

# Mechanized semantics

Sandrine Blazy

IRISA - INRIA and University Rennes 1

4th Asian-Pacific summer school on formal methods,  
2012-07-19

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

```

# Part I

## Operational and denotational semantics



# Operational and denotational semantics

## Warm-up: symbolic expressions

A language of expressions comprising

- ▶ variables  $x, y, \dots$
- ▶ integer constants  $0, 1, -5, \dots, n$
- ▶  $e_1 + e_2$  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**.

Definition `ident := nat`.

```
Inductive expr : Type :=  
  | Evar: ident -> expr          (* Evar (v:ident) *)  
  | Econst: Z -> expr            (* Econst (i:Z) *)  
  | Eadd: expr -> expr -> expr   (* Eadd (e1 e2: expr) *)  
  | Esub: expr -> expr -> expr   (* Esub (e1 e2: expr) *).
```

`Evar`, `Econst`, etc. are functions that construct terms of type `expr`.

All terms of type `expr` are finitely generated by these 4 functions.

✓ Enables case analysis and induction.

## Denotational semantics of expressions

Define  $\llbracket e \rrbracket s$  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:

$$\llbracket x \rrbracket s = s(x)$$

$$\llbracket n \rrbracket s = n$$

$$\llbracket e_1 + e_2 \rrbracket s = \llbracket e_1 \rrbracket s + \llbracket e_2 \rrbracket s$$

$$\llbracket e_1 - e_2 \rrbracket s = \llbracket e_1 \rrbracket s - \llbracket e_2 \rrbracket s$$

# Mechanizing the denotational semantics

In Coq, the denotational semantics is presented as a **recursive function** ( $\approx$  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.
```

```
(** 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.
```

(The predicate `occurs_in` was defined in the previous lecture.)

## Variant 1: interpreting arithmetic differently

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

```
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  $n$ ”.  
(see the previous lecture on data types)

## 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
  | 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 and denotational semantics

# The IMP language

A prototypical imperative language with structured control.

Expressions:

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

Boolean expressions (conditions):

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

Commands (statements):

$c ::= \text{skip}$	(do nothing)
$\mid x := e$	(assignment)
$\mid c_1; c_2$	(sequence)
$\mid \text{if } b \text{ then } c_1 \text{ else } c_2$	(conditional)
$\mid \text{while } b \text{ do } c \text{ done}$	(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”.

View computations as sequences of **reductions**

$$M \rightarrow M_1 \rightarrow M_2 \rightarrow \dots$$

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:

$$(x := e, s) \rightarrow (\text{skip}, \text{update } s \ x \ n) \quad \text{if } \llbracket e \rrbracket s = n$$

## Reduction semantics for IMP

Reduction rules for sequences:

$$\begin{aligned} ((\text{skip}; c), s) &\rightarrow (c, s) \\ ((c_1; c_2), s) &\rightarrow ((c'_1; c_2), s') \quad \text{if } (c_1, s) \rightarrow (c'_1, s') \end{aligned}$$

### Example

$$\begin{aligned} ((x := x + 1; x := x - 2), s) &\rightarrow ((\text{skip}; x := x - 2), s') \\ &\rightarrow (x := x - 2, s') \\ &\rightarrow (\text{skip}, s'') \end{aligned}$$

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:

$$(\text{if } b \text{ then } c_1 \text{ else } c_2, s) \rightarrow (c_1, s) \quad \text{if } \llbracket b \rrbracket s = \text{true}$$

$$(\text{if } b \text{ then } c_1 \text{ else } c_2, s) \rightarrow (c_2, s) \quad \text{if } \llbracket b \rrbracket s = \text{false}$$

$$(\text{while } b \text{ do } c \text{ done}, s) \rightarrow (\text{skip}, s) \quad \text{if } \llbracket b \rrbracket s = \text{false}$$

$$(\text{while } b \text{ do } c \text{ done}, s) \rightarrow ((c; \text{while } b \text{ do } c \text{ done}), s) \\ \text{if } \llbracket s \rrbracket b = \text{true}$$

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

$$\begin{array}{c}
 (x := e, s) \rightarrow (\text{skip}, s[x \leftarrow \llbracket e \rrbracket s]) \\
 \\
 \frac{(c_1, s) \rightarrow (c'_1, s')}{((c_1; c_2), s) \rightarrow ((c'_1; c_2), s')} \qquad ((\text{skip}; c), s) \rightarrow (c, s) \\
 \\
 \frac{\llbracket b \rrbracket s = \text{true}}{((\text{if } b \text{ then } c_1 \text{ else } c_2), s) \rightarrow (c_1, s)} \\
 \\
 \frac{\llbracket b \rrbracket s = \text{false}}{((\text{if } b \text{ then } c_1 \text{ else } c_2), s) \rightarrow (c_2, s)}
 \end{array}$$

## Reduction semantics as inference rules (cont'd)

$$\frac{\llbracket b \rrbracket s = \text{true}}{((\text{while } b \text{ do } c \text{ done}), s) \rightarrow ((c; \text{while } b \text{ do } c \text{ done}), s)}$$
$$\frac{\llbracket b \rrbracket s = \text{false}}{((\text{while } b \text{ do } c \text{ done}), s) \rightarrow (\text{skip}, s)}$$

## Expressing inference rules in Coq

Step 1: write each rule as a proper logical formula

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

$$\frac{(c_1, s) \rightarrow (c'_1, s)}{((c_1; c_2), s) \rightarrow ((c'_1; c_2), s')}$$

```
forall x e s,
  red (Cassign x e, s) (Cskip, update s x (eval_expr s e))
```

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

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 `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').

## Pros and cons of operational semantics

### Pros:

- ▶ Clean, unquestionable characterization of program behaviors (termination, divergence, going wrong).
- ▶ Extends even to unstructured constructs (goto, concurrency).
- ▶ De facto standard in the type systems community and in the concurrency community.

### Cons:

- ▶ Does not follow the structure of programs; lack of a powerful induction principle.
- ▶ This is not the way interpreters are written!
- ▶ Some extensions require unnatural extensions of the syntax of terms (e.g. with call contexts in the case of IMP + procedures).

## Part II

# Natural semantics

## Natural semantics

Also called “big-step semantics”.

An alternate presentation of operational semantics, closer to an interpreter.

## Natural semantics: Intuitions

Consider a terminating reduction sequence for  $c; c'$ :

$$\begin{aligned} ((c; c'), s) &\rightarrow ((c_1; c'), s_1) \rightarrow \cdots \rightarrow ((\text{skip}; c'), s_2) \\ &\rightarrow (c', s_2) \rightarrow \cdots \rightarrow (\text{skip}, s_3) \end{aligned}$$

It contains a terminating reduction sequence for  $c$ :

$$(c, s) \rightarrow (c_1, s_1) \rightarrow \cdots \rightarrow (\text{skip}, s_2)$$

followed by another for  $c'$ .

Idea: 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  $s'$ ”.



# Rules for natural semantics (terminating case)

$$\begin{array}{c}
 \text{skip}, s \Rightarrow s \\
 \\
 \frac{c_1, s \Rightarrow s_1 \quad c_2, s_1 \Rightarrow s_2}{c_1; c_2, s \Rightarrow s_2} \qquad \frac{
 \begin{array}{l}
 c_1, s \Rightarrow s' \text{ if } \llbracket b \rrbracket s = \text{true} \\
 c_2, s \Rightarrow s' \text{ if } \llbracket b \rrbracket s = \text{false}
 \end{array}
 }{
 \text{if } b \text{ then } c_1 \text{ else } c_2, s \Rightarrow s'
 } \\
 \\
 \frac{\llbracket b \rrbracket s = \text{false}}{\text{while } b \text{ do } c \text{ done}, s \Rightarrow s} \\
 \\
 \frac{\llbracket b \rrbracket s = \text{true} \quad c, s \Rightarrow s_1 \quad \text{while } b \text{ do } c \text{ done}, s_1 \Rightarrow s_2}{\text{while } b \text{ do } c \text{ done}, s \Rightarrow s_2}
 \end{array}$$

# Their Coq transcription

```

Inductive exec: state -> cmd -> state -> Prop :=
| exec_skip: forall s,      exec s Cskip s
| exec_assign: forall s x e,
    exec s (Cassign x e) (update s x (eval_expr s e))
| exec_seq: forall s c1 c2 s1 s2,
    exec s c1 s1 -> exec s1 c2 s2 ->
    exec s (Cseq c1 c2) s2
| exec_if: forall s be c1 c2 s',
    exec s (if eval_bool_expr s be then c1 else c2) s' ->
    exec s (Cifthenelse be c1 c2) s'
| exec_while_loop: forall s be c s1 s2,
    eval_bool_expr s be = true ->
    exec s c s1 -> exec s1 (Cwhile be c) s2 ->
    exec s (Cwhile be c) s2
| exec_while_stop: forall s be c,
    eval_bool_expr s be = false ->
    exec s (Cwhile be c) s.

```

## Equivalence between natural and reduction semantics

Whenever we have two different semantics for the same language, try to prove that they are **equivalent**:

*Both semantics predict the same “terminates / diverges / goes wrong” behaviors for any given program.*

- ▶ Strengthens the confidence we have in both semantics.
- ▶ Justifies using whichever semantics is more convenient to prove a given property.

## From natural to reduction semantics

Theorem `exec_terminates` If  $c, s \Rightarrow s'$ , then  $(c, s) \xrightarrow{*} (\text{skip}, s')$ .

`forall s c s', exec s c s' -> terminates c s s'.`

Proof: by induction on a derivation of  $c, s \Rightarrow s'$  and case analysis on the last rule used. A representative case:

Hypothesis:  $c_1; c_2, s \Rightarrow s'$ .

Inversion:  $c_1, s \Rightarrow s_1$  and  $c_2, s_1 \Rightarrow s'$  for some intermediate state  $s_1$ .

Induction hypothesis:  $(c_1, s) \xrightarrow{*} (\text{skip}, s_1)$  and  $(c_2, s_1) \xrightarrow{*} (\text{skip}, s')$ .

Context lemma (separate induction):

$((c_1; c_2), s) \xrightarrow{*} ((\text{skip}; c_2), s_1)$

Assembling the pieces together, using the transitivity of  $\xrightarrow{*}$ :

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

# From reduction to natural semantics

## Theorem (terminates\_exec)

*If  $(c, s) \xrightarrow{*} (\text{skip}, s')$  then  $c, s \Rightarrow s'$ .*

## Lemma (red\_preserves\_exec)

*If  $(c, s) \rightarrow (c', s')$  and  $c', s' \Rightarrow s''$ , then  $c, s \Rightarrow s''$ .*

$$\begin{aligned}
 & (c_1, s_1) \rightarrow \cdots (c_i, s_i) \rightarrow (c_{i+1}, s_{i+1}) \rightarrow \cdots (\text{skip}, s_n) \\
 & (c_1, s_1) \rightarrow \cdots (c_i, s_i) \rightarrow (c_{i+1}, s_{i+1}) \rightarrow \cdots (\text{skip}, s_n) \Rightarrow s_n \\
 & \quad \vdots \\
 & (c_1, s_1) \rightarrow \cdots (c_i, s_i) \rightarrow (c_{i+1}, s_{i+1}) \Rightarrow s_n \\
 & (c_1, s_1) \rightarrow \cdots (c_i, s_i) \Rightarrow s_n \\
 & \quad \vdots \\
 & c_1, s_1 \Rightarrow s_n
 \end{aligned}$$

## Pros and cons of big-step semantics

### Pros:

- ▶ Follows naturally the structure of programs.
- ▶ Close connection with interpreters.
- ▶ Powerful induction principle (on the structure of derivations).
- ▶ Easy to extend with various structured constructs (functions and procedures, other forms of loops)

### Cons:

- ▶ Fails to characterize diverging executions.  
(More precisely: no distinction between divergence and going wrong.)
- ▶ Concurrency, unstructured control (goto) nearly impossible to handle.

## Part III

### Proving a toy compiler

# Proving a toy 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

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

<i>stack</i>	$\epsilon$	12	1 12	13	$\epsilon$
<i>state</i>	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 13$
<i>p.c.</i>	0	1	2	3	4
<i>code</i>	<code>var(x);</code>	<code>const(1);</code>	<code>add;</code>	<code>setvar(x);</code>	<code>branch(-5)</code>

## Small-step semantics of the machine

A transition relation, 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,  
  code_at c pc = Some(Iadd) ->  
  transition c (pc, n2 :: n1 :: stk, s) (pc+1, (n1+n2) :: stk, s)  
| trans_sub: forall pc stk s n1 n2,  
  code_at c pc = Some(Isub) ->  
  transition c (pc, n2 :: n1 :: stk, s) (pc+1, (n1-n2) :: stk, s)  
| trans_branch: forall pc stk s ofs pc',  
  code_at c pc = Some(Ibranch ofs) ->  
  pc' = pc + 1 + ofs ->  
  transition c (pc, stk, s) (pc', stk, s)  
| trans_bne: forall pc stk s ofs n1 n2 pc',  
  code_at c pc = Some(Ibne ofs) ->  
  pc' = (if Z_eq_dec n1 n2 then pc + 1 else pc + 1 + ofs) ->  
  transition c (pc, n2 :: n1 :: stk, s) (pc', stk, s)  
| trans_bge: forall pc stk s ofs n1 n2 pc',  
  code_at c pc = Some(Ibge ofs) ->  
  pc' = (if Z_lt_dec n1 n2 then pc + 1 else pc + 1 + ofs) ->  
  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 *)
```

# Proving a toy compiler

## Compilation scheme for expressions

The code  $\text{comp\_e}(e)$  for an expression should:

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

$$\text{comp\_e}(x) = \text{var}(x)$$

$$\text{comp\_e}(n) = \text{const}(n)$$

$$\text{comp\_e}(e_1 + e_2) = \text{comp\_e}(e_1); \text{comp\_e}(e_2); \text{add}$$

$$\text{comp\_e}(e_1 - e_2) = \text{comp\_e}(e_1); \text{comp\_e}(e_2); \text{sub}$$

(= 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.

$$\text{comp\_b}(e_1 = e_2, ofs) = \text{comp\_e}(e_1); \text{comp\_e}(e_2); \text{bne}(ofs)$$
$$\text{comp\_b}(e_1 < e_2, ofs) = \text{comp\_e}(e_1); \text{comp\_e}(e_2); \text{bge}(ofs)$$

### Example

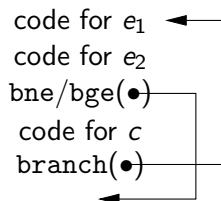
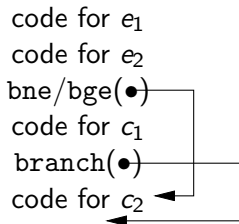
$$\text{comp\_b}(x + 1 < y - 2, ofs) =$$
$$\text{var}(x); \text{const}(1); \text{add};$$
$$(\text{compute } x + 1)$$
$$\text{var}(y); \text{const}(2); \text{sub};$$
$$(\text{compute } y - 2)$$
$$\text{bge}(ofs)$$
$$(\text{branch if } \geq)$$

## 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.

$$\begin{aligned}\text{comp}(\text{skip}) &= \epsilon \\ \text{comp}(x := e) &= \text{comp\_e}(e); \text{setvar}(x) \\ \text{comp}(c_1; c_2) &= \text{comp}(c_1); \text{comp}(c_2)\end{aligned}$$

# Compilation scheme for commands



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

## 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

<code>var(x); const(10); bge(5);</code>	skip over loop if $x \geq 10$
<code>var(y); var(x); add; setvar(y);</code>	do $y := y + x$
<code>branch(-8);</code>	branch back to beginning of loop
<code>halt</code>	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.

## Compiler verification

To run a program, we compile it, then run the generated virtual machine code.

We now have two ways to run a program:

- ▶ Interpret it using e.g. the definitional interpreter of part I.
- ▶ Compile it, then run the generated virtual machine code.

Will we get the same results either way?

### The compiler verification problem

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

# Proving a toy compiler

## Verifying the compilation of expressions

Remember the “contract” for the code `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,  
star (transition (comp_e a))  
  (0, stk, st)  
  (length (comp_e a), 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 `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 (2),

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

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

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

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

Combining with an add transition, we obtain:

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

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

which is the desired result since

$\text{comp\_e}(a_1 + a_2) = \text{comp\_e}(a_1); \text{comp\_e}(a_2); \text{add}.$

## Historical note

As simple as this proof looks, it is of historical importance:

- ▶ First published proof of compiler correctness.  
McCarthy & Painter, 1967,  
Correctness of a compiler for arithmetic expressions.
- ▶ First mechanized proof of compiler correctness.  
Milner and Weyrauch, 1972, using Stanford LCF,  
Proving compiler correctness in a mechanized logic.

## 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 (`comp c`) from initial state  $st$ , makes infinitely many transitions.

This completes the proof of safe forward simulation.

## Application: The CompCert project

X.Leroy, S.Blazy et. al - [compcert.inria.fr](http://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.

## Verified in Coq

```
Theorem transf_c_program_correct:  
  forall prog tprog behavior,  
    transf_c_program prog = OK tprog ->  
    not_wrong behavior ->  
    Csem.exec_program prog behavior ->  
    Asm.exec_program tprog behavior.
```

A composition of 14 proofs.

# Performances of the generated code

