt{true} \qquad c_2, s \Rightarrow s' \text{ if } [\![b]\!]\ s = \texttt{false}}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2), s \Rightarrow s'} \quad \text{[exec\_if]}$$

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

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

We now have two different semantics for the same language. A legitimate question to ask is whether they are equivalent: do both semantics predict the same "terminates / diverges / goes wrong" behaviors for any given program? Such an equivalence result strengthens the confidence we have in both semantics. Moreover, it enables us to use whichever semantics is more convenient to prove a property of interest. We first show an implication from natural semantics to terminating reduction sequences.

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

The proof is a straightforward induction on a derivation of $c, s \Rightarrow s'$ and case analysis on the last rule used. Here is a representative case: $c = c_1; c_2$. By hypothesis, $c_1; c_2, s \Rightarrow s'$. By inversion, we know that $c_1, s \Rightarrow s_1$ and $c_2, s_1 \Rightarrow s'$ for some intermediate state s_1. Applying the induction hypothesis twice, we obtain $(c_1, s) \xrightarrow{*} (\texttt{skip}, s_1)$ and $(c_2, s_1) \xrightarrow{*} (\texttt{skip}, s')$. A context lemma (proved separately by induction) shows that $((c_1; c_2), s) \xrightarrow{*} ((\texttt{skip}; c_2), s_1)$. To obtain the expected result, all we need to do is to assemble the reduction sequences together, using the transitivity of $\xrightarrow{*}$:

$$((c_1; c_2), s) \xrightarrow{*} ((\texttt{skip}; c_2), s_1) \to (c_2, s_1) \xrightarrow{*} (\texttt{skip}, s')$$

The converse implication (from terminating reduction sequences to natural semantics) is more difficult. The idea is to consider mixed executions that start with some reduction steps and finish in one big step using the natural semantics:

$$(c_1, s_1) \to \cdots \to (c_i, s_i) \Rightarrow s'$$

We first show that the last reduction step can always be "absorbed" by the final big step:

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

Combining this lemma with an induction on the sequence of reduction, we obtain the desired semantic implication:

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

2.4. Natural semantics for divergence

Kahn-style natural semantics correctly characterize programs that terminate, either normally (as in section 2.3) or by going wrong (through the addition of so-called error rules). For a long time it was believed that natural semantics is un-

able to account for divergence. As observed by Grall and Leroy [33], this is not true: diverging executions can also be described in the style of natural semantics, provided a coinductive definition (greatest fixpoint) is used. Define the infinite execution relation $c, s \Rightarrow \infty$ (from initial state s, the command c diverges). [execinf]

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

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

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

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

As denoted by the double horizontal bars, these rules must be interpreted *coinductively* as a greatest fixpoint [33, section 2]. Equivalently, the coinductive interpretation corresponds to conclusions of *possibly infinite* derivation trees, while the inductive interpretation corresponds to *finite* derivation trees. Coq provides built-in support for coinductive definitions of data types and predicates.

As in section 2.3 and perhaps even more so here, we need to prove an equivalence between the $c, s \Rightarrow \infty$ predicate and the existence of infinite reduction sequences. One implication follows from the decomposition lemma below:

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

A simple argument by coinduction, detailed in [33], then concludes the expected implication:

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

The reverse implication uses two inversion lemmas:

- If $((c_1; c_2),\ s) \Uparrow$, either $(c_1, s) \Uparrow$ or there exists s' such that $(c_1, s) \xrightarrow{*} (\texttt{skip}, s')$ and $(c_2, s') \Uparrow$.
- If $(\texttt{while } b \texttt{ do } c \texttt{ done},\ s) \Uparrow$, then $[\![b]\!]\ s = \texttt{true}$ and either $(c, s) \Uparrow$ or there exists s' such that $(c, s) \xrightarrow{*} (\texttt{skip}, s')$ and $(\texttt{while } b \texttt{ do } c \texttt{ done},\ s') \Uparrow$

These lemmas follow from determinism of the $\to$ relation and the seemingly obvious fact that any reduction sequence is either infinite or stops, after finitely many reductions, on an irreducible configuration:

$$\forall c, s, \quad (c, s) \Uparrow \ \vee\ \exists c', \exists s', \ (c, s) \xrightarrow{*} (c', s') \wedge (c', s') \not\to$$

The property above cannot be proved in Coq's constructive logic: such a constructive proof would be, in essence, a program that decides the halting problem. However, we can add the law of excluded middle ($\forall P,\ P \vee \neg P$) to Coq as an axiom, without breaking logical consistency. The fact above can easily be proved from the law of excluded middle.

Theorem 6 [diverges_execinf] *If* $(c, s) \Uparrow$, *then* $c, s \Rightarrow \infty$.

2.5. Definitional interpreter

As mentioned at the end of section 1, we cannot write a Coq function with type $\texttt{cmd} \to \texttt{state} \to \texttt{state}$ that would execute a command and return its final state whenever the command terminates: this function would not be total. We can, however, define a Coq function $\mathcal{I}(n, c, s)$ that executes c in initial state s, taking as extra argument a natural number n used to bound the amount of computation performed. This function 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 $\rhd$, reminiscent of monads in functional programming, is defined by $\bot \rhd f = \bot$ and $\lfloor s \rfloor \rhd f = f(s)$.

A crucial property of this definitional interpreter is that it is monotone with respect to the maximal recursion depth n. Evaluation results are ordered by taking $\bot \leq \lfloor s \rfloor$ [res_le].

Lemma 7 [interp_mon] *(Monotonicity of* $\mathcal{I}$*.) If* $n \leq m$, *then* $\mathcal{I}(n, c, s) \leq \mathcal{I}(m, c, s)$.

Exploiting this property, we can show partial correctness results of the definitional interpreter with respect to the natural semantics:

Lemma 8 [interp_exec] *If* $\mathcal{I}(n, c, s) = \lfloor s' \rfloor$, *then* $c, s \Rightarrow s'$.

Lemma 9 [exec_interp] *If* $c, s \Rightarrow s'$, *there exists an* n *such that* $\mathcal{I}(n, c, s) = \lfloor s' \rfloor$.

Lemma 10 [execinf_interp] *If* $c, s \Rightarrow \infty$, *then* $\mathcal{I}(n, c, s) = \bot$ *for all* n.

2.6. Denotational semantics

A simple form of denotational semantics [42] can be obtained by "letting n goes to infinity" in the definitional interpreter.

Lemma 11 [interp_limit_dep] *For every c, there exists a function $[\![c]\!]$ from states to evaluation results such that $\forall s,\ \exists m,\ \forall n \geq m,\ \mathcal{I}(n, c, s) = [\![c]\!]\ s$.*

Again, this result cannot be proved in Coq's constructive logic and requires the axiom of excluded middle and an axiom of description.

This denotation function $[\![c]\!]$ satisfies the equations of denotational semantics:

$$[\![\texttt{skip}]\!]\ s = \lfloor s \rfloor$$

$$[\![x := e]\!]\ s = \lfloor s[x \leftarrow [\![e]\!]\ s] \rfloor$$

$$[\![c_1; c_2]\!]\ s = [\![c_1]\!]\ s \rhd (\lambda s'.\ [\![c_2]\!]\ s')$$

$$[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\ s = [\![c_1]\!]\ s \text{ if } [\![b]\!]\ s = \texttt{true}$$

$$[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\ s = [\![c_2]\!]\ s \text{ if } [\![b]\!]\ s = \texttt{false}$$

$$[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}]\!]\ s = \lfloor s \rfloor \text{ if } [\![b]\!]\ s = \texttt{false}$$

$$[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}]\!]\ s = [\![c]\!]\ s \rhd (\lambda s'.\ [\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}]\!]\ s') \text{ if } [\![b]\!]\ s = \texttt{true}$$

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

Using these properties of $[\![c]\!]$, we can show full equivalence between the denotational and natural semantics.

Theorem 12 [denot_exec] [exec_denot] *$c, s \Rightarrow s'$ if and only if $[\![c]\!]\ s = \lfloor s' \rfloor$.*

Theorem 13 [denot_execinf] [execinf_denot] *$c, s \Rightarrow \infty$ if and only if $[\![c]\!]\ s = \bot$.*

2.7. Further reading

The material presented in this section is inspired by Nipkow [45] (in Isabelle/HOL, for the IMP language) and by Grall and Leroy [33] (in Coq, for the call-by-value λ-calculus).

We followed Plotkin's "SOS" presentation [51] 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 [56] and is very popular to reason about type systems [49].

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 [33] connect the two approaches.

The definitional interpreter approach was identified by Reynolds in 1972. See [52] 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 [47] (in Coq) and Benton *et al.* [9] (in Coq).

3. Axiomatic semantics and program verification

Operational semantics as in section 2 focuses on describing actual executions of programs. In contrast, axiomatic semantics (also called Hoare logic) 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.

3.1. Weak Hoare triples and their rules

Following Hoare's seminal work [25], we consider logical formulas of the form $\{\, P \,\}\ c\ \{\, Q \,\}$, meaning "if precondition P holds, the command c does not go wrong, and if it terminates, the postcondition Q holds". Here, P and Q are arbitrary predicates over states. A formula $\{\, P \,\}\ c\ \{\, Q \,\}$ is called a *weak Hoare triple* (by opposition with strong Hoare triples discussed in section 3.4, which guarantee termination as well). We first define some useful operations over predicates:

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

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

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

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

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

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

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

Example. The triple $\{\, a = bq + r \,\}\ r := r - b; q := q + 1\ \{\, a = bq + r \,\}$ is derivable from rules `triple_assign`, `triple_seq` and `triple_consequence` because the following logical equivalences hold:

$$(a = bq + r)[q \leftarrow q + 1] \Longleftrightarrow a = b(q + 1) + r$$
$$(a = b(q + 1) + r)[r \leftarrow r - b] \Longleftrightarrow a = b(q + 1) + (r - b) = bq + r$$

3.2. Soundness of the axiomatic semantics

Intuitively, a weak 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]

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

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

In an inductive interpretation, rule `finally_step` could only be applied a finite number of steps, and therefore (c, s) `finally` Q would be equivalent to $\exists s',\ (c, s) \xrightarrow{*} (\texttt{skip}, s') \wedge Q(s')$. In the coinductive interpretation, rule `finally_step` can also be applied infinitely many times, capturing diverging executions as well.

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

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

We now proceed to show that if $\{\, P \,\}\ c\ \{\, Q \,\}$ is derivable, the proposition $[\![\{\, P \,\}\ c\ \{\, Q \,\}]\!]$ above holds. We start by some lemmas about the `finally` predicate.

Lemma 14 [finally_seq] *If* (c_1, s) `finally` Q *and* $[\![\{\, Q \,\}\ c_2\ \{\, R \,\}]\!]$*, then* $((c_1; c_2), s)$ `finally` R*.*

Lemma 15 [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 16 [finally_consequence] *If* (c, s) `finally` Q *and* $Q \Longrightarrow Q'$*, then* (c, s) `finally` Q'*.*

We can then prove the expected soundness result by a straightforward induction on a derivation of $\{\, P \,\}\ c\ \{\, Q \,\}$:

Theorem 17 [triple_correct] *If* $\{\, P \,\}\ c\ \{\, Q \,\}$ *can be derived by the rules of axiomatic semantics, then* $[\![\{\, P \,\}\ c\ \{\, Q \,\}]\!]$ *holds.*

3.3. 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 $\texttt{assert}(P)$ to $\texttt{skip}$. [erase]

The $\texttt{wp}$ 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 $\texttt{vcg}$ function (verification condition generator) computes a conjunction of implications that must hold for the triple $\{\, \texttt{wp}(c, Q) \,\}\ c\ \{\, Q \,\}$ to hold. [vcg]

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

Lemma 18 [vcg_correct] *If* $\texttt{vcg}(c, Q)$ *holds, then* $\{\, \texttt{wp}(c, Q) \,\}\ c\ \{\, Q \,\}$ *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 $\texttt{vcgen}(P, c, Q)$ logical formula, and its proof. [vcgen]

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

Theorem 19 [vcgen_correct] *If* $\texttt{vcgen}(P, c, Q)$ *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:

$$P \stackrel{\text{def}}{=} \lambda s.\ s(\mathtt{a}) \geq 0 \wedge s(\mathtt{b}) > 0$$

$$I \stackrel{\text{def}}{=} \lambda s.\ s(\mathtt{r}) \geq 0 \wedge s(\mathtt{b}) > 0 \wedge s(\mathtt{a}) = s(\mathtt{b}) \times s(\mathtt{q}) + s(\mathtt{r})$$

$$Q \stackrel{\text{def}}{=} \lambda s.\ s(\mathtt{q}) = s(\mathtt{a})/s(\mathtt{b})$$

To prove that $\{\, P \,\}\ c\ \{\, Q \,\}$, we apply theorem 19, then ask Coq to compute and simplify the formula $\mathtt{vcgen}(P, c, Q)$. We obtain the conjunction of three implications:

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

$$\neg(s(\mathtt{b}) < s(\mathtt{r}) + 1) \wedge s(\mathtt{r}) \geq 0 \wedge s(\mathtt{b}) > 0 \wedge s(\mathtt{a}) = s(\mathtt{b}) \times s(\mathtt{q}) + s(\mathtt{r}) \Longrightarrow s(\mathtt{q}) = s(\mathtt{a})/s(\mathtt{b})$$

$$s(\mathtt{b}) < s(\mathtt{r}) + 1 \wedge s(\mathtt{r}) \geq 0 \wedge s(\mathtt{b}) > 0 \wedge s(\mathtt{a}) = s(\mathtt{b}) \times s(\mathtt{q}) + s(\mathtt{r}) \Longrightarrow s(\mathtt{r}) - s(\mathtt{b}) \geq 0 \wedge s(\mathtt{b}) > 0 \wedge s(\mathtt{a}) = s(\mathtt{b}) \times (s(\mathtt{q}) + 1) + (s(\mathtt{r}) - s(\mathtt{b}))$$

which are easy to prove by purely arithmetic reasoning.

3.4. Strong Hoare triples

The axiomatic semantics we have seen so far enables us to prove partial correctness properties of programs, but not their termination. To prove termination as well, we need to use strong Hoare triples $[\, P \,]\ c\ [\, Q \,]$, meaning "if precondition P holds, the command c terminates and moreover the postcondition Q holds".

The rules defining valid strong Hoare triples are similar to those for weak triples, with the exception of the `while` rule, which contains additional requirements that ensure termination of the loop. [Triple]

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

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

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

$$\frac{(\forall v \in \mathbb{Z},\ [\, b\ \mathtt{true} \wedge e_m \doteq v \wedge P \,]\ c\ [\, 0 \dot{\leq} e_m \dot{<} v \wedge P \,]\ )}{[\, P \,]\ \mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done}\ [\, b\ \mathtt{false} \wedge P \,]} \quad \text{[Triple\_while]}$$

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

In the `Triple_while` rule, e_m stands for an expression whose value should decrease but remain nonnegative at each iteration. The precondition $e_m \doteq v$ and the postcondition $0 \dot{\leq} e_m \dot{<} v$ capture this fact:

$$e_m \doteq v \stackrel{\text{def}}{=} \lambda s.\ [\![e_m]\!]\ s = v \qquad 0 \dot{\leq} e_m \dot{<} v \stackrel{\text{def}}{=} \lambda s.\ 0 \leq [\![e_m]\!]\ s < v$$

The v variable therefore denotes the value of the measure expression at the beginning of the loop body. Since it is not statically known in general, rule `Triple_while` quantifies universally over every possible $v \in \mathbb{Z}$. Conceptually, rule `Triple_while` has infinitely many premises, one for each possible value of v. Such infinitely branching inference rules cause no difficulty in Coq.

Note that the `Triple_while` rule above is not powerful enough to prove termination for some loops that occur in practice, for example if the termination argument is based on a lexicographic ordering. A more general version of the rule could involve an arbitrary well-founded ordering between states.

The semantic interpretation $[\![[\, P \,]\ c\ [\, Q \,]]\!]$ of a strong Hoare triple is the proposition

$$\forall s,\ P\ s \Longrightarrow \exists s',\ (c, s \Rightarrow s') \wedge Q(s') \qquad \text{[sem\_Triple]}$$

As previously done for weak triples, we now prove the soundness of the inference rules for strong triples with respect to this semantic interpretation.

Theorem 20 [Triple_correct] *If* $[\, P \,]\ c\ [\, Q \,]$ *can be derived by the rules of axiomatic semantics, then* $[\![[\, P \,]\ c\ [\, Q \,]]\!]$ *holds.*

The proof is by an outer induction on a derivation of $[\, P \,]\ c\ [\, Q \,]$ followed, in the `while` case, by an inner induction on the value of the associated measure expression.

3.5. Further reading

The material in this section follows Nipkow [45] (in Isabelle/HOL) and Bertot [12] (in Coq), themselves following Gordon [38].

Separation logic [46,53] 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.* [36] in Coq, Tuch *et al.* [55] in Isabelle/HOL, Appel *et al* [5] in Coq, and Myreen and Gordon [44] in HOL4.

The generation of verification conditions (section 3.3) is an instance of a more general technique known as "proof by reflection", which aims at replacing deduction steps by computations [13, chap. 16]. The derivation of $\{\, P \,\}\ c\ \{\, Q \,\}$ from the rules of section 3.1 (a nonobvious process involving nondeterminstic proof search) is replaced by the computation of $\texttt{vcgen}(P, c, Q)$ (a trivial evaluation of a recursive function application). Proofs by reflection can tremendously speed up the verification of combinatorial properties, as illustrated by Gonthier and Werner's mechanized proof of the 4-color theorem [23].

4. Compilation to a virtual machine

There are several ways to execute programs:

- Interpretation: a program (the interpreter) traverses the abstract syntax tree of the program to be executed, performing the intended computations on the fly.
- Compilation to native code: before execution, the program is translated to a sequence of machine instructions. These instructions are those of a real microprocessor and are executed in hardware.
- Compilation to virtual machine code: before execution, the program is translated to a sequence of instructions, These instructions are those of a *virtual machine*. They do not correspond to that of an existing hardware processor, but are chosen close to the basic operations of the source language. Then, the virtual machine code is either interpreted (more efficiently than source-level interpretation) or further translated to real machine code.

In this section, we study the compilation of the IMP language to an appropriate virtual machine.

4.1. The IMP virtual machine

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 store s (mapping variables to integers).

The instruction set is as follows: [instruction] [code]

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

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

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.

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

As in section 2.2, 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 $C(pc') = \texttt{halt}$.
- Divergence $C \vdash (pc, \sigma, s) \Uparrow$ if the machine makes infinitely many transitions from (pc, σ, s).
- Going wrong, otherwise.

Example. The table below depicts the first 4 transitions of the execution of the code $\texttt{var}(x); \texttt{const}(1); \texttt{add}; \texttt{setvar}(x); \texttt{branch}(-5)$.

stack	ε	$12.\varepsilon$	$1.12.\varepsilon$	$13.\varepsilon$	ε
store	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 13$
p.c.	0	1	2	3	4
code	$\texttt{var}(x);$	$\texttt{const}(1);$	$\texttt{add};$	$\texttt{setvar}(x);$	$\texttt{branch}(-5)$

The fifth transition executes the $\texttt{branch}(-5)$ instruction, setting the program counter back to 0. The overall effect is that of an infinite loop that increments x by 1 at each iteration.

4.2. The compilation scheme

The code $\texttt{comp}(e)$ for an expression evaluates e and pushes its value on top of the stack [compile_expr]. It executes linearly (no branches) and leaves the store unchanged. (This is the familiar translation from algebraic notation to reverse Polish notation.)

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

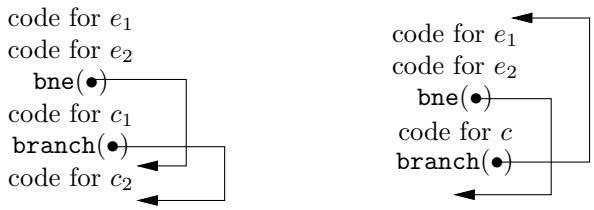

Figure 2. Shape of generated code for `if` $e_1 = e_2$ `then` c_1 `else` c_2 (left) and `while` $e_1 = e_2$ `do` c `done` (right)

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

$$\texttt{comp}(e_1 = e_2,\ \delta) = \texttt{comp}(e_1); \texttt{comp}(e_2); \texttt{bne}(\delta)$$
$$\texttt{comp}(e_1 < e_2,\ \delta) = \texttt{comp}(e_1); \texttt{comp}(e_2); \texttt{bge}(\delta)$$

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

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

$|C|$ 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 $\texttt{compile}(c) = \texttt{comp}(c); \texttt{halt}$. [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 $\texttt{comp}(c)$ 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 2?

4.3. 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?

To make this question precise, we assume given operational semantics for the two languages that associate to P_1, P_2 sets $\mathcal{B}(P_1), \mathcal{B}(P_2)$ of observable behaviors. In our case, observable behaviors are: termination on a final state s, divergence, and "going wrong". The set $\mathcal{B}(P)$ contains exactly one element if P has deterministic semantics, two or more otherwise.

Here are several possible formal characterizations of the informal claim that P_2 preserves the semantics of P_1.

- Bisimulation (equivalence): $\mathcal{B}(P_1) = \mathcal{B}(P_2)$
- Backward simulation (refinement): $\mathcal{B}(P_1) \supseteq \mathcal{B}(P_2)$
- Backward simulation for correct source programs: if $\texttt{wrong} \notin \mathcal{B}(P_1)$ then $\mathcal{B}(P_1) \supseteq \mathcal{B}(P_2)$
- Forward simulation: $\mathcal{B}(P_1) \subseteq \mathcal{B}(P_2)$
- Forward simulation for correct source programs: if $\texttt{wrong} \notin \mathcal{B}(P_1)$ then $\mathcal{B}(P_1) \subseteq \mathcal{B}(P_2)$

Bisimulation is the strongest notion of semantic preservation, ensuring that the two programs are indistinguishable. It is often too strong in practice. For example, the C language has non-deterministic semantics because the evaluation order for expressions is not fully specified; yet, C compilers choose one particular evaluation order while generating deterministic machine code; therefore, the generated code has fewer behaviors than the source code. This intuition corresponds to the backward simulation property defined above: all behaviors of P_2 are possible behaviors of P_1, but P_1 can have more behaviors.

In addition to reducing nondeterminism, compilers routinely optimize away "going wrong" behaviors. For instance, the source program P_1 contains an integer division $z := x/y$ that can go wrong if $y = 0$, but the compiler eliminated this division because z is not used afterwards, therefore generating a program P_2 that does not go wrong if $y = 0$. This additional degree of liberty is reflected in the "backward simulation for correct source programs" above.

Finally, the two "forward simulation" properties reverse the roles of P_1 and P_2, expressing the fact that any (non-wrong) behavior of the source program P_1 is a possible behavior of the compiled code P_2. Such forward simulation properties are generally much easier to prove than backward simulations, but provide apparently weaker guarantees: P_2 could have additional behaviors, not exhibited by P_1, that are undesirable, such as "going wrong". This cannot happen, however, if P_2 has deterministic semantics.

Lemma 21 *(Simulation and determinism.) If P_2 has deterministic semantics, then "forward simulation for correct programs" implies "backward simulation for correct programs".*

In conclusion, for deterministic languages such as IMP and IMP virtual machine code, "forward simulation for correct programs" is an appropriate notion of semantic preservation to prove the correctness of compilers and program transformations.

4.4. Semantic preservation for the compiler

Recall the informal specification for the code $\texttt{comp}(e)$ generated by the compilation of expression e: it should evaluate e and push its value on top of the stack, execute linearly (no branches), and leave the store unchanged. Formally, we should have $\texttt{comp}(e) : (0, \sigma, s) \xrightarrow{*} (|\texttt{comp}(e)|, (\llbracket e \rrbracket\ s).\sigma, s)$ for all stacks σ and stores s. Note that $pc = |\texttt{comp}(e)|$ means that the program counter is one past the last instruction in the sequence $\texttt{comp}(e)$. To enable a proof by induction, we need to strengthen this result and consider codes of the form $C_1; \texttt{comp}(e); C_2$, where the code for e is bracketed by two arbitrary code sequences C_1 and C_2. The program counter, then, should go from $|C_1|$ (pointing to the first instruction of $\texttt{comp}(e)$) to $|C_1| + |\texttt{comp}(e)|$ (pointing one past the last instruction of $\texttt{comp}(e)$, or equivalently to the first instruction of C_2).

Lemma 22 [compile_expr_correct] *For all instruction sequences C_1, C_2, stacks σ and states s,*

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

The proof is a simple induction on the structure of e. Here is a representative case: $e = e_1 + e_2$. Write $v_1 = \llbracket e_1 \rrbracket\ s$ and $v_2 = \llbracket e_2 \rrbracket\ s$. The code C is $C_1; \texttt{comp}(e_1); \texttt{comp}(e_2); \texttt{add}; C_2$. Viewing C as $C_1; \texttt{comp}(e_1); (\texttt{comp}(e_2); \texttt{add}; C_2)$, we can apply the induction hypothesis to e_1, obtaining the transitions

$$(|C_1|, \sigma, s) \xrightarrow{*} (|C_1| + |\texttt{comp}(e_1)|, v_1.\sigma, s)$$

Likewise, viewing C as $(C_1; \texttt{comp}(e_1)); \texttt{comp}(e_2); (\texttt{add}; C_2)$, we can apply the induction hypothesis to e_2, obtaining

$$(|C_1; \texttt{comp}(e_1)|, v_1.\sigma, s) \xrightarrow{*} (|C_1; \texttt{comp}(e_1)| + |\texttt{comp}(e_2)|, v_2.v_1.\sigma, s)$$

Combining these two sequences with an `add` transition, we obtain

$$(|C_1|, \sigma, s) \xrightarrow{*} (|C_1; \texttt{comp}(e_1); \texttt{comp}(e_2)| + 1, (v_1 + v_2).\sigma, s)$$

which is the desired result.

The statement and proof of correctness for the compilation of boolean expressions is similar. Here, the stack and the store are left unchanged, and control is transferred either to the end of the generated instruction sequence or to the given offset relative to this end, depending on the truth value of the condition.

Lemma 23 [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 $\llbracket b \rrbracket\ s = \texttt{true}$ and $pc = |C_1| + |\texttt{comp}(b)| + \delta$ otherwise.

To show semantic preservation between an IMP command and its compiled code, we prove a "forward simulation for correct programs" result. We therefore have two cases to consider: (1) the command terminates normally, and (2) the command diverges. In both cases, we use the natural semantics to conduct the proof, since its compositional nature is a good match for the compositional nature of the compilation scheme.

Theorem 24 [compile_cmd_correct_terminating] *Assume $c, s \Rightarrow s'$. Then, for all instruction sequences C_1, C_2 and stack σ,*

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

The proof is by induction on a derivation of $c, s \Rightarrow s'$ and uses the same techniques as that of lemma 22.

For the diverging case, we need the following special-purpose coinduction principle.

Lemma 25 *Let X be a set of (machine code, machine state) pairs such that*

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

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

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

Theorem 26 [compile_cmd_correct_diverging] *Assume $c, s \Rightarrow \infty$. Then, for all instruction sequences C_1, C_2 and stacks σ,*

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

This completes the proof of forward simulation for correct programs.

4.5. Further reading

The virtual machine used in this section matches a small subset of the Java Virtual Machine [35]. Other examples of mechanized verification of nonoptimizing compilers producing virtual machine code include Bertot [10] (for the IMP language), Klein and Nipkow [30] (for a subset of Java), and Grall and Leroy [33] (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 [24], for call-by-name and call-by-value λ-calculus.

Lemma 22 (correctness of compilation of arithmetic expression to stack machine code) is historically important: it is the oldest published compiler correctness proof (McCarthy and Painter [37], in 1967) and the oldest mechanized compiler correctness proof (Milner and Weyhrauch, [40], 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 [20] lists 99 references up to 2002.

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

Compilers are typically structured as a sequence of program transformations, also called *passes*. Some passes translate from one language to another, lower-level language, closer to machine code. The compilation scheme of section 4 is a representative example. Other passes are optimizations: they rewrite the program to an equivalent, but more efficient program. For example, the optimized program runs faster, or is smaller.

In this section, we study a representative optimization: dead code elimination. The purpose of this optimization, performed on the IMP source language, 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. This reduces both the execution time and the code size.

Example. Consider the command `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*.

5.1. 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 5.4 for discussion.) To keep things simple, we bound arbitrarily to N the number of iterations, and return a default over-approximation 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.

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

Lemma 27 [live_while_charact] *Let* $A' = \texttt{live}(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done},\ A)$. *Then:*

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

5.2. 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" ($\texttt{q} \notin A$), 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. In contrast, if `q` is live "after" ($\texttt{q} \in A$), all computations are necessary and $\texttt{dce}(c, A)$ returns c unchanged.

5.3. 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]

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

Lemma 28 [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 $[\![b]\!]\ s = [\![b]\!]\ s'$.*

The following two key lemmas show that agreement is preserved by parallel assignment to a live variable, or by unilateral assignment to a dead variable. The latter case corresponds to the replacement of $x := e$ by `skip`.

Lemma 29 [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 30 [agree_update_dead] *(Assignment to a dead variable.) If $s \approx s' : A$ and $x \notin A$, then $s[x \leftarrow v] \approx s' : A$.*

Using these lemmas, we can show forward simulation diagrams both for terminating and diverging commands c. In both case, we assume agreement on the variables live "before" c, namely $\texttt{live}(c, A)$.

Theorem 31 [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 32 [dce_correct_diverging] *If $c, s \Rightarrow \infty$ and $s \approx s_1 : \texttt{live}(c, A)$, then $\texttt{dce}(c, A), s_1 \Rightarrow \infty$.*

5.4. 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 [43] for a catalog of classic optimizations.

The results of liveness analysis can be exploited to perform register allocation (a crucial optimization performance-wise), following Chaitin's approach [18] [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 [31,32].

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 [**?**] prove, in Coq, the correctness of

several optimizations based on dataflow analyses, such as constant propagation and common subexpression elimination. Cachera *et al.* [17] present a reusable Coq framework for dataflow analyses. Besson *et al.* [16] describe a more general Coq framework for static analyses based on abstract interpretation.

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 [28]. 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 [14,19,30].

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 [48,15], 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 [17]. The shortcut we took in this section (bounding arbitrarily the number of iterations) is inelegant but a reasonable engineering compromise.

6. State of the art and current trends

While this lecture was illustrated using "toy" languages and machines, the techniques we presented, based on operational and axiomatic semantics and on their mechanization using proof assistants, do scale to realistic programming languages and systems. Here are some recent achievements using similar techniques, in reverse chronological order.

- The Verified Software Toolchain: a separation logic embedded in Coq to reason about concurrent C programs `http://vst.cs.princeton.edu/` [5].
- The verification of the seL4 secure micro-kernel (`http://nicta.com.au/research/projects/l4.verified/`) [29].
- 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/`) [31].
- 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 [30], 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 [22]
- 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) [41].

Here are 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? [8,34].

Binders and α-conversion. 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].

Shared-memory concurrency. Shared-memory concurrency raises major semantic difficulties, ranging from formalizing the "weakly-consistent" memory models implemented by today's multicore processors [54] to mechanizing program logics appropriate for proving concurrent programs correct [21,26].

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

[2] S. 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] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

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

[5] A. W. Appel. Verified software toolchain - (invited talk). In *Programming Languages and Systems – 20th European Symposium on Programming, ESOP 2011*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011.

[6] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. 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] G. Barthe, B. Grégoire, C. Kunz, and T. 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] N. Benton, A. Kennedy, and C. Varming. Some domain theory and denotational semantics in Coq. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2009.

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

[11] Y. Bertot. Coq in a hurry. Tutorial available at http://cel.archives-ouvertes.fr/inria-00001173, Oct. 2008.

[12] Y. Bertot. Theorem proving support in programming language semantics. In Y. Bertot, G. Huet, J.-J. LÃ©vy, and G. Plotkin, editors, *From Semantics to Computer Science — Essays in Honour of Gilles Kahn*, pages 337–362. Cambridge University Press, 2009.

[13] Y. Bertot and P. CastÃ©ran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer, 2004.

[14] Y. Bertot, B. GrÃ©goire, and X. 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] Y. Bertot and V. 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] F. Besson, D. Cachera, T. P. Jensen, and D. Pichardie. Certified static analysis by abstract interpretation. In *Foundations of Security Analysis and Design*, volume 5705 of *Lecture Notes in Computer Science*, pages 223–257. Springer, 2009.

[17] D. Cachera, T. P. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

[18] G. 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.

[19] S. Coupet-Grimal and W. 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.

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

[21] X. Feng, R. Ferreira, and Z. 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.

[22] A. 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.

[23] G. Gonthier. Formal proof — the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[24] T. Hardin, L. Maranget, and B. Pagano. Functional runtimes within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[26] A. Hobor, A. W. Appel, and F. 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.

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

[28] G. A. Kildall. A unified approach to global program optimization. In *1st symposium Principles of Programming Languages*, pages 194–206. ACM Press, 1973.
[29] G. Klein. Operating system verification — an overview. *Sadhana*, 34(1):27–69, 2009.
[30] G. Klein and T. 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.
[31] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
[32] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
[33] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.
[34] G. Li, S. Owens, and K. 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.
[35] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1999. Second edition.
[36] N. Marti, R. Affeldt, and A. 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.
[37] J. McCarthy and J. 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.
[38] Michael J. C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 387–439. Springer, 1988.
[39] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML (revised)*. The MIT Press, 1997.
[40] R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. In B. Meltzer and D. Michie, editors, *Proc. 7th Annual Machine Intelligence Workshop*, volume 7 of *Machine Intelligence*, pages 51–72. Edinburgh University Press, 1972.
[41] J. S. Moore. *Piton: a mechanically verified assembly-language*. Kluwer, 1996.
[42] P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B*, pages 577–631. The MIT Press/Elsevier, 1990.
[43] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.
[44] M. O. Myreen and M. 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.
[45] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics. *Formal Aspects of Computing*, 10(2):171–186, 1998.
[46] P. W. O'Hearn, J. C. Reynolds, and H. 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.
[47] C. Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In Y. Bertot, G. Huet, J.-J. LÃ©vy, and G. Plotkin, editors, *From Semantics to Computer Science — Essays in Honour of Gilles Kahn*, pages 383–414. Cambridge University Press, 2009.
[48] L. C. Paulson. Set theory for verification. II: Induction and recursion. *Journal of Automated Reasoning*, 15(2):167–215, 1995.
[49] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
[50] B. C. Pierce et al. *Software Foundations*. 2013. Electronic book, http://www.cis.upenn.edu/~bcpierce/sf/.
[51] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

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

[53] J. 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.

[54] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *36th symposium Principles of Programming Languages*, pages 379–391. ACM Press, 2009.

[55] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *34th symposium Principles of Programming Languages*, pages 97–108. ACM Press, 2007.

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