CompCert: formal verification of a realistic C compiler

Sandrine Blazy

IRISA - INRIA and University Rennes 1

3rd Asian-Pacific summer school on formal methods, 2011-08-19

(many slides from Xavier Leroy)
Formal semantics of programming languages

Provide a mathematically-precise answer to the question

*What does this program do, exactly?*
What does this program do, exactly?

#include <stdio.h>

int l; int main(int o, char **O, int I) {
    char c, *D = O[1]; if (o > 0) {
        for (l = 0; D[l]; D[l]++)
            D[l] += 20; putchar((D[l] + 1032) / 20);
    } else {
        c = o + (D[I] + 82) % 10 - (I > l / 2) * (D[I - l + l] + 72) / 10 - 9;
        D[I] += I < 0 ? 0 : !o = main(c / 10, O, I - 1) * ((c + 999) % 10 - (D[I] + 92) % 10);
    }
    return o;
}

(Raymond Cheong, 2001)
What does this program do, exactly?

#include <stdio.h>

int l; int main(int o, char **O, int I)
{
    char c, *D = O[1];
    if (o > 0)
    {
        for (l = 0; D[l]; D[l] += 10)
        {
            D[l] -= 120;
            D[l] -= 110;
            while (!main(0, O, l)) D[l] += 20;
            putchar((D[l] + 1032) / 20);
        }
        putchar(10);
    } else
    {
        c = o + (D[I] + 82) % 10 - (I > l / 2) * (D[I - l + I] + 72) / 10 - 9;
        D[I] += I < 0 ? 0 : !(o = main(c / 10, O, I - 1)) * ((c + 999) % 10 - (D[I] + 92) % 10);
    }
    return o;
}

(Raymond Cheong, 2001)

(It computes arbitrary-precision square roots.)
Why indulge in formal semantics?

- An intellectually challenging issue.
- When English prose is not enough.
  (e.g. language standardization documents.)
- A prerequisite to formal program verification.
  (Program proof, model checking, static analysis, etc.)
- A prerequisite to building reliable “meta-programs”
  (Programs that operate over programs: compilers, code generators, program verifiers, type-checkers, . . . )
Is this program transformation correct?

double dotproduct(int n, double * a, double * b)
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}

Compiled for the Alpha processor with all optimizations and manually decompiled back to C...
double dotproduct(int n, double * a, double * b)
{
    double dp, a0, a1, a2, a3, b0, b1, b2, b3;
    double s0, s1, s2, s3, t0, t1, t2, t3;
    int i, k;
    dp = 0.0;
    if (n <= 0) goto L5;
    s0 = s1 = s2 = s3 = 0.0;
    i = 0; k = n - 3;
    if (k <= 0 || k > n) goto L19;
    i = 4; if (k <= i) goto L14;
    a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1];
    i = 8; if (k <= i) goto L16;
    L17: a2 = a[2]; b2 = b[2]; t0 = a0 * b0;
        a3 = a[3]; b3 = b[3]; t1 = a1 * b1;
        a0 = a[4]; b0 = b[4]; t2 = a2 * b2; t3 = a3 * b3;
        a1 = a[5]; b1 = b[5];
        s0 += t0; s1 += t1; s2 += t2; s3 += t3;
        a += 4; i += 4; b += 4;
        prefetch(a + 20); prefetch(b + 20);
        if (i < k) goto L17;
    L16: s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
        a += 4; b += 4;
        a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1];
    L18: s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
        a += 4; b += 4;
        dp = s0 + s1 + s2 + s3;
        if (i >= n) goto L5;
    L19: dp += a[0] * b[0];
        i += 1; a += 1; b += 1;
        if (i < n) goto L19;
    L5: return dp;
    L14: a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1]; goto L18;
}
Proof assistants

- Implementations of well-defined mathematical logics.
- Provide a specification language to write definitions and state theorems.
- Provide ways to build proofs in interaction with the user. (Not fully automated proving.)
- Check the proofs for soundness and completeness.

Formal semantics for realistic programming languages are large (but shallow) formal systems.

Computers are better than humans at checking large but shallow proofs.
Using the IMP toy language as an example, we will review and show how to mechanize (see the associated Coq development):

1. Operational semantics.
2. Compilation to virtual machine code and its correctness proof.
3. The last part will be devoted to the CompCert compiler.
Part I

Operational semantics
Operational semantics

1. Warm-up: expressions and their denotational semantics

2. The IMP language and its reduction semantics
Warm-up: symbolic expressions

A language of expressions comprising

- variables $x, y, \ldots$
- integer constants $0, 1, -5, \ldots, n$
- $e_1 + e_2$ and $e_1 - e_2$
  where $e_1, e_2$ are themselves expressions.

Objective: mechanize the syntax and semantics of expressions.
Syntax of expressions

Modeled as an **inductive type** (see the lecture on inductive data types).

**Definition** \( \text{ident} := \text{nat} \).

**Inductive** \( \text{expr} : \text{Type} := \)

\[
\begin{align*}
| \ \text{Evar}: \text{ident} & \rightarrow \text{expr} \quad (* \ \text{Evar} (v:\text{ident}) *) \\
| \ \text{Econst}: \text{Z} & \rightarrow \text{expr} \quad (* \ \text{Econst} (i:\text{Z}) *) \\
| \ \text{Eadd}: \text{expr} & \rightarrow \text{expr} \rightarrow \text{expr} \quad (* \ \text{Eadd} (e1 \ e2:\ \text{expr}) *) \\
| \ \text{Esub}: \text{expr} & \rightarrow \text{expr} \rightarrow \text{expr} \quad (* \ \text{Esub} (e1 \ e2:\ \text{expr}) *) \).
\end{align*}
\]

\( \text{Evar}, \ \text{Econst}, \ \text{etc.} \) are functions that construct terms of type \( \text{expr} \).

All terms of type \( \text{expr} \) are finitely generated by these 4 functions \( \rightarrow \) enables case analysis and induction.
Denotational semantics of expressions

Define $[e]_s$ as the **denotation** of expression $e$ (the integer it evaluates to) in state $s$ (a mapping from variable names to integers).

In ordinary mathematics, the denotational semantics is presented as a set of equations:

$$
[x]_s = s(x) \\
[n]_s = n \\
[e_1 + e_2]_s = [e_1]_s + [e_2]_s \\
[e_1 - e_2]_s = [e_1]_s - [e_2]_s
$$
Mechanizing the denotational semantics

In Coq, the denotational semantics is presented as a **recursive function** (≈ a definitional interpreter).

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.
Using the denotational semantics (1/3)

As an interpreter, to evaluate expressions.

Definition initial_state: state := fun (x: ident) => 0.

Definition update (s: state) (x: ident) (n: Z) : state :=
  fun y => if eq_ident x y then n else s y.

Eval compute in (let x : ident := 0 in
  let s : state := update initial_state x 12 in
  eval_expr s (Eadd (Evar x) (Econst 1))).

Coq prints = 13 : Z.
Using the denotational semantics (1/3, cont’d)

Can also generate Caml code automatically (Coq’s extraction mechanism).

**Extraction eval_expr.**

```ml
(** val eval_expr : state -> expr -> z **)
let rec eval_expr s = function
  | Evar x -> s x
  | Econst n -> n
  | Eadd (e1, e2) -> zplus (eval_expr s e1) (eval_expr s e2)
  | Esub (e1, e2) -> zminus (eval_expr s e1) (eval_expr s e2)
```

Using the denotational semantics (1/3, cont’d)

Can also generate Caml code automatically (Coq’s extraction mechanism).

Recursive Extraction eval_expr.

... type expr = Evar of ident | Econst of z |
\ | Eadd of expr * expr | Esub of expr * expr ...

let zplus x y = ...

(** val eval_expr : state -> expr -> z **) let rec eval_expr s = function |
\ | Evar x -> s x |
\ | Econst n -> n |
\ | Eadd (e1, e2) -> zplus (eval_expr s e1) (eval_expr s e2) |
\ | Esub (e1, e2) -> zminus (eval_expr s e1) (eval_expr s e2)
Using the denotational semantics (2/3)

To reason symbolically over expressions.

Lemma expr_add_pos:
   forall s x,
   s x >= 0 -> eval_expr s (Eadd (Evar x) (Econst 1)) > 0.
Proof.
   simpl.
   (* goal becomes: forall s x, s x >= 0 -> s x + 1 > 0 *)
   intros. omega.
Qed.
Using the denotational semantics (3/3)

To prove “meta” properties of the semantics. For example: 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, occurs_in x e -> s1 x = s2 x) ->
eval_expr s1 e = eval_expr s2 e.

where the predicate occurs_in is defined by

Fixpoint occurs_in (x: ident) (e: expr) {struct e} : Prop :=
match e with
| Evar y => x = y
| Econst n => False
| Eadd e1 e2 => occurs_in x e1 \/ occurs_in x e2
| Esub e1 e2 => occurs_in x e1 \/ occurs_in x e2
end.
Variant 1: interpreting arithmetic differently

Example: signed, modulo $2^{32}$ arithmetic (as in Java).

```plaintext
Fixpoint eval_expr1 (s: state) (e: expr) {struct e} : Z :=
match e with
| Evar x => s x
| Econst n => n
| Eadd e1 e2 => normalize(eval_expr1 s e1 + eval_expr1 s e2)
| Esub e1 e2 => normalize(eval_expr1 s e1 - eval_expr1 s e2)
end.

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

Definition normalize (x : Z) : Z :=
let y := x mod 4294967296 in
if Z_lt_dec y 2147483648 then y else y - 4294967296.
```
Variant 2: accounting for undefined expressions

In some languages, the value of an expression can be undefined:
- if it mentions an undefined variable;
- in case of arithmetic operation overflows (ANSI C);
- in case of division by zero;
- etc.

Recommended approach: use option types, with None meaning “undefined” and Some \( n \) meaning “defined and having value \( n \).

```
Inductive option (A: Type): A -> option A :=
  | None: option A
  | Some: A -> option A.
```
Variant 2: accounting for undefined expressions

Definition ostate := ident -> option Z.

Fixpoint eval_expr2 (s: ostate) (e: expr) {struct e} : option Z :=
    match e with
    | Evar x => s x
    | Econst n => Some n
    | Eadd e1 e2 =>
        match eval_expr2 s e1, eval_expr2 s e2 with
        | Some n1, Some n2 => Some (n1 + n2)
        | _, _ => None
        end
    end
    | Esub e1 e2 =>
        match eval_expr2 s e1, eval_expr2 s e2 with
        | Some n1, Some n2 => Some (n1 - n2)
        | _, _ => None
        end
    end.
Summary

The “denotational semantics as a Coq function” is natural and convenient...

... but limited by a fundamental aspect of Coq: all Coq functions must be total (= terminating).

❌ Cannot use this approach to give semantics to languages featuring general loops or general recursion.

✔️ Use relational presentations “predicate state term result” instead of functional presentations “result = function state term”.
Operational semantics

1. Warm-up: expressions and their denotational semantics

2. The IMP language and its reduction semantics
The IMP language

A prototypical imperative language with structured control.

Expressions:

\[ e ::= x \ | \ n \ | \ e_1 + e_2 \ | \ e_1 - e_2 \]

Boolean expressions (conditions):

\[ b ::= e_1 = e_2 \ | \ e_1 < e_2 \]

Commands (statements):

\[ c ::= \text{skip} \quad \text{(do nothing)} \]
\[ \quad | \quad x := e \quad \text{(assignment)} \]
\[ \quad | \quad c_1 ; c_2 \quad \text{(sequence)} \]
\[ \quad | \quad \text{if } b \text{ then } c_1 \text{ else } c_2 \quad \text{(conditional)} \]
\[ \quad | \quad \text{while } b \text{ do } c \text{ done} \quad \text{(loop)} \]
Abstract syntax

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

Inductive bool_expr : Type :=
  | Bequal: expr -> expr -> bool_expr
  | Bless: expr -> expr -> bool_expr.

Inductive cmd : Type :=
  | Cskip: cmd
  | Cassign: ident -> expr -> cmd
  | Cseq: cmd -> cmd -> cmd
  | Cifthenelse: bool_expr -> cmd -> cmd -> cmd
  | Cwhile: bool_expr -> cmd -> cmd.
Reduction semantics

Also called “structured operational semantics” (Plotkin) or “small-step semantics”.

Like the λ-calculus: view computations as sequences of reductions

\[ M \xrightarrow{\beta} M_1 \xrightarrow{\beta} M_2 \xrightarrow{\beta} \ldots \]

Each reduction \( M \rightarrow M' \) represents an elementary computation. \( M' \) represents the residual computations that remain to be done later.
Reduction semantics for IMP

Reductions are defined on (command, state) pairs (to keep track of changes in the state during assignments).

Reduction rule for assignments:

$$(x := e, s) \rightarrow (\text{skip}, \text{update } s \times n) \quad \text{if } [e]_s = n$$
Reduction semantics for IMP

Reduction rules for sequences:

\[
((\text{skip}; c), s) \rightarrow (c, s)
\]

\[
((c_1; c_2), s) \rightarrow ((c'_1; c_2), s') \quad \text{if} \quad (c_1, s) \rightarrow (c'_1, s')
\]

Example

\[
((x := x + 1; x := x - 2), s) \rightarrow ((\text{skip}; x := x - 2), s')
\]
\[
\rightarrow (x := x - 2, s')
\]
\[
\rightarrow (\text{skip}, s'')
\]

where \(s' = \text{update } s \times (s(x) + 1)\) and \(s'' = \text{update } s' \times (s'(x) - 2)\).
Reduction semantics for IMP

Reduction rules for conditionals and loops:

\[(\text{if } b \text{ then } c_1 \text{ else } c_2, s) \rightarrow (c_1, s) \quad \text{if } \llbracket b \rrbracket s = \text{true}\]
\[(\text{if } b \text{ then } c_1 \text{ else } c_2, s) \rightarrow (c_2, s) \quad \text{if } \llbracket b \rrbracket s = \text{false}\]
\[(\text{while } b \text{ do } c \text{ done}, s) \rightarrow (\text{skip}, s) \quad \text{if } \llbracket b \rrbracket s = \text{false}\]
\[(\text{while } b \text{ do } c \text{ done}, s) \rightarrow ((c; \text{while } b \text{ do } c \text{ done}), s) \quad \text{if } \llbracket s \rrbracket b = \text{true}\]

with

\[\llbracket e_1 = e_2 \rrbracket s = \begin{cases} 
\text{true} & \text{if } \llbracket e_1 \rrbracket s = \llbracket e_2 \rrbracket s; \\
\text{false} & \text{if } \llbracket e_1 \rrbracket s \neq \llbracket e_2 \rrbracket s
\end{cases}\]

and likewise for \(e_1 < e_2\).
Reduction semantics as inference rules

\[(x := e, \ s) \rightarrow (\text{skip, } \ s[x \leftarrow [e] \ s])\]

\[
\begin{align*}
(c_1, s) & \rightarrow (c'_1, s') \\
((c_1; c_2), s) & \rightarrow ((c'_1; c_2), s')
\end{align*}
\]

\[
\begin{align*}
[(b)] s = \text{true} & \quad \Rightarrow (\text{if } b \text{ then } c_1 \text{ else } c_2, s) \rightarrow (c_1, s) \\
[(b)] s = \text{false} & \quad \Rightarrow (\text{if } b \text{ then } c_1 \text{ else } c_2, s) \rightarrow (c_2, s)
\end{align*}
\]

\[
\begin{align*}
[(b)] s = \text{true} & \quad \Rightarrow ((\text{while } b \text{ do } c \text{ done}), s) \rightarrow ((c; \text{while } b \text{ do } c \text{ done}), s) \\
[(b)] s = \text{false} & \quad \Rightarrow ((\text{while } b \text{ do } c \text{ done}), s) \rightarrow (\text{skip}, s)
\end{align*}
\]
Expressing inference rules in Coq

Step 1: write each rule as a proper logical formula

\[(x := e, s) \rightarrow (\text{skip}, s[x \leftarrow [e] s])\]

\[((c_1; c_2), s) \rightarrow ((c_1'; c_2), s')\]

\[
\begin{align*}
\forall x. e. s, \\
\text{red}(\text{Cassign } x. e. s) (\text{Cskip. update } s. x. (\text{eval_expr } s. e))
\end{align*}
\]

\[
\begin{align*}
\forall c_1. c_2. s. c_1'. s', \\
\text{red}(c_1, s) (c_1', s') \rightarrow \\
\text{red}(\text{Cseq } c_1. c_2. s) (\text{Cseq } c_1'. c_2. s')
\end{align*}
\]

Step 2: give a name to each rule and wrap them in an inductive predicate definition.
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).
Using inductive definitions

Each case of the definition is a theorem that lets you conclude \( \text{red} \ (c, s) \ (c', s') \) appropriately.

Moreover, the proposition \( \text{red} \ (c, s) \ (c', s') \) holds only if it was derived by applying these theorems a finite number of times (smallest fixpoint).

Reasoning principles: by case analysis on the last rule used; by induction on a derivation.

Example

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

Proved by induction on a derivation of \( \text{red} \ cs \ cs1 \) and a case analysis on the last rule used to prove \( \text{red} \ cs \ cs2 \).
Sequences of reductions

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.

\[
(c, s) \rightarrow \cdots \rightarrow (\text{skip}, s')
\]

- **Divergence** \((c, s \Uparrow)\): infinite sequence of reductions.

\[
\forall (c', s'), (c, s) \rightarrow \cdots \rightarrow (c', s') \Rightarrow \exists c'', s'', (c', s') \rightarrow (c'', s'')
\]

- **Going wrong** \((c, s \Downarrow \text{wrong})\): finite sequence of reductions to an irreducible state that is not \text{skip}.

\[
(c, s) \rightarrow \cdots \rightarrow (c', s') \not\rightarrow \text{ with } c' \neq \text{skip}
\]
Sequences of reductions

The Coq presentation uses a generic library of closure operators over relations \( R : A \rightarrow A \rightarrow \text{Prop} \):

- \text{star} \( R : A \rightarrow A \rightarrow \text{Prop} \) (reflexive transitive closure)
- \text{infseq} \( R : A \rightarrow \text{Prop} \) (infinite sequences)
- \text{irred} \( R : A \rightarrow \text{Prop} \) (no reduction is possible)

Definition terminates (c: cmd) (s s’: state) : Prop :=
  star red (c, s) (Cskip, s’).

Definition diverges (c: cmd) (s: state) : Prop :=
  infseq red (c, s).

Definition goes_wrong (c: cmd) (s: state) : Prop :=
  exists c’, exists s’,
  star red (c, s) (c’, s’) \&\& c’ <> Cskip \&\& irred red (c’, s’).
Part II

Compilation to a virtual machine
Execution models for a programming language

1. **Interpretation:**
   the program is represented by its abstract syntax tree. The interpreter traverses this tree during execution.
Execution models for a programming language

1. **Interpretation:**
   the program is represented by its abstract syntax tree. The interpreter traverses this tree during execution.

2. **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.
Execution models for a programming language

1. **Interpretation:**
   the program is represented by its abstract syntax tree. The interpreter traverses this tree during execution.

2. **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.

3. **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,
   - either the virtual machine instructions are interpreted (efficiently)
   - or they are further translated to machine code (JIT).
Compilation to a virtual machine

3 The IMP virtual machine

4 Compiling IMP programs to virtual machine code

5 Notions of semantic preservation

6 Semantic preservation for our compiler
The IMP virtual machine

Components of the machine:

- The **code** $C$: a list of instructions.
- The **program counter** $pc$: an integer, giving the position of the currently-executing instruction in $C$.
- The **state** $s$ (a.k.a. store): a mapping from variable names to integer values.
- The **stack** $\sigma$: a list of integer values (used to store intermediate results temporarily).
The instruction set

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

By default, each instruction increments \( pc \) by 1.

Exception: branch instructions increment it by \( 1 + ofs \).

(\( ofs \) is a branch offset relative to the next instruction.)
Example

<table>
<thead>
<tr>
<th>stack</th>
<th>$\epsilon$</th>
<th>12</th>
<th>12</th>
<th>13</th>
<th>$\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>$x \mapsto 12$</td>
<td>$x \mapsto 12$</td>
<td>$x \mapsto 12$</td>
<td>$x \mapsto 12$</td>
<td>$x \mapsto 13$</td>
</tr>
<tr>
<td>p.c.</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>code</td>
<td><code>var(x);</code></td>
<td><code>const(1);</code></td>
<td><code>add;</code></td>
<td><code>setvar(x);</code></td>
<td><code>branch(-5)</code></td>
</tr>
</tbody>
</table>
Semantics of the machine

Given by a transition relation (small-step), representing the execution of one instruction.

Definition code := list instruction.
Definition stack := list Z.
Definition machine_state := (Z * stack * state).

Inductive transition (c: code):
    machine_state -> machine_state -> Prop :=
    | trans_const: forall pc stk s n,
      code_at c pc = Some(Iconst n) ->
      transition c (pc, stk, s) (pc + 1, n :: stk, s)
    | trans_var: forall pc stk s x,
      code_at c pc = Some(Ivar x) ->
      transition c (pc, stk, s) (pc + 1, s x :: stk, s)
    | trans_setvar: forall pc stk s x n,
      code_at c pc = Some(Isetvar x) ->
      transition c (pc, n :: stk, s) (pc + 1, stk, update s x n)
Semantics of the machine

- **trans_add**: $\forall pc \ stk \ s \ n1 \ n2,\newline\quad \text{code_at} c \ pc = \text{Some}(\text{Iadd}) \rightarrow \newline\quad \text{transition} c \ (pc, n2 :: n1 :: stk, s) \ (pc + 1, (n1 + n2) :: stk, s)$

- **trans_sub**: $\forall pc \ stk \ s \ n1 \ n2,\newline\quad \text{code_at} c \ pc = \text{Some}(\text{Isub}) \rightarrow \newline\quad \text{transition} c \ (pc, n2 :: n1 :: stk, s) \ (pc + 1, (n1 - n2) :: stk, s)$

- **trans_branch**: $\forall pc \ stk \ s \ ofs \ pc',\newline\quad \text{code_at} c \ pc = \text{Some}(\text{Ibranch} \ ofs) \rightarrow \newline\quad pc' = pc + 1 + ofs \rightarrow \newline\quad \text{transition} c \ (pc, stk, s) \ (pc', stk, s)$

- **trans_bne**: $\forall pc \ stk \ s \ ofs \ n1 \ n2 \ pc',\newline\quad \text{code_at} c \ pc = \text{Some}(\text{Ibne} \ ofs) \rightarrow \newline\quad pc' = (\text{if} \ Z_{eq\_dec} n1 \ n2 \ \text{then} \ pc + 1 \ \text{else} \ pc + 1 + ofs) \rightarrow \newline\quad \text{transition} c \ (pc, n2 :: n1 :: stk, s) \ (pc', stk, s)$

- **trans_bge**: $\forall pc \ stk \ s \ ofs \ n1 \ n2 \ pc',\newline\quad \text{code_at} c \ pc = \text{Some}(\text{Ibge} \ ofs) \rightarrow \newline\quad pc' = (\text{if} \ Z_{lt\_dec} n1 \ n2 \ \text{then} \ pc + 1 \ \text{else} \ pc + 1 + ofs) \rightarrow \newline\quad \text{transition} c \ (pc, n2 :: n1 :: stk, s) \ (pc', stk, s)$.
Executing machine programs

By iterating the transition relation:

- **Initial (machine) states**: \( pc = 0 \), initial state, empty stack.
- **Final (machine) states**: \( pc \) points to a `halt` instruction, empty stack.

Definition `mach_terminates (c: code) (s_init s_fin: state) :=
exists pc,
code_at c pc = Some Ihalt /
star (transition c) (0, nil, s_init) (pc, nil, s_fin).

Definition `mach_diverges (c: code) (s_init: state) :=
infseq (transition c) (0, nil, s_init).

Definition `mach_goes_wrong (c: code) (s_init: state) :=
(* otherwise *)
Compilation to a virtual machine

3. The IMP virtual machine

4. Compiling IMP programs to virtual machine code

5. Notions of semantic preservation

6. Semantic preservation for our compiler
Compilation scheme for expressions

The code `comp_e(e)` for an expression should:
- evaluate `e` and push its value on top of the stack;
- execute linearly (no branches);
- leave the state unchanged.

\[
\begin{align*}
\text{comp}_e(x) & = \text{var}(x) \\
\text{comp}_e(n) & = \text{const}(n) \\
\text{comp}_e(e_1 + e_2) & = \text{comp}_e(e_1); \text{comp}_e(e_2); \text{add} \\
\text{comp}_e(e_1 - e_2) & = \text{comp}_e(e_1); \text{comp}_e(e_2); \text{sub}
\end{align*}
\]

(= translation to “reverse Polish notation”.)
Compilation scheme for conditions

The code \( \text{comp}_b(b, ofs) \) for a boolean expression should:

- evaluate \( b \);
- fall through (continue in sequence) if \( b \) is true;
- branch to relative offset \( ofs \) if \( b \) is false;
- leave the stack and the state unchanged.

\[
\begin{align*}
\text{comp}_b(e_1 = e_2, \ ofs) &= \text{comp}_e(e_1); \text{comp}_e(e_2); \text{bne}(ofs) \\
\text{comp}_b(e_1 < e_2, \ ofs) &= \text{comp}_e(e_1); \text{comp}_e(e_2); \text{bge}(ofs)
\end{align*}
\]

Example

\[
\begin{align*}
\text{comp}_b(x + 1 < y - 2, \ ofs) &= \\
\text{var}(x); \text{const}(1); \text{add}; \\
\text{var}(y); \text{const}(2); \text{sub}; \\
\text{bge}(ofs)
\end{align*}
\]

(\text{compute } x + 1) \\
(\text{compute } y - 2) \\
(\text{branch if } \geq)
Compilation scheme for commands

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

\[
\begin{align*}
\text{comp}(\text{skip}) &= \epsilon \\
\text{comp}(x := e) &= \text{comp}_e(e); \text{setvar}(x) \\
\text{comp}(c_1; c_2) &= \text{comp}(c_1); \text{comp}(c_2)
\end{align*}
\]
Compilation scheme for commands

\[ \text{code for } e_1 \]
\[ \text{code for } e_2 \]
\[ \text{bne/bge(\bullet)} \]
\[ \text{code for } c_1 \]
\[ \text{branch(\bullet)} \]
\[ \text{code for } c_2 \]

\[ \text{code for } e_1 \]
\[ \text{code for } e_2 \]
\[ \text{bne/bge(\bullet)} \]
\[ \text{code for } c \]
\[ \text{branch(\bullet)} \]

\[
\text{comp(if } b \text{ then } c_1 \text{ else } c_2) = \text{comp}_b(b, |C_1| + 1); C_1; \text{branch}(|C_2|); C_2 \\
\text{where } C_1 = \text{comp}(c_1) \text{ and } C_2 = \text{comp}(c_2)
\]

\[
\text{comp(while } b \text{ do } c \text{ done)} = B; C; \text{branch}(-(|B| + |C| + 1)) \\
\text{where } C = \text{comp}(c) \\
\text{and } B = \text{comp}_b(b, |C| + 1)
\]
Compiling whole program

The compilation of a program $c$ is the code

\[
\text{compile}(c) = \text{comp}(c); \text{halt}
\]

Example

The compiled code for while $x < 10$ do $y := y + x$ done is

- `var(x); const(10); bge(5);` skip over loop if $x \geq 10$
- `var(y); var(x); add; setvar(y);` do $y := y + x$
- `branch(-8);` branch back to beginning of loop
- `halt` finished
Coq mechanization of the compiler

As recursive functions:

Fixpoint comp_e (e: expr): code :=
  match e with ... end.

Definition comp_b (b: bool_expr) (ofs: Z): code :=
  match b with ... end.

Fixpoint comp (c: cmd): code :=
  match c with ... end.

Definition compile_program (c: cmd) : code :=
  comp c ++ Ihalt :: nil.

These functions can be executed from within Coq, or extracted to executable Caml code.
To run a program, we compile it, then run the generated virtual machine code.

The compiler verification problem

Verify that a compiler is semantics-preserving: the generated code behaves as prescribed by the semantics of the source program.
Compilation to a virtual machine

3 The IMP virtual machine

4 Compiling IMP programs to virtual machine code

5 Notions of semantic preservation

6 Semantic preservation for our compiler
Comparing the behaviors of two programs

Consider two programs $P_1$ and $P_2$, possibly in different languages.

(For example, $P_1$ is an IMP command and $P_2$ is virtual machine code generated by compiling $P_1$.)

The operational semantics of the two languages associate to $P_1$, $P_2$ sets $\mathcal{B}(P_1), \mathcal{B}(P_2)$ of observable behaviors. In our case:

$$\text{observable behavior ::= terminates(s) | diverges | goeswrong}$$

Note that $\text{card}(\mathcal{B}(P)) = 1$ if $P$ is deterministic, and $\text{card}(\mathcal{B}(P)) > 1$ if not.
Observable behaviors

For an imperative language with I/O: add a trace of input-output operations performed during execution.

Example

\[
\begin{align*}
x & := 1; \ x := 2; & \approx & \ x := 2; \\
\text{print}(1); \ \text{print}(2); & \not\approx & \ \text{print}(2);
\end{align*}
\]
Bisimulation (equivalence)

\[ B(P_1) = B(P_2) \]

The source and transformed programs are completely indistinguishable.

Often too strong in practice ...
Reducing non-determinism during compilation

Languages such as C leave evaluation order partially unspecified.

```c
int x = 0;
int f(void) { x = x + 1; return x; }
int g(void) { x = x - 1; return x; }
```

The expression \( f() + g() \) can evaluate either to

- 1 if \( f() \) is evaluated first (returning 1), then \( g() \) (returning 0);
- -1 if \( g() \) is evaluated first (returning 1), then \( f() \) (returning 0).

Every C compiler chooses one evaluation order at compile-time.

The compiled code therefore has fewer behaviors than the source program (1 instead of 2).
Backward simulation (refinement)

\[ \mathcal{B}(P_1) \supseteq \mathcal{B}(P_2) \]

All possible behaviors of \( P_2 \) are legal behaviors of \( P_1 \), but \( P_2 \) can have fewer behaviors (e.g. because some behaviors were eliminated during compilation).
Should going wrong behaviors be preserved?

Compilers routinely optimize away going-wrong behaviors. For example:

\[
x := \frac{1}{y}; \quad x := 42 \quad \text{optimized to} \quad x := 42
\]

(goes wrong if \( y = 0 \))

(always terminates normally)
Safe backward simulation

Restrict ourselves to source programs that cannot go wrong:

\[
\text{goeswrong} \notin \mathcal{B}(P_1) \implies \mathcal{B}(P_1) \supseteq \mathcal{B}(P_2)
\]
The pains with backward simulation

Safe backward simulation looks like the semantic preservation property we expect from a correct compiler.

It is however rather difficult to prove:

- We need to consider all steps that the compiled code can take, and trace them back to steps the source program can take.
- This is problematic if one source-level step is broken into several machine-level steps.
  (E.g. $x:=a$ is one step in IMP, but several instructions in the VM)
Forward simulations

If $P_2$ is compiler-generated from $P_1$, it is generally much easier to reason inductively on an execution of $P_1$ than on an execution of $P_2$.

Forward simulation property: $\mathcal{B}(P_1) \subseteq \mathcal{B}(P_2)$

Safe forward simulation property:

$$\text{goeswrong} \notin \mathcal{B}(P_1) \implies \mathcal{B}(P_1) \subseteq \mathcal{B}(P_2)$$

Significantly easier to prove than backward simulations, but not informative enough, apparently:

$P_2$ has all the good behaviors of $P_1$, but could have additional bad behaviors.
Determinism to the rescue

**Lemma**

If $P_2$ is deterministic ($\mathcal{B}(P_2)$ is a singleton), then
- “forward simulation” implies “backward simulation”
- “safe forward simulation” implies “safe backward simulation”

Utterly trivial result: follows from $\emptyset \subset X \subseteq \{y\} \Rightarrow X = \{y\}$. 
Our plan for verifying a compiler

1. Prove “safe forward simulation” between source and compiled codes.
2. Prove that the target language (machine code) is deterministic.
3. Conclude that all functional specifications are preserved by compilation.
Compilation to a virtual machine

3. The IMP virtual machine

4. Compiling IMP programs to virtual machine code

5. Notions of semantic preservation

6. Semantic preservation for our compiler
Verifying the compilation of expressions

Remember the “contract” for the code $\text{comp}_e(e)$: it should

- evaluate $e$ and push its value on top of the stack;
- execute linearly (no branches);
- leave the state unchanged.

\[
\forall st \ a \ pc \ stk,\ 
\text{star} \ (\text{transition} \ (\text{comp}_e \ a)) \\
(0, stk, st) \\
(length \ (\text{comp}_e \ a), \text{eval_expr} \ st \ a :: stk, st).
\]

For this statement to be provable by induction over the structure of the expression $a$, we need to generalize it so that

- the start PC is not necessarily 0,
- the code $\text{comp}_e \ a$ appears as a fragment of a larger code.
Verifying the compilation of expressions

Lemma compile_expr_correct:
  forall st a pc stk c1 c2,
  pc = length c1 ->
  star (transition (c1 ++ comp_e a ++ c2))
    (pc, stk, st)
    (pc + length (comp_e a), eval_expr st a :: stk, st).

Proof: a simple induction on the structure of \( a \), using the associativity of ++ and +.

The base cases are trivial.

- \( a = n \): a single Iconst transition.
- \( a = x \): a single Ivar transition.
An inductive case: $a = a_1 + a_2$

Write $v_1 = \llbracket a_1 \rrbracket s$ and $v_2 = \llbracket a_2 \rrbracket s$. By induction hypothesis (twice),

$C_1; \text{comp\_e}(a_1); (\text{comp\_e}(a_2); \text{add}; C_2):$

$$\left( |C_1|, stk, s \right) \xrightarrow{*} \left( |C_1| + |\text{comp\_e}(a_1)|, v_1.stk, s \right)$$

$(C_1; \text{comp\_e}(a_1)); \text{comp\_e}(a_2); (\text{add}; C_2):$

$$\left( |C_1; \text{comp\_e}(a_1)|, v_1.stk, s \right) \xrightarrow{*} \left( |C_1; \text{comp\_e}(a_1)| + |\text{comp\_e}(a_2)|, v_2.v_1.stk \right)$$

Combining with an add transition, we obtain:

$C_1; (\text{comp\_e}(a_1); \text{comp\_e}(a_2); \text{add}); C_2 :$

$$\left( |C_1|, stk, s \right) \xrightarrow{*} \left( |C_1; \text{comp\_e}(a_1); \text{comp\_e}(a_2)| + 1, (v_1 + v_2).stk, s \right)$$

which is the desired result since

$\text{comp\_e}(a_1 + a_2) = \text{comp\_e}(a_1); \text{comp\_e}(a_2); \text{add}$.
As simple as this proof looks, it is of historical importance:

Other verifications

- Boolean expressions: similar approach
  Proof: induction on the structure of \( b \), plus copious case analysis.

- Commands, terminating case
  An induction on the structure of \( c \) fails because of the WHILE case.
  An induction on a derivation tree representing the execution of \( c \) works perfectly.

- Commands, diverging case
  If command \( c \) diverges when started in state \( st \), then in the virtual machine, execution code \( (\text{comp } c) \) from initial state \( st \), makes infinitely many transitions.

This completes the proof of safe forward simulation.
Part III

The CompCert verified C compiler
The CompCert project
X.Leroy, S.Blazy et. al - compcert.inria.fr

Develop and prove correct a realistic compiler, targeted to critical embedded software.

- Source language: a subset of C.
- Target languages: PowerPC and ARM assembly.
- Generates reasonably compact and fast code ⇒ some optimizations.

This is “software-proof codesign” (as opposed to proving an existing compiler).

Used Coq to mechanize the proof of semantic preservation and also to implement most of the compiler.
The subset of C supported

Supported:
- Types: integers, floats, arrays, pointers, struct, union.
- Operators: arithmetic, pointer arithmetic.
- Control: if/then/else, loops, goto, switch.
- Functions, recursive functions, function pointers.

Not supported at all:
- The long long and long double types.
- longjmp/setjmp.

Supported through de-sugaring after parsing (not proved !):
- Block-scoped variables.
- Variable-arity functions.
- Assignment & pass-by-value of struct and union.
- Bit-fields.
The formally verified part of the compiler

- **Compcert C**
  - side-effects out of expressions

- **Clight**
  - type elimination
  - loop simplifications

- **C#minor**
  - stack allocation
  - of variables

- **RTL**
  - CFG construction
  - expr. decomp.

- **CminorSel**
  - instruction selection

- **Cminor**
  - (Instruction scheduling)

- **LTL**
  - linearization of the CFG

- **LTLin**
  - spilling, reloading
  - calling conventions

- **Linear**
  - layout of
  - stack frames

- **Asm**
  - asm code generation

- **Mach**

Optimizations: constant prop., CSE, tail calls, (LCM)

S. Blazy (IRISA-INRIA)
The whole CompCert compiler

C source → construct AST → AST C → AST CompCert C

parsing

AST C → simplifications → AST CompCert C

type-checking
de-sugaring

Assembly → assembling → Executable

Type reconstruction

Graph coloring

Code linearization heuristic

printing of asm syntax

Not proved yet

Proved in Coq

S. Blazy (IRISA-INRIA)
Theorem transf_c_program_correct:
  \forall prog tprog behavior,
  transf_c_program prog = OK tprog \rightarrow
  not_wrong behavior \rightarrow
  Csem.exec_program prog behavior \rightarrow
  Asm.exec_program tprog behavior.

A composition of 14 proofs
(13 safe forward simulation proofs and 1 safe backward simulation proof).
Observable behaviors

Inductive program_behavior: Type :=
| Terminates: trace -> int -> program_behavior
| Diverges: trace -> program_behavior
| Reacts: traceinf -> program_behavior
| Goes_wrong: trace -> program_behavior.

trace = list of input-output events.
traceinf = infinite list of I/O events.

I/O events are generated for:
- Calls to external functions (system calls)
- Memory accesses to global variables (hardware devices)
The Coq proof

4 person-years of work + many experiments

Size of proof : 50 000 lines of Coq

Size of program proved : 8 000 lines

Low proof automation (could be improved).

<table>
<thead>
<tr>
<th>Code</th>
<th>Sem.Statements</th>
<th>Proof scripts</th>
<th>Misc</th>
</tr>
</thead>
<tbody>
<tr>
<td>13%</td>
<td>8%</td>
<td>17%</td>
<td>55%</td>
</tr>
<tr>
<td>7%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Programmed in Coq

The verified parts of the compiler are directly programmed in Coq’s specification language, in pure functional style.

- Monads are used to handle errors and state.
- Purely functional data structures.

Coq’s extraction mechanism produces executable Caml code from these Coq definitions, which is then linked with hand-written Caml parts.

Claim: pure functional programming is the shortest path between an executable program and its proof.
Performances of the generated code
(On a PowerPC G5 processor)

Execution time

Compilation times: within a factor of 2 of gcc -01.
Conclusions

Proving correct a realistic compiler - appears feasible.

Moreover, proof assistants such as Coq are adequate (but barely) for this task.

What next?
Enhancements to CompCert

Upstream :
- Formalize some of the emulated features (bitfields, etc.).
- Is there anything to prove about the C parser? preprocessor??

Downstream :
- Currently, we stop at assembly language with a C-like memory model.
- Refine the memory model to a flat array of bytes.
  (Issues with bounding the total stack size used by the program.)
- Refine to real machine language?
  (Cf. Moore’s Piton & Gypsy projects circa 1995)
Enhancements to CompCert

In the middle:
- More static analyses, esp. for nonaliasing.
- More optimizations? Possibly using verified translation validation (e.g. instruction scheduling, and software pipelining)?
- Other target architectures.
Extensions on the “input” side

1. Verification of front-ends / code generators for other source languages.
2. Formal connections with verification tools.