Mechanized semantics

with applications to program proof and compiler verification

Xavier Leroy

INRIA Paris-Rocquencourt

Verification Technology, Systems & Applications 2013

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
++]-=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)

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.)

What about this one?

```
#define crBegin static int state=0; switch(state) { case 0:
#define crReturn(x) do { state=__LINE__; return x; \
                         case __LINE__:; } while (0)
#define crFinish }
```

```
int decompressor(void) {
    static int c, len;
    crBegin;
    while (1) {
        c = getchar();
        if (c == EOF) break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--) crReturn(c);
        } else crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}
```

(Simon Tatham, author of PuTTY)

What about this one?

```
#define crBegin static int state=0; switch(state) { case 0:
#define crReturn(x) do { state=__LINE__; return x; \
                         case __LINE__:; } while (0)
#define crFinish }
```

```
int decompressor(void) {
    static int c, len;
    crBegin;
    while (1) {
        c = getchar();
        if (c == EOF) break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--) crReturn(c);
        } else crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}
```

(Simon Tatham, author of PuTTY)

(It's a co-routined version of a decompressor for run-length encoding.)

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?

```
struct list { int head; struct list * tail; };

struct list * foo(struct list * p)
{
  return (p->tail = NULL);                p->tail = NULL;
                                          return p->tail;
}
```

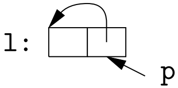
Is this program transformation correct?

```
struct list { int head; struct list * tail; };

struct list * foo(struct list * p)
{
  return (p->tail = NULL);
}
```

```
p->tail = NULL;
return p->tail;
```

No, not if `p == &(p->tail)` (circular list).



What about this one?

```
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.

Some mature proof assistants:

ACL2	HOL	PVS
Agda	Isabelle	Twelf
Coq	Mizar	

Using proof assistants to mechanize semantics

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

Computers are better than humans at checking large but shallow proofs.

✗ *The proofs of the remaining 18 cases are similar and make extensive use of the hypothesis that [. . .]*

✓ *The proof was mechanically checked by the XXX proof assistant. This development is publically available for review at* `http://...`

This lecture

Using the Coq proof assistant, illustrate how to mechanize formal semantics and some of the uses for these semantics.

Main objective: motivate students to try and mechanize some of their own work.

Side objective: familiarize students with the Coq specification language.

Not an objective: teaching how to conduct proofs in Coq.
(See Bertot's *Coq in a Hurry* and Pierce's *Software Foundations*.)

Lecture material

`http://gallium.inria.fr/~xleroy/courses/VTSA-2013/`

- A Coq development.
- The handout: summary of results in ordinary mathematical notation (+ references and further reading)

Contents

Using the IMP toy language as an example, we will review and show how to mechanize:

1. Operational semantics and a bit of denotational semantics.
2. Axiomatic semantics, with applications to program proof.
3. Compilation to virtual machine code and its correctness proof.
4. One optimizing program transformation and its correctness proof.

Part I

Operational and denotational semantics

Operational and denotational semantics

1. Warm-up: expressions and their denotational semantics
2. The IMP language and its reduction semantics
3. Natural semantics
4. Definitional interpreters
5. From definitional interpreters to denotational semantics
6. Summary

Warm-up: symbolic expressions

A language of expressions comprising

- variables x, y, ...
- integer constants 0, 1, -5, ..., 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
  | Econst: Z -> expr
  | Eadd: expr -> expr -> expr
  | Esub: expr -> expr -> expr.
```

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

All terms of type `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:

$$\begin{array}{rcl} [\![x]\!]\ s & = & s(x) \\ [\![n]\!]\ s & = & n \\ [\![e_1 + e_2]\!]\ s & = & [\![e_1]\!]\ s + [\![e_2]\!]\ s \\ [\![e_1 - e_2]\!]\ s & = & [\![e_1]\!]\ s - [\![e_2]\!]\ s \end{array}$$

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 := O in
  let s : state := update initial_state x 12 in
  eval_expr s (Eadd (Evar x) (Econst 1))).
```

Coq prints `= 13 : Z`.

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

Using the denotational semantics (2/3)

To reason symbolically over expressions.

```
Remark 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, is_free x e -> s1 x = s2 x) ->
  eval_expr s1 e = eval_expr s2 e.
```

where the predicate `is_free` is defined by

```
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.
```

Variant 1: interpreting arithmetic differently

Example: 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 => 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.
```

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 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 state := ident -> option Z.

Fixpoint eval_expr (s: state) (e: expr) {struct e} : option Z :=
  match e with
  | Evar x => s x        (* None if x is unininitialized *)
  | Econst n => Some n
  | Eadd e1 e2 =>
      match eval_expr s e1, eval_expr s e2 with
      | Some n1, Some n2 =>
          if representable (n1 + n2) then Some (n1 + n2) else None
      | _, _ => None
      end
  | Esub e1 e2 =>
      match eval_expr s e1, eval_expr s e2 with
      | Some n1, Some n2 =>
          if representable (n1 - n2) then Some (n1 - n2) else None
      | _, _ => 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)

- either because they are structurally recursive (recursive calls on a strict sub-term of the argument);
- or because they are defined by Noetherian recursion (not treated here).

$\rightarrow$ Cannot use this approach to give semantics to languages featuring general loops or general recursion (e.g. the λ-calculus).

$\rightarrow$ Use relational presentations “*predicate state term result*” instead of functional presentations “*result = function state term*”.

Operational and denotational semantics

1. Warm-up: expressions and their denotational semantics
2. The IMP language and its reduction semantics
3. Natural semantics
4. Definitional interpreters
5. From definitional interpreters to denotational semantics
6. Summary

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):

$$
\begin{array}{rll}
c ::= & \texttt{skip} & \text{(do nothing)} \\
\mid & x := e & \text{(assignment)} \\
\mid & c_1; c_2 & \text{(sequence)} \\
\mid & \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 & \text{(conditional)} \\
\mid & \texttt{while } b \texttt{ do } c \texttt{ done} & \text{(loop)}
\end{array}
$$

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

$$M \stackrel{\beta}{\to} M_1 \stackrel{\beta}{\to} M_2 \stackrel{\beta}{\to} \ldots$$

Each reduction $M \to M'$ 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:

$$(x := e,\ s) \to (\texttt{skip}, \texttt{update}\ s\ x\ n) \qquad \text{if } [\![e]\!]\ s = n$$

Reduction semantics for IMP

Reduction rules for sequences:

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

Example

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

where $s' = \texttt{update}\ s\ x\ (s(x) + 1)$ and $s'' = \texttt{update}\ s'\ x\ (s'(x) - 2)$.

Reduction semantics for IMP

Reduction rules for conditionals and loops:

$$
\begin{array}{rcll}
(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2,\ s) & \to & (c_1, s) & \text{if } [\![b]\!]\ s = \texttt{true} \\
(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2,\ s) & \to & (c_2, s) & \text{if } [\![b]\!]\ s = \texttt{false} \\
(\texttt{while } b \texttt{ do } c \texttt{ done},\ s) & \to & (\texttt{skip},\ s) & \text{if } [\![b]\!]\ s = \texttt{false} \\
(\texttt{while } b \texttt{ do } c \texttt{ done},\ s) & \to & ((c; \texttt{while } b \texttt{ do } c \texttt{ done}),\ s) & \\
 & & \text{if } [\![s]\!]\ b = \texttt{true} &
\end{array}
$$

with

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

and likewise for $e_1 < e_2$.

Reduction semantics as inference rules

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

$$\frac{(c_1, s) \to (c_1', s')}{((c_1; c_2),\ s) \to ((c_1'; c_2),\ s')}$$

$$((\texttt{skip}; c),\ s) \to (c, s)$$

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

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

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

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

Expressing inference rules in Coq

Step 1: write each rule as a proper logical formula

$$(x := e,\ s) \rightarrow (\mathtt{skip},\ s[x \leftarrow [\![e]\!]\ s]) \qquad \frac{(c_1, s) \rightarrow (c_1', s)}{((c_1; c_2),\ s) \rightarrow ((c_1'; c_2),\ s')}$$

```
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 `red` (c, s) (c', s') appropriately.

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).

$\rightarrow$ 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 `red cs cs1` and a case analysis on the last rule used to prove `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`.

$$(c, s) \rightarrow \cdots \rightarrow (\texttt{skip}, s')$$

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

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

Sequences of reductions

The Coq presentation uses a generic library of closure operators over relations $R : A \to A \to \texttt{Prop}$:

- `star` $R : A \to A \to \texttt{Prop}$ (reflexive transitive closure)
- `infseq` $R : A \to \texttt{Prop}$ (infinite sequences)
- `irred` $R : A \to \texttt{Prop}$ (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').
```

Exercise 1

Extend IMP and its semantics with:

(*) A richer language of boolean expressions:

Boolean expressions (conditions):

$$
\begin{array}{rll}
b ::= & e_1 = e_2 & \text{equality test} \\
\mid & e_1 < e_2 & \text{less-than test} \\
\mid & \texttt{not}(b_1) & \text{negation} \\
\mid & b_1 \texttt{ and } b_2 & \text{conjunction}
\end{array}
$$

(*) A new form of loop, `do` c `while` b, where the loop condition b is tested at the end of every iteration of c, not at the beginning.

(**) The `break` and `continue` commands. `break` terminates the nearest enclosing loop. `continue` jumps to the next iteration of this loop.

(***) What about adding procedures or functions?

Operational and denotational semantics

1. Warm-up: expressions and their denotational semantics
2. The IMP language and its reduction semantics
3. Natural semantics
4. Definitional interpreters
5. From definitional interpreters to denotational semantics
6. Summary

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 $c; c'$:

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

It contains a terminating reduction sequence for c:

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

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)

$$\mathtt{skip}, s \Rightarrow s \qquad\qquad x := e, s \Rightarrow s[x \leftarrow [\![e]\!]\ s]$$

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

$$\frac{[\![b]\!]\ s = \mathtt{false}}{\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done}, s \Rightarrow s}$$

$$\frac{[\![b]\!]\ s = \mathtt{true} \qquad c, s \Rightarrow s_1 \qquad \mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done}, s_1 \Rightarrow s_2}{\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done}, s \Rightarrow s_2}$$

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

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

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

Context lemma (separate induction): $((c_1; c_2), s) \xrightarrow{*} ((\texttt{skip}; c_2), s_1)$

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

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

From reduction to natural semantics

Theorem

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

Lemma

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

$$(c_1, s_1) \to \cdots (c_i, s_i) \to (c_{i+1}, s_{i+1}) \to \cdots (\mathtt{skip}, s_n)$$

$$(c_1, s_1) \to \cdots (c_i, s_i) \to (c_{i+1}, s_{i+1}) \to \cdots (\mathtt{skip}, s_n) \Rightarrow s_n$$

$$\vdots$$

$$(c_1, s_1) \to \cdots (c_i, s_i) \to (c_{i+1}, s_{i+1}) \Rightarrow s_n$$

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

$$\vdots$$

$$c_1, s_1 \Rightarrow s_n$$

Natural semantics for divergence

Our natural semantics correctly characterizes programs that terminate normally. What about the other two possible behaviors?

- Going wrong: can also give a set of natural-style rules characterizing this behavior, but not very interesting.
- Divergence: can also give a set of natural-style rules characterizing this behavior, but need to interpret them coinductively.

Natural semantics for divergence: Intuitions

Consider an infinite reduction sequence for $c; c'$. It must be of one of the following two forms:

$$((c; c'),\ s) \xrightarrow{*} ((c_i; c'),\ s_i) \rightarrow \cdots$$
$$((c; c'),\ s) \xrightarrow{*} ((\texttt{skip}; c'),\ s_i) \rightarrow (c', s_i) \xrightarrow{*} (c'_j, s_j) \rightarrow \cdots$$

I.e. either c diverges or it terminates normally and c' diverges.

Idea: write inference rules that follow this structure and define a predicate $c, s \Rightarrow \infty$, meaning "in initial state s, the command c diverges".

Natural semantics for divergence: Rules

$$\frac{c_1, s \Rightarrow \infty}{c_1; c_2, s \Rightarrow \infty} \qquad \frac{c_1, s \Rightarrow s_1 \qquad c_2, s_1 \Rightarrow \infty}{c_1; c_2, s \Rightarrow \infty}$$

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

$$\frac{[\![b]\!]\ s = \texttt{true} \qquad c, s \Rightarrow s_1 \qquad \texttt{while } b \texttt{ do } c \texttt{ done}, s_1 \Rightarrow \infty}{\texttt{while } b \texttt{ do } c \texttt{ done}, s \Rightarrow \infty}$$

Problem: interpreted normally as an inductive predicate, these rules define a predicate $c, s \Rightarrow \infty$ that is always false! (no axioms. . .)

Induction vs. coinduction in a nutshell

A set of axioms and inference rules can be interpreted in two ways:

Inductive interpretation:

- In set theory: the least defined predicate that satisfies the axioms and rules (smallest fixpoint).
- In proof theory: conclusions of finite derivation trees.

Coinductive interpretation:

- In set theory: the most defined predicate that satisfies the axioms and rules (biggest fixpoint).
- In proof theory: conclusions of finite or infinite derivation trees.

(See section 2 of *Coinductive big-step semantics* by H. Grall and X. Leroy.)

Example of inductive and coinductive interpretations

Consider the following inference rules for the predicate $\mathtt{even}(n)$

$$\mathtt{even}(0) \qquad\qquad \frac{\mathtt{even}(n)}{\mathtt{even}(S(S(n)))}$$

Assume that n ranges over $\mathbb{N} \cup \{\infty\}$, with $S(\infty) = \infty$.

With the inductive interpretation of the rules, the even predicate holds on the following numbers: 0, 2, 4, 6, 8, ... But $\mathtt{even}(\infty)$ does not hold.

With the coinductive interpretation, even holds on $\{2n \mid n \in \mathbb{N}\}$, and also on ∞. This is because we have an infinite derivation tree $\mathcal{T}$ that concludes $\mathtt{even}(\infty)$:

$$\mathcal{T} = \frac{\mathcal{T}}{\mathtt{even}(\infty)}$$

Coinductive predicates in Coq

```
CoInductive execinf: state -> cmd -> Prop :=
  | execinf_seq_left: forall s c1 c2,
      execinf s c1 ->
      execinf s (Cseq c1 c2)
  | execinf_seq_right: forall s c1 c2 s1,
      exec s c1 s1 -> execinf s1 c2 ->
      execinf s (Cseq c1 c2)
  | execinf_if: forall s b c1 c2,
      execinf s (if eval_bool_expr s b then c1 else c2) ->
      execinf s (Cifthenelse b c1 c2)
  | execinf_while_body: forall s b c,
      eval_bool_expr s b = true ->
      execinf s c ->
      execinf s (Cwhile b c)
  | execinf_while_loop: forall s b c s1,
      eval_bool_expr s b = true ->
      exec s c s1 -> execinf s1 (Cwhile b c) ->
      execinf s (Cwhile b c).
```

Example of divergence

The coinductive interpretation of the rules for $c, s \Rightarrow \infty$ captures classic examples of divergence.

Consider $c = \texttt{while}\ i < 0\ \texttt{do}\ i := i - 1\ \texttt{done}$
and the states $s_n = [x \mapsto -n]$.

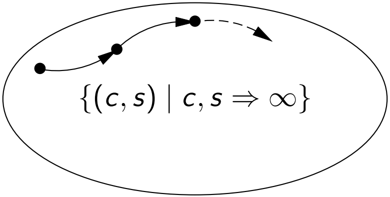
$$
\dfrac{
\begin{array}{c} [\![i < 0]\!]\ s_1 = \texttt{true} \\ i := i - 1, s_1 \Rightarrow s_2 \end{array}
\qquad
\dfrac{
\begin{array}{c} [\![i < 0]\!]\ s_2 = \texttt{true} \\ i := i - 1, s_2 \Rightarrow s_3 \end{array}
\qquad
\dfrac{
\begin{array}{c} [\![i < 0]\!]\ s_3 = \texttt{true} \\ i := i - 1, s_3 \Rightarrow s_4 \end{array}
\qquad
\dfrac{
\begin{array}{c} [\![i < 0]\!]\ s_4 = \texttt{true} \\ i := i - 1, s_4 \Rightarrow s_5 \end{array}
\qquad
\vdots
}{c, s_4 \Rightarrow \infty}
}{c, s_3 \Rightarrow \infty}
}{c, s_2 \Rightarrow \infty}
}{c, s_1 \Rightarrow \infty}
$$

Does the $c, s \Rightarrow \infty$ coinductive predicate capture the correct notion of divergence?

From natural semantics to reduction semantics

Lemma

If $c, s \Rightarrow \infty$, there exists c' and s' such that $(c, s) \rightarrow (c', s')$ and $c', s' \Rightarrow \infty$.



Theorem

If $c, s \Rightarrow \infty$, then $(c, s) \Uparrow$.

From reduction semantics to natural semantics

Theorem

If $(c, s) \Uparrow$ *, then* $c, s \Rightarrow \infty$*.*

The proof uses two inversion lemmas:

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

Note that these lemmas cannot be proved in Coq's constructive logic and require the excluded middle axiom $(\forall P,\ P \vee \neg P)$ from classical logic.

Constructive logic in a nutshell

In Coq's constructive logic, a proof is a terminating functional program:

- A proof of $A \to B$ $\approx$ a total function from proofs of A to proofs of B.
- A proof of $A \land B$ $\approx$ a pair of proofs, one for A and another for B.
- A proof of $A \lor B$ $\approx$ a decision procedure that decides which of A and B holds and returns either a proof of A or a proof of B.

A proposition such as $(c, s) \Uparrow \;\lor\; \exists c', \exists s',\ (c, s) \xrightarrow{*} (c', s') \land (c', s') \not\rightarrow$ cannot be proved constructively. (A constructive proof would solve the halting problem. The natural proof uses the excluded middle axiom, which is not constructive.)

Excluded middle or the axiom of choice can however be added to Coq as axioms without breaking consistency.

Operational and denotational semantics

1. Warm-up: expressions and their denotational semantics
2. The IMP language and its reduction semantics
3. Natural semantics
4. Definitional interpreters
5. From definitional interpreters to denotational semantics
6. Summary

Definitional interpreters

We cannot write a Coq function `cmd` $\rightarrow$ `state` $\rightarrow$ `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

$$\texttt{nat} \rightarrow \texttt{cmd} \rightarrow \texttt{state} \rightarrow (\texttt{Bot} \mid \texttt{Res}(\texttt{state}))$$

that takes as extra argument a natural number used to bound the amount of computation performed. (The "fuel".)

`Res`(s) is returned if, within that bound, the execution of the command terminates with final state s.

`Bot` is returned if the computation "runs out of fuel".

An IMP definitional interpreter in Coq

```
Inductive result: Type := Bot: result | Res: state -> result.

Definition bind (r: result) (f: state -> result) : result :=
  match r with Res s => f s | Bot => Bot end.

Fixpoint interp (n: nat) (c: cmd) (s: state) {struct n} : result :=
  match n with
  | O => Bot
  | S n' =>
      match c with
      | Cskip => Res s
      | Cassign x e => Res (update s x (eval_expr s e))
      | Cseq c1 c2 =>
          bind (interp n' c1 s) (fun s1 => interp n' c2 s1)
      | Cifthenelse b c1 c2 =>
          interp n' (if eval_bool_expr s b then c1 else c2) s
      | Cwhile b c1 =>
          if eval_bool_expr s b
          then bind (interp n' c1 s) (fun s1 => interp n' (Cwhile b c1) s1)
          else Res s
      end
```

Monotonicity property

Giving more fuel to the interpreter can only make the results more precise.

Order results r by $r \leq r$ and $\texttt{Bot} \leq r$.

Lemma

If $n \leq m$, then $\texttt{interp}\ n\ c\ s \leq \texttt{interp}\ m\ c\ s$.

Connections with natural semantics

Lemma

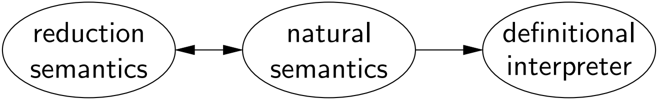
If $\texttt{interp}\ n\ c\ s = \texttt{Res}(s')$*, then* $c, s \Rightarrow s'$*.*

Lemma

If $c, s \Rightarrow s'$*, there exists an* n *such that* $\texttt{interp}\ n\ c\ s = \texttt{Res}(s')$*.*

Lemma

If $c, s \Rightarrow \infty$*, then* $\texttt{interp}\ n\ c\ s = \texttt{Bot}$ *for all* n*.*



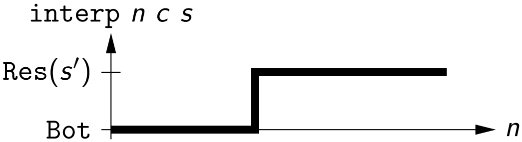
Operational and denotational semantics

1. Warm-up: expressions and their denotational semantics
2. The IMP language and its reduction semantics
3. Natural semantics
4. Definitional interpreters
5. From definitional interpreters to denotational semantics
6. Summary

From definitional interpreter to denotational semantics

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

For a terminating command:



For a diverging command:

A denotational semantics

Lemma

For every c, there exists a function $[\![c]\!]$ from states to evaluation results such that $\forall s,\ \exists m,\ \forall n \geq m,\ \texttt{interp}\ n\ c\ s = [\![c]\!]\ s$.

(The proof uses excluded middle and axiom of description, but no domain theory.)

The equations of denotational semantics

The denotation function $[\![\cdot]\!]$ satisfies the equations of denotational semantics:

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

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

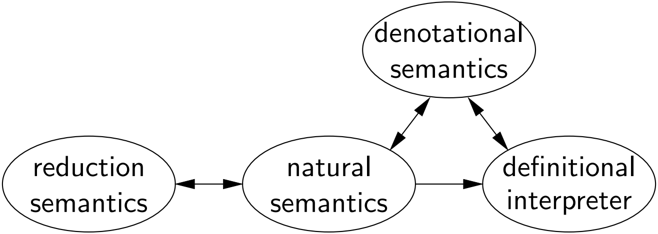
Relating denotational and natural semantics

Theorem

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

Theorem

$c, s \Rightarrow \infty$ *if and only if* $[\![c]\!]\ s = \perp$.



Operational and denotational semantics

1. Warm-up: expressions and their denotational semantics
2. The IMP language and its reduction semantics
3. Natural semantics
4. Definitional interpreters
5. From definitional interpreters to denotational semantics
6. Summary

Summary

Most forms of operational semantics can be mechanized easily and effectively in modern proof assistants:

- Relational presentations: reduction semantics, natural semantics.
- Functional presentations: definitional interpreters.

Denotational semantics either works "out of the box" (for strongly normalizing languages) or runs into deep theoretic issues (for Turing-complete languages).

(But see Agerholm, Paulin, and Benton et al for mechanizations of domain theory.)

Coming next: some nice things to do with these mechanized semantics.

Part II

Axiomatic semantics and program proof

Reasoning about a program

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

What does this program fragment do?

Under which conditions on `a` and `b`?

Can you prove it?

Reasoning about a program using operational semantics

We can stare long and hard at reduction sequences (or execution derivations), but . . .

$$
\begin{array}{l}
(\texttt{while}\ b < r + 1\ \texttt{do}\ r := r - b; q := q + 1\ \texttt{done},\ s) \\
\to (r := r - b; q := q + 1; \texttt{while}\ \ldots\ \texttt{done},\ s) \\
\to (\texttt{skip}; q := q + 1; \texttt{while}\ \ldots\ \texttt{done},\ s[r \leftarrow s(r) - s(b)]) \\
\to (q := q + 1; \texttt{while}\ \ldots\ \texttt{done},\ s[r \leftarrow s(r) - s(b)]) \\
\to (\texttt{skip}; \texttt{while}\ \ldots\ \texttt{done},\ s[r \leftarrow s(r) - s(b), q \leftarrow s(q) - 1])]) \\
\to (\texttt{while}\ b < r + 1\ \texttt{do}\ r := r - b; q := q + 1\ \texttt{done}, \\
\qquad\qquad\qquad\qquad s[r \leftarrow s(r) - s(b), q \leftarrow s(q) - 1])])
\end{array}
$$

Reasoning about a program using logical assertions

Better: reason about logical assertions about the program state at various program points.

$\{\ b > 0\ \}$

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

$\quad \{\ a = bq + r\ \ \wedge\ \ r \geq 0\ \ \wedge\ \ b > 0\ \}$

```
    r := r - b;
    q := q + 1
done
```

$\{\ a = bq + r\ \ \wedge\ \ 0 \leq r < b\ \}$

Are these assertions consistent with the run-time behavior of the program? How can we prove this consistency?

(The annotations above are not consistent, by the way.)

Axiomatic semantics and program proof

7. Axiomatic semantics (Hoare logic)
8. Automatic generation of verification conditions (VCgen)
9. Computing within proofs
10. Further reading

Hoare triples

Weak triple: $\{\, P \,\}\; c \;\{\, Q \,\}$

Intuitive meaning: if the initial state satisfies assertion P, the execution of c either

- terminates in a state satisfying assertion Q
- or diverges,
- but does not go wrong.

Strong triple: $[\, P \,]\; c \;[\, Q \,]$

Intuitive meaning: if the initial state satisfies assertion P, the execution of c always terminates and the final state satisfies Q.

Logical assertions

Modeled as any Coq predicate about the state.

```
Definition assertion := state -> Prop.
```

For example:

```
Definition invariant : assertion :=
  fun (s: state) =>
    s vr >= 0 /\ s vb > 0 /\ s va = s vb * s vq + s vr.
```

Axiomatic semantics / Hoare logic

Objective: define inference rules characterizing the predicates $\{ P \}\ c\ \{ Q \}$ and $[P]\ c\ [Q]$ for all commands c.

… and check that these rules are consistent with the operational semantics.

These rules can be viewed both

- As a program logic (Hoare logic), enabling reasoning over programs.
- As an alternate form of semantics (axiomatic semantics), defining the behavior of programs in terms of which assertions they satisfy.

The rules for weak triples (1/3)

```
Inductive triple: assertion -> cmd -> assertion -> Prop :=

  | triple_skip: forall P,
      triple P Cskip P

  | triple_assign: forall P x e,
      triple (aupdate P x e) (Cassign x e) P

  | triple_seq: forall c1 c2 P Q R,
      triple P c1 Q -> triple Q c2 R ->
      triple P (Cseq c1 c2) R
```

The aupdate operation over assertions is defined as:

```
Definition aupdate (P: assertion) (x: ident) (e: expr) :=
  fun (s: state) => P (update s x (eval_expr s e)).
```

The rules for weak triples (2/3)

```
| triple_if: forall be c1 c2 P Q,
    triple (aand (atrue be) P) c1 Q ->
    triple (aand (afalse be) P) c2 Q ->
    triple P (Cifthenelse be c1 c2) Q

| triple_while: forall be c P,
    triple (aand (atrue be) P) c P ->
    triple P (Cwhile be c) (aand (afalse be) P)
```

With the following operators over assertions:

```
Definition atrue (be: bool_expr) : assertion :=
  fun s => eval_bool_expr s be = true.
Definition afalse (be: bool_expr) : assertion :=
  fun s => eval_bool_expr s be = false.
Definition aand (P Q: assertion) : assertion :=
  fun s => P s /\ Q s.
```

The rules for weak triples (3/3)

```
  | triple_consequence: forall c P Q P' Q',
      triple P' c Q' -> aimp P P' -> aimp Q' Q ->
      triple P c Q.
```

Where `aimp` is pointwise implication:

```
Definition aimp (P Q: assertion) : Prop :=
  forall (s: state), P s -> Q s.
```

Example

$$\{\, a = bq + r \,\}\ r := r - b; q := q + 1\ \{\, a = bq + r \,\}$$

because

$$\begin{array}{rcl}
\texttt{aupdate}\ (a = bq + r)\ q\ (q+1) & \Longleftrightarrow & a = b(q+1) + r \\
\texttt{aupdate}\ (a = b(q+1) + r)\ r\ (r-b) & \Longleftrightarrow & a = b(q+1) + (r-b) \\
 & \Longleftrightarrow & a = bq + r
\end{array}$$

Soundness of the axiomatic semantics

Recall the intuition for the triple $\{ P \} \ c \ \{ Q \}$:

> *The command c, executed in an initial state satisfying P, either*
>
> - *terminates and the final state satisfies Q,*
> - *or diverges.*

We capture the conclusion using a coinductive predicate `finally`:

```
CoInductive finally: state -> cmd -> assertion -> Prop :=
  | finally_done: forall s (Q: assertion),
      Q s ->
      finally s Cskip Q
  | finally_step: forall c s c' s' Q,
      red (c, s) (c', s') -> finally s' c' Q ->
      finally s c Q.
```

(Remember: coinductive predicate $\Leftrightarrow$ finite or infinite derivation.)

Soundness of the axiomatic semantics

The validity of a weak Hoare triple $\{ P \}\ c\ \{ Q \}$ is, then, defined as the proposition

```
Definition sem_triple (P: assertion) (c: cmd) (Q: assertion) :=
  forall s, P s -> finally s c Q.
```

The soundness of the axiomatic semantics then follows from the theorem:

```
Theorem triple_correct:
  forall P c Q, triple P c Q -> sem_triple P c Q.
```

(Proof: by induction on a derivation of `triple P c Q`, plus auxiliary lemmas about `finally` proved by coinduction.)

Rules for strong Hoare triples

Same rules as for weak triples, except the `while` rule:

```
Inductive Triple: assertion -> cmd -> assertion -> Prop :=
  | Triple_while: forall be c P (measure: expr),
      (forall v,
        Triple (aand (atrue be) (aand (aequal measure v) P))
               c
               (aand (alessthan order measure v) P)) ->
      Triple P (Cwhile be c) (aand (afalse be) P)
  | ...
```

`measure` is an expression whose value must be ≥ 0 and strictly decrease at each loop iteration.

Rules for strong Hoare triples

```
Inductive Triple: assertion -> cmd -> assertion -> Prop :=
  | Triple_while: forall be c P (measure: expr),
      (forall v,
        Triple (aand (atrue be) (aand (aequal measure v) P))
               c
               (aand (alessthan measure v) P)) ->
      Triple P (Cwhile be c) (aand (afalse be) P)
  | ...
```

The `aequal` and `alessthan` assertions are defined as:

```
Definition aequal (e: expr) (v: Z) :=
  fun (s: state) => eval_expr s e = v.
Definition alessthan (e: expr) (v: Z) :=
  fun (s: state) => 0 <= eval_expr s e < v.
```

Rules for strong Hoare triples

```
Inductive Triple: assertion -> cmd -> assertion -> Prop :=
  | Triple_while: forall be c P (measure: expr),
      (forall v,
        Triple (aand (atrue be) (aand (aequal measure v) P))
               c
               (aand (alessthan measure v) P)) ->
      Triple P (Cwhile be c) (aand (afalse be) P)
  | ...
```

This rule has an infinity of premises: one for each possible value `v` of the expression `measure` at the beginning of the loop. Coq supports inductive reasoning on such rules just fine.

Soundness of the axiomatic semantics

A strong Hoare triple $[\,P\,]\ c\ [\,Q\,]$ is valid if, started in any state satisfying P, the command c terminates and its final state satisfies Q.

```
Definition sem_Triple (P: assertion) (c: cmd) (Q: assertion) :=
  forall s, P s -> exists s', exec s c s' /\ Q s'.

Theorem Triple_correct:
  forall P c Q, Triple P c Q -> sem_Triple P c Q.
```

(Proof: by induction on a derivation of `Triple P c Q`. The `while` case uses an inner Peano induction on the value of the `measure` expression.)

Axiomatic semantics and program proof

7 Axiomatic semantics (Hoare logic)

8 Automatic generation of verification conditions (VCgen)

9 Computing within proofs

10 Further reading

Weakest preconditions

Given a loop-free command c and a postcondition Q, we can compute (effectively) a weakest precondition P such that $\{ P \}\ c\ \{ Q \}$ holds.

Just run the rules of Hoare logic "backward", e.g. the weakest precondition of $x := e; y := e'$ is

$$\texttt{aupdate}\ (\texttt{aupdate}\ Q\ y\ e')\ x\ e$$

In the presence of loops, the weakest precondition is not computable
$\rightarrow$ ask the user to provide loop invariants as program annotations.

Program annotations

Annotated commands:

$$\begin{array}{rll} c ::= & \texttt{while}\ b\ \texttt{do}\ \{P\}\ c\ \texttt{done} & \text{loop with invariant} \\ \mid & \texttt{assert}(P) & \text{explicit assertion} \\ \mid & \texttt{skip} \mid x := e \mid c_1; c_2 & \text{as in IMP} \\ \mid & \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 & \text{as in IMP} \end{array}$$

The erase function turns annotated commands back into regular commands:

$$\begin{array}{rcl} \texttt{erase}(\texttt{while}\ b\ \texttt{do}\ \{P\}\ c\ \texttt{done}) & = & \texttt{while}\ b\ \texttt{do}\ \texttt{erase}(c)\ \texttt{done} \\ \texttt{erase}(\texttt{assert}(P)) & = & \texttt{skip} \end{array}$$

Computing the weakest precondition

The weakest (liberal) precondition for an annotated command a with postcondition Q is computed in two parts:

- An assertion P that acts as a precondition.
- A logical formula which, if true, implies that $\{ P \}$ $\texttt{erase}(a)$ $\{ Q \}$ holds.

Computing the precondition

```
Fixpoint wp (a: acmd) (Q: assertion) struct a : assertion :=
  match a with
  | Askip => Q
  | Aassign x e => aupdate Q x e
  | Aseq a1 a2 => wp a1 (wp a2 Q)
  | Aifthenelse b a1 a2 =>
      aor (aand (atrue b) (wp a1 Q)) (aand (afalse b) (wp a2 Q))
      (* either b is true and wp a1 Q holds,
           or b is false and wp a2 Q holds. *)
  | Awhile b P a1 => P
      (* the user-provided loop invariant is the precondition *)
  | Aassert P => P
      (* ditto *)
  end.
```

Computing the validity conditions

In essence: making sure that the user-provided loop invariants and assertions are true during every execution.

```
Fixpoint vcg (a: acmd) (Q: assertion) {struct a} : Prop :=
  match a with
  | Askip => True
  | Aassign x e => True
  | Aseq a1 a2 => vcg a1 (wp a2 Q) /\ vcg a2 Q
  | Aifthenelse b a1 a2 => vcg a1 Q /\ vcg a2 Q
  | Awhile b P a1 =>
      vcg a1 P /\
      aimp (aand (afalse b) P) Q /\
      aimp (aand (atrue b) P) (wp a1 P)
  | Aassert P =>
      aimp P Q
  end.
```

The verification condition generator

Combining the two together, we obtain the verification condition generator for the triple $\{ P \} \; a \; \{ Q \}$:

```
Definition vcgen (P: assertion) (a: acmd) (Q: assertion) : Prop :=
  aimp P (wp a Q) /\ vcg a Q.
```

This v.c.gen. is correct in the following sense:

```
Lemma vcg_correct:
  forall a Q, vcg a Q -> triple (wp a Q) (erase a) Q.

Theorem vcgen_correct:
  forall P a Q, vcgen P a Q -> triple P (erase a) Q.
```

The verification condition generator

```
Theorem vcgen_correct:
  forall P a Q, vcgen P a Q -> triple P (erase a) Q.
```

What did we gain?

We replaced deduction by computation:

- Deducing `triple P (erase a) Q` requires some guesswork (to find loop invariants; to know when to apply the rule of consequence).
- Computing `vcgen P a Q` is purely mechanical.

Moreover, the proposition `vcgen P a Q` is a "plain" logic formula, where the command `a` and its semantics no longer appear. It could therefore be fed to any automated or semi-automated prover.

An example of verification

Consider the following annotated IMP program *a*:

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

and the following precondition *Pre*, loop invariant *Inv* and postcondition *Post*:

$$\begin{aligned} Pre &= \lambda s.\ s(\mathtt{a}) \geq 0 \land s(\mathtt{b}) > 0 \\ Inv &= \lambda s.\ s(\mathtt{r}) \geq 0 \land s(\mathtt{b}) > 0 \land s(\mathtt{a}) = s(\mathtt{b}) \times s(\mathtt{q}) + s(\mathtt{r}) \\ Post &= \lambda s.\ s(\mathtt{q}) = s(\mathtt{a}) / s(\mathtt{b}) \end{aligned}$$

Let us show $\{\ Pre\ \}\ \mathtt{erase}(a)\ \{\ Post\ \}$ using the v.c.gen... [demo]

Exercise 2

Write a v.c.gen. for strong Hoare triples (those that guarantee termination). The language of annotated commands is extended so that every `while` loop carries both an invariant P and an expression e_m that is the decreasing measure for this loop.

Annotated commands:

$c ::= \texttt{while}\ b\ \texttt{do}\ \{P, e_m\}\ c\ \texttt{done}$	loop + invariant + measure
$\mid \texttt{assert}(P)$	explicit assertion
$\mid \texttt{skip} \mid x := e \mid c_1; c_2$	as in IMP
$\mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2$	as in IMP

Axiomatic semantics and program proof

7 Axiomatic semantics (Hoare logic)

8 Automatic generation of verification conditions (VCgen)

9 Computing within proofs

10 Further reading

Computation vs. deduction

Henri Poincaré writing about Peano-style proofs of $2 + 2 = 4$:

> *Ce n'est pas une démonstration proprement dite [. . .], c'est une vérification. [. . .] La vérification diffère précisément de la véritable démonstration, parce qu'elle est purement analytique et parce qu'elle est stérile.*

[This is not a proper demonstration, it is a mere verification. Verification differs from true demonstration because it is purely analytical and because it is sterile.]

Computation vs. deduction

Omitting computational steps in Coq's proofs via the conversion rule:

$$\frac{\Gamma \vdash a : P \qquad P \stackrel{\beta\iota\delta}{\equiv} Q}{\Gamma \vdash a : Q}\ [\text{conv}]$$

$\stackrel{\beta\iota\delta}{\equiv}$ represents computations: reducing function applications, unfolding definitions and fixpoints, simplifying pattern-matchings.

Using convertibility in proofs

```
Theorem refl_equal: forall (A: Type) (x: A), x = x.

Fixpoint plus (a b: nat) {struct a} : nat :=
  match a with O => b | S a' => S (plus a' b) end.
```

By instanciation, `refl_equal nat 4` proves `4 = 4`.

But it also proves `plus 2 2 = 4` because this proposition and `4 = 4` are convertible.

Likewise, `plus O x = x` is proved trivially for any `x`.

However, `plus x O = x` requires a proof by induction on `x` ...

Proofs by reflection

A step of Gonthier and Werner's proof of the 4-color theorem involves checking 4-colorability for a large number of elementary graphs $g_1 \dots g_N$.

```
Definition is_colorable (g:graph): Prop :=
  ∃ f: vertices(g) -> {1,2,3,4}.
  ∀(a,b) ∈ edges(g). f(a) ≠ f(b).
```

The naive approach: for each graph g_i of interest, provide (manually) the function f and prove (manually or using tactics) that it is a coloring.

Proofs by reflection

The "reflection" approach: invoke a proved decision procedure, thus replacing deductions by computations.

```
Definition colorable (g:graph): bool :=
  (* combinatorial search for a 4-coloring *)

Theorem colorable_is_sound:
  forall (g:graph), colorable g = true -> is_colorable g.
```

The proof that g_i is colorable is now just the term `colorable_is_sound` g_i `(refl_equal bool true)`.

Internally, the checker reduces (`colorable` g_i) to `true`
$\rightarrow$ large amounts of computations, but still much faster than the equivalent amounts of deductions.

Axiomatic semantics and program proof

7 Axiomatic semantics (Hoare logic)

8 Automatic generation of verification conditions (VCgen)

9 Computing within proofs

10 Further reading

Program proof in “the real world”

Hoare-like logics are at the core of industrial-strength program provers for realistic languages, e.g.

- ESC/Java
- Boogie (for C#)
- Caveat, Frama-C (for C)

Separation logic

For programs with pointers (e.g. in C), program proof entails much reasoning about separation (non-aliasing) between mutable data structures.

Separation logic is an extension of Hoare logic that makes it easy to reason both about

- the current contents of a mutable data structure
- separation between different data structures.

Adding pointers to IMP

The shape of a semantics for a language with pointers:

Values: $v ::= \mathtt{int}(n) \mid \mathtt{ptr}(\ell)$

Environments: $e ::= x \mapsto v$

Heaps: $h ::= \ell \mapsto v$

States: $s ::= (e, h)$

Pointer dereference, pointer assignment:

$$\frac{[\![a]\!]\ (e, h) = \mathtt{ptr}(\ell)}{(x := *a), (e, h) \Rightarrow (e[x \leftarrow h(\ell)], h)}$$

$$\frac{[\![a]\!]\ (e, h) = \mathtt{ptr}(\ell) \qquad [\![b]\!]\ (e, h) = v}{(*a := b), (e, h) \Rightarrow (e, h[\ell \leftarrow v])}$$

Assertions of separation logic

"The heap is empty"

$$\texttt{empty} \stackrel{\text{def}}{=} \lambda(e,h).\ \texttt{dom}(h) = \emptyset$$

"*a* is a valid pointer"

$$a \mapsto \_ \stackrel{\text{def}}{=} \lambda(e,h).\ \exists \ell.\ [\![a]\!]\ (e,h) = \texttt{ptr}(\ell) \wedge \texttt{dom}(h) = \{\ell\}$$

"*a* points to *b*"

$$a \mapsto b \stackrel{\text{def}}{=} \lambda(e,h).\ \exists \ell.$$
$$[\![a]\!]\ (e,h) = \texttt{ptr}(\ell) \wedge \texttt{dom}(h) = \{\ell\} \wedge h(\ell) = [\![b]\!]\ (e,h)$$

Separating conjunction

$$P \star Q \stackrel{\text{def}}{=} \lambda(e,h).\ \exists h_1, h_2.$$
$$h = h_1 \cup h_2 \wedge \texttt{dom}(h_1) \cap \texttt{dom}(h_2) = \emptyset \wedge P(h_1) \wedge Q(h_2)$$

Some rules of separation logic

$$\{\,\texttt{empty}\,\}\ (x := \texttt{alloc})\ \{\,x \mapsto \_\,\} \qquad \{\,a \mapsto \_\,\}\ (*a := b)\ \{\,a \mapsto b\,\}$$

The rules for $\{\,P\,\}\ c\ \{\,Q\,\}$ enforce that any heap location modified by c is in the "footprint" of the precondition P. This justifies the frame rule:

$$\frac{\{\,P\,\}\ c\ \{\,Q\,\}}{\{\,P \star R\,\}\ c\ \{\,Q \star R\,\}}$$

which enables local reasoning on the part of the heap that c actually modifies, and guarantees that the other part (characterized by R) does not change.

Concurrent separation logic

Such separation guarantees also enable reasoning over shared-memory concurrency where different threads concurrently modify independent parts of the heap:

$$\frac{\{\, P_1 \,\}\; c_1 \;\{\, Q_1 \,\} \qquad \{\, P_2 \,\}\; c_2 \;\{\, Q_2 \,\}}{\{\, P_1 \star P_2 \,\}\; (c_1 \mid c_2) \;\{\, Q_1 \star Q_2 \,\}}$$

(For mechanizations of separation logic, see for instance Marti et al, Tuch et al, Myreen & Gordon. For concurrent separation logic, see the VST project of Appel et al.)

Part III

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,
   1. either the virtual machine instructions are interpreted (efficiently)
   2. or they are further translated to machine code (JIT).

Compilation to a virtual machine

11 The IMP virtual machine

12 Compiling IMP programs to virtual machine code

13 Notions of semantic preservation

14 Semantic preservation for our compiler

15 Further reading

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 store s: a mapping from variable names to integer values.
- The stack σ: a list of integer values (used to store intermediate results temporarily).

The instruction set

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

By default, each instruction increments pc by 1.

Exception: branch instructions increment it by $1 + \delta$.
(δ is a branch offset relative to the next instruction.)

Example

stack	ϵ	12	1 12	13	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	$\texttt{var}(x)$;	$\texttt{const}(1)$;	$\texttt{add}$;	$\texttt{setvar}(x)$;	$\texttt{branch}(-5)$

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)%type.

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 states: $pc = 0$, initial store, empty stack.
- Final 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

11 The IMP virtual machine

12 Compiling IMP programs to virtual machine code

13 Notions of semantic preservation

14 Semantic preservation for our compiler

15 Further reading

Compilation scheme for expressions

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

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

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

(= translation to "reverse Polish notation".)

Compilation scheme for conditions

The code $\texttt{comp}(b, \delta)$ for a boolean expression should:

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

$$
\begin{array}{rcl}
\texttt{comp}(e_1 = e_2,\ \delta) & = & \texttt{comp}(e_1); \texttt{comp}(e_2); \texttt{bne}(\delta) \\
\texttt{comp}(e_1 < e_2,\ \delta) & = & \texttt{comp}(e_1); \texttt{comp}(e_2); \texttt{bge}(\delta)
\end{array}
$$

Example

$\texttt{comp}(x + 1 < y - 2,\ \delta) =$

$\texttt{var}(x); \texttt{const}(1); \texttt{add};$	(compute $x + 1$)
$\texttt{var}(y); \texttt{const}(2); \texttt{sub};$	(compute $y - 2$)
$\texttt{bge}(\delta)$	(branch if $\geq$)

Exercise 3

Extend the compilation scheme for the richer language of boolean expressions from Exercise 1:

Boolean expressions (conditions):

$$\begin{array}{rll} b ::= & e_1 = e_2 & \text{equality test} \\ \mid & e_1 < e_2 & \text{less-than test} \\ \mid & \texttt{not}(b_1) & \text{negation} \\ \mid & b_1 \ \texttt{and} \ b_2 & \text{conjunction} \end{array}$$

Feel free to add new instructions to the machine, such as "branch if equal" and "branch if less than", if it helps generate more efficient code.

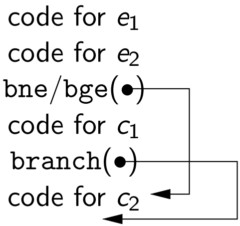
Try to "short-circuit" conjunctions: in b_1 `and` b_2, if b_1 evaluated to `false`, there is no need to evaluate b_2.

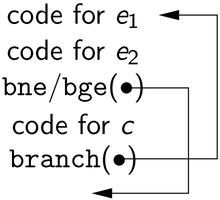
Compilation scheme for commands

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

$$\begin{array}{rcl}
\texttt{comp}(\texttt{skip}) & = & \epsilon \\
\texttt{comp}(x := e) & = & \texttt{comp}(e); \texttt{setvar}(x) \\
\texttt{comp}(c_1; c_2) & = & \texttt{comp}(c_1); \texttt{comp}(c_2)
\end{array}$$

Compilation scheme for commands





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

Compiling whole program

The compilation of a program c is the code

$$\texttt{compile}(c) = \texttt{comp}(c); \texttt{halt}$$

Example

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

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

Coq mechanization of the compiler

As recursive functions:

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

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

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

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

These functions can be executed from within Coq, or extracted to executable Caml code.

Compiler verification

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.

Compilation to a virtual machine

11 The IMP virtual machine

12 Compiling IMP programs to virtual machine code

13 Notions of semantic preservation

14 Semantic preservation for our compiler

15 Further reading

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:

$$\textit{observable behavior} ::= \texttt{terminates}(s) \mid \texttt{diverges} \mid \texttt{goeswrong}$$

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

Bisimulation (equivalence)

$$\mathcal{B}(P_1) = \mathcal{B}(P_2)$$

Often too strong in practice (see next slides).

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.

Example: a C compiler chooses one evaluation order for expressions among the several permitted by the C semantics.

Backward simulation for correct programs

$$\texttt{goeswrong} \notin \mathcal{B}(P_1) \implies \mathcal{B}(P_1) \supseteq \mathcal{B}(P_2)$$

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

`x := 1 / y; x := 42`	optimized to	`x := 42`
(goes wrong if $\mathrm{y} = 0$)		(always terminates normally)

Let *Spec* be the functional specification of a program (a set of correct behaviors, not containing `goeswrong`).

Lemma

If "backward simulation for correct programs" holds, and P_1 satisfies Spec (i.e. $\mathcal{B}(P_1) \subseteq Spec$), then P_2 satisfies Spec (i.e. $\mathcal{B}(P_2) \subseteq Spec$).

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 (the source program) than on an execution of P_2 (the compiled code).

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

Forward simulation for correct programs:

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

(P_2 has all the behaviors of P_1, but maybe more.)

Determinism to the rescue

Lemma

If P_2 is deterministic ($\mathcal{B}(P_2)$ is a singleton), then

- *"forward simulation" implies "backward simulation"*
- *"forward simulation for correct programs" implies "backward simulation for correct programs"*

$\rightarrow$ Our plan for verifying a compiler:

- Prove "forward simulation for correct programs" between source and compiled codes.
- Argue that the target language (machine code) is deterministic.
- Conclude that all functional specifications are preserved by compilation.

Compilation to a virtual machine

11 The IMP virtual machine

12 Compiling IMP programs to virtual machine code

13 Notions of semantic preservation

14 Semantic preservation for our compiler

15 Further reading

Compilation of expressions

Remember the "contract" for the code $\mathtt{comp}(e)$: it should

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

More formally: $\mathtt{comp}(e) : \ (0, \sigma, s) \xrightarrow{*} (|comp(e)|, ([\![e]\!]\ s).\sigma, s)$.

To make this result more usable and permit a proof by induction, need to strengthen this result to codes of the form $C_1; \mathtt{comp}(e); C_2$.

Compilation of expressions

```
Lemma compile_expr_correct:
  forall s e pc stk C1 C2,
  pc = length C1 ->
  star (transition (C1 ++ compile_expr e ++ C2))
       (pc, stk, s)
       (pc + length (compile_expr e), eval_expr s e :: stk, s).
```

Proof: structural induction over the expression e, using associativity of ++ (list concatenation) and + (integer addition).

Outline of the proof

Base cases (variables, constants): trivial. An inductive case: $e = e_1 + e_2$. Write $v_1 = [\![e_1]\!]\ s$ and $v_2 = [\![e_2]\!]\ s$. By induction hypothesis (twice),

$$C_1; \texttt{comp}(e_1); (\texttt{comp}(e_2); \texttt{add}; C_2) :$$
$$(|C_1|, \sigma, s) \xrightarrow{*} (|C_1| + |\texttt{comp}(e_1)|, v_1.\sigma, s)$$
$$(C_1; \texttt{comp}(e_1)); \texttt{comp}(e_2); (\texttt{add}; C_2) :$$
$$(|C_1; \texttt{comp}(e_1)|, v_1.\sigma, s) \xrightarrow{*} (|C_1; \texttt{comp}(e_1)| + |\texttt{comp}(e_2)|, v_2.v_1.\sigma, s)$$

Combining with an add transition, we obtain:

$$C_1; (\texttt{comp}(e_1); \texttt{comp}(e_2); \texttt{add}); C_2 :$$
$$(|C_1|, \sigma, s) \xrightarrow{*} (|C_1; \texttt{comp}(e_1); \texttt{comp}(e_2)| + 1, (v_1 + v_2).\sigma, s)$$

which is the desired result since $\texttt{comp}(e_1 + e_2) = \texttt{comp}(e_1); \texttt{comp}(e_2); \texttt{add}$.

Historical note

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

- First published proof of compiler correctness.
  (McCarthy and Painter, 1967).
- First mechanized proof of compiler correctness.
  (Milner and Weyrauch, 1972, using Stanford LCF).

John McCarthy
James Painter[1]

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS[2]

1. **Introduction.** This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language.

The definition of correctness, the formalism used to express the description of source language, object language and compiler, and the methods of proof are all intended to serve as prototypes for the more complicated task of proving the correctness of usable compilers. The ultimate goal, as outlined in references [1], [2], [3] and [4] is to make it possible to use a computer to check proofs that compilers are correct.

Mathematical Aspects of Computer Science, 1967

3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch

Computer Science Department
Stanford University

Abstract

We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is presented only in outline. However, we present in full the expression-compiling part of the algorithm. More than half of the complete proof has been machine checked, and we anticipate no difficulty with the remainder. We discuss our experience in conducting the proof, which indicates that a large part of it may be automated to reduce the human contribution.

Machine Intelligence (7), 1972.

APPENDIX 2: command sequence for McCarthy-Painter lemma

```
GOAL ∀e sp,Iswfse e::MT(compe e,sp)≡svof(sp)|((MSE(e,svof sp))&pdof(sp)),
     ∀e,Iswfse e::Iswft(compe e)≡TT,
     ∀e,Iswfse e::(count(compe e)=0)≡TT;

TRY 1 INDUCT 56;
 TRY 1 SIMPL;
 LABEL INDHYP;
 TRY 2 ABSTR;
  TRY 1 CASES wfsefun(f,e);
  LABEL TT;
   TRY 1 CASES type e=_N;
    TRY 1 SIMPL BY ,FMT1,,FMSE,,FCOMPE,,FISWFT1,,FCOUNT;
    TRY 2;SS-,TT;SIMPL,TT;QED;
    TRY 3 CASES type e=_E;
      TRY 1 SUBST ,FCOMPE;
      SS-,TT;SIMPL,TT;USE BOTH3 -;SS+,TT;
      INCL-,1;SS+-;INCL--,2;SS+-;INCL---,3;SS+-;
       TRY 1 CONJ;
        TRY 1 SIMPL;
         TRY 1 USE COUNT1;
          TRY 1;
          APPL ,INDHYP+2,arg1of e;
          LABEL CARG1;
          SIMPL-;QED;
          TRY 2 USE COUNT1;
           TRY 1;
```

(Even the proof scripts look familiar!)

Compilation of conditions

The code comp(b, δ) for a boolean expression should:

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

```
Lemma compile_bool_expr_correct:
  forall s e pc stk ofs C1 C2,
  pc = length C1 ->
  star (transition (C1 ++ compile_bool_expr e ofs ++ C2))
       (pc, stk, s)
       (pc + length (compile_bool_expr e ofs)
            + (if eval_bool_expr s e then 0 else ofs),
        stk, s).
```

Compilation of commands, terminating case

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

We use the natural semantics for commands because, like the compilation scheme itself, it follows the structure of commands. For the terminating case:

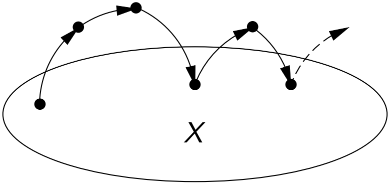
```
Lemma compile_cmd_correct_terminating:
  forall s c s', exec s c s' ->
  forall stk pc C1 C2,
  pc = length C1 ->
  star (transition (C1 ++ compile_cmd c ++ C2))
       (pc, stk, s)
       (pc + length (compile_cmd c), stk, s').
```

(By induction on a derivation of `exec s c s'`.)

Compilation of commands, diverging case

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

For all $(C, pc, \sigma, s) \in X$, there exists pc', σ', s' such that $C : (pc, \sigma, s) \xrightarrow{+} (pc', \sigma', s')$ and $(C, pc', \sigma', s') \in X$.




```
Lemma compile_cmd_correct_diverging:
  forall s c , execinf s c ->
  forall pc stk C1 C2,
  pc = length C1 ->
  infseq (transition (C1 ++ compile_cmd c ++ C2)) (pc, stk, s).
```

This completes the proof of forward simulation for correct programs.

Compilation to a virtual machine

11 The IMP virtual machine

12 Compiling IMP programs to virtual machine code

13 Notions of semantic preservation

14 Semantic preservation for our compiler

15 Further reading

Further reading

Other examples of verification of nonoptimizing compilers producing virtual machine code:

- Klein and Nipkow (subset of Java $\rightarrow$ subset of the JVM)
- Bertot (for the IMP language)
- Grall and Leroy (CBV λ-calculus $\rightarrow$ modern SECD).

The techniques presented in this lecture do scale up to compilers from realistic languages (e.g. C) to "real" machine code.
(See the CompCert project.)

Part IV

An optimizing program transformation

Compiler optimizations

Automatically transform the programmer-supplied code into equivalent code that

- Runs faster
  - Removes redundant or useless computations.
  - Use cheaper computations (e.g. `x * 5` $\rightarrow$ `(x << 2) + x`)
  - Exhibits more parallelism (instruction-level, thread-level).
- Is smaller
  (For cheap embedded systems.)
- Consumes less energy
  (For battery-powered systems.)
- Is more resistant to attacks
  (For smart cards and other secure systems.)

Dozens of compiler optimizations are known, each targeting a particular class of inefficiencies.

Compiler optimization and static analysis

Some optimizations are unconditionally valid, e.g.:

$$
\begin{aligned}
\mathtt{x} * 2 &\;\to\; \mathtt{x} + \mathtt{x} \\
\mathtt{x} * 4 &\;\to\; \mathtt{x} \ll 2
\end{aligned}
$$

Most others apply only if some conditions are met:

$$
\begin{array}{rcll}
\mathtt{x} \,/\, 4 & \to & \mathtt{x} >> 2 & \text{only if } \mathtt{x} \geq 0 \\
\mathtt{x} + 1 & \to & 1 & \text{only if } \mathtt{x} = 0 \\
\mathtt{if}\ \mathtt{x} < \mathtt{y}\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 & \to & c_1 & \text{only if } \mathtt{x} < \mathtt{y} \\
\mathtt{x} := \mathtt{y} + 1 & \to & \mathtt{skip} & \text{only if } \mathtt{x} \text{ unused later}
\end{array}
$$

$\to$ need a static analysis prior to the actual code transformation.

Static analysis

Determine some properties of all concrete executions of a program.

Often, these are properties of the values of variables at a given program point:

$$\mathrm{x} = n \qquad \mathrm{x} \in [n, m] \qquad \mathrm{x} = e \qquad n \leq a.\mathrm{x} + b.\mathrm{y} \leq m$$

Requirements:

- The inputs to the program are unknown.
- The analysis must terminate.
- The analysis must run in reasonable time and space.

Running example: dead code elimination via liveness analysis

Remove assignments $x := e$, turning them into `skip`, whenever the variable x is never used later in the program execution.

Example

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

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

Builds on a static analysis called liveness analysis.

An optimizing program transformation

16 Liveness analysis

17 Dead code elimination

18 Semantic preservation

19 Advanced topic: register allocation

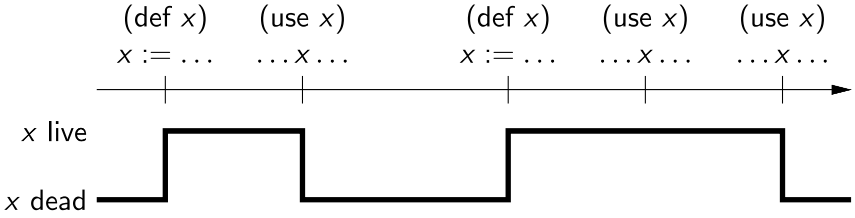
Notions of liveness

A variable is dead at a program point if its value is not used later in any execution of the program:

- either the variable is not mentioned again before going out of scope
- or it is always redefined before further use.

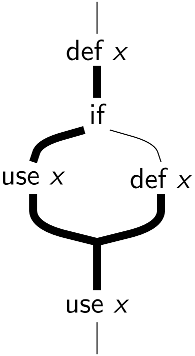
A variable is live if it is not dead.

Easy to compute for straight-line programs (sequences of assignments):



Notions of liveness

Liveness information is more delicate to compute in the presence of conditionals and loops:



Conservatively over-approximate liveness, assuming all `if` conditionals can be true or false, and all `while` loops are taken 0 or several times.

Liveness equations

Given a set L of variables live "after" a command c, write $\texttt{live}(c, L)$ for the set of variables live "before" the command.

(A form of reverse, abstract execution.)

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

Fixpoints, a.k.a "the recurring problem"

Consider $F = \lambda X.\ L \cup FV(b) \cup \mathtt{live}(c, X)$.

For `while` loops, we need to compute a fixpoint of F, i.e. an X such that $F(X) = X$, preferably the smallest.

The mathematician's approach: notice that

- F is increasing;
- we can restrict us to subsets of the set V of all variables mentioned in the program;
- the $\subset$ ordering over these sets is well-founded.

Therefore, the sequence $\emptyset, F(\emptyset), \ldots, F^n(\emptyset), \ldots$ eventually stabilizes to a set that is the smallest fixpoint of F.

Fixpoints, a.k.a "the recurring problem"

$$F = \lambda X.\ L \cup FV(b) \cup \texttt{live}(c, X)$$

The engineer's approach:

- Compute $F(\emptyset), F(F(\emptyset)), \ldots, F^N(\emptyset)$ up to some fixed N.
- If a fixpoint is found, great.
- Otherwise, return a safe over-approximation (in our case, $L \cup FV(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done})$).

A compromise between analysis speed and analysis precision.

Both approaches can be mechanized in Coq, but the mathematician's requires advanced features not covered here, so we'll use the engineer's approach.

Liveness analysis as a Coq function

```
Module VS := FSetAVL.Make(Nat_as_OT).    (* sets of variables *)

Fixpoint live (c: cmd) (L: VS.t) {struct c} : VS.t :=
  match c with
  | Cskip => L
  | Cassign x e =>
      if VS.mem x L
      then VS.union (VS.remove x L) (fv_expr e)
      else L
  | Cseq c1 c2 => live c1 (live c2 L)
  | Cifthenelse b c1 c2 =>
      VS.union (fv_bool_expr b) (VS.union (live c1 L) (live c2 L))
  | Cwhile b c =>
      let a' := VS.union (fv_bool_expr b) L in
      let default := VS.union (fv_cmd (Cwhile b c)) L in
      fixpoint (fun x => VS.union a' (live c x)) default
  end.
```

An optimizing program transformation

16 Liveness analysis

17 Dead code elimination

18 Semantic preservation

19 Advanced topic: register allocation

Dead code elimination

The program transformation eliminates assignments to dead variables:

$x := e$ becomes `skip` if x is not live "after" the assignment

Presented as a function `dce` : `cmd` $\rightarrow$ `VS.t` $\rightarrow$ `cmd`
taking the set of variables live "after" as second parameter
and maintaining it during its traversal of the command.

Dead code elimination in Coq

```
Fixpoint dce (c: cmd) (L: VS.t) {struct c}: cmd :=
  match c with
  | Cskip => Cskip
  | Cassign x e =>
      if VS.mem x L then Cassign x e else Cskip
  | Cseq c1 c2 =>
      Cseq (dce c1 (live c2 L)) (dce c2 L)
  | Cifthenelse b c1 c2 =>
      Cifthenelse b (dce c1 L) (dce c2 L)
  | Cwhile b c =>
      Cwhile b (dce c (live (Cwhile b c) L))
  end.
```

Example

Consider again Euclidean division:

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

If q is not live "after" ($q \notin L$), it is not live throughout this program either. `dce` c L then slices away all computations of q, producing

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

If q is live "after", the program is unchanged.

An optimizing program transformation

16 Liveness analysis

17 Dead code elimination

18 Semantic preservation

19 Advanced topic: register allocation

The semantic meaning of liveness

What does it mean, semantically, for a variable x to be live at some program point?

Hmmm...

The semantic meaning of liveness

What does it mean, semantically, for a variable x to be live at some program point?

Hmmm. . .

What does it mean, semantically, for a variable x to be dead at some program point?

That its precise value has no impact on the rest of the program execution!

Liveness as an information flow property

Consider two executions of the same command c in different initial states:

$$\begin{array}{rcl} c, s_1 & \Rightarrow & s_2 \\ c, s_1' & \Rightarrow & s_2' \end{array}$$

Assume that the initial states agree on the variables $\texttt{live}(c, L)$ live "before" c:

$$\forall x \in \texttt{live}(c, L), \quad s_1(x) = s_1'(x)$$

Then, the two executions terminate on final states that agree on the variables L live "after" c:

$$\forall x \in L, \quad s_1(x) = s_1'(x)$$

The proof of semantic preservation for dead-code elimination follows this pattern, relating executions of c and `dce` c L instead.

Agreement and its properties

```
Definition agree (L: VS.t) (s1 s2: state) : Prop :=
  forall x, VS.In x L -> s1 x = s2 x.
```

Agreement is monotone w.r.t. the set of variables L:

```
Lemma agree_mon:
  forall L L' s1 s2,
  agree L' s1 s2 -> VS.Subset L L' -> agree L s1 s2.
```

Expressions evaluate identically in states that agree on their free variables:

```
Lemma eval_expr_agree:
  forall L s1 s2, agree L s1 s2 ->
  forall e, VS.Subset (fv_expr e) L ->
                  eval_expr s1 e = eval_expr s2 e.
Lemma eval_bool_expr_agree:
  forall L s1 s2, agree L s1 s2 ->
  forall b, VS.Subset (fv_bool_expr b) L ->
                  eval_bool_expr s1 b = eval_bool_expr s2 b.
```

Agreement and its properties

Agreement is preserved by parallel assignment to a variable:

```
Lemma agree_update_live:
  forall s1 s2 L x v,
  agree (VS.remove x L) s1 s2 ->
  agree L (update s1 x v) (update s2 x v).
```

Agreement is also preserved by unilateral assignment to a variable that is dead “after”:

```
Lemma agree_update_dead:
  forall s1 s2 L x v,
  agree L s1 s2 -> ~VS.In x L ->
  agree L (update s1 x v) s2.
```

Forward simulation for dead code elimination

For terminating source programs:

```
Lemma dce_correct_terminating:
  forall s c s', exec s c s' ->
  forall L s1,
  agree (live c L) s s1 ->
  exists s1', exec s1 (dce c L) s1' /\ agree L s' s1'.
```

(Proof: a simple induction on the derivation of `exec s c s'`.)

$$\begin{tikzcd} s \arrow[r, dash, "\texttt{agree (live } c\ L\texttt{)}"] \arrow[d, "\texttt{exec } c"'] & s_1 \arrow[d, dashed, "\texttt{exec (dce } c\ L\texttt{)}"] \\ s' \arrow[r, dash, dashed, "\texttt{agree } L"] & s_1' \end{tikzcd}$$

Forward simulation for dead code elimination

The result extends simply to diverging source programs:

```
Lemma dce_correct_diverging:
  forall s c, execinf s c ->
  forall L s1,
  agree (live c L) s s1 -> execinf s1 (dce c L).
```

$$\begin{tikzcd}
s \arrow[rr, dash, "\texttt{agree}\ (\texttt{live}\ c\ L)"] \arrow[d, "\texttt{execinf}\ c"'] & & s_1 \arrow[d, dashed, "\texttt{execinf}\ (\texttt{dce}\ c\ L)"] \\
\infty & & \infty
\end{tikzcd}$$

An optimizing program transformation

16 Liveness analysis

17 Dead code elimination

18 Semantic preservation

19 Advanced topic: register allocation

The register allocation problem

Place the variables used by the program (in unbounded number) into:

- either hardware registers
  (very fast access, but available in small quantity)
- or memory locations (generally allocated on the stack)
  (available in unbounded quantity, but slower access)

Try to maximize the use of hardware registers.

A crucial step for the generation of efficient machine code.

Approaches to register allocation

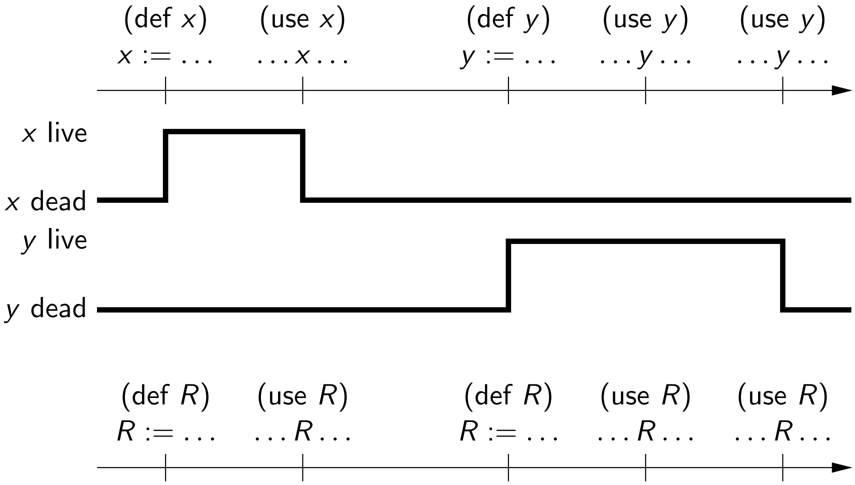
Naive approach (injective allocation):

- Assign the N most used variables to the N available registers.
- Assign the remaining variables to memory locations.

Optimized approach (non-injective allocation):

- Notice that two variables can share a register
  as long as they are not simultaneously live.

Example of register sharing



Register allocation for IMP

Properly done:

1. Break complex expressions by introducing temporaries.
   (E.g. `x = (a + b) * y` becomes `tmp = a + b; x = tmp * y`.)
2. Translate IMP to a variant IMP′ that uses registers ∪ memory locations instead of variables.

Simplified as follows in this lecture:

1. Do not break expressions.
2. Translate from IMP to IMP, by renaming identifiers.
   (Convention: low-numbered identifiers ≈ hardware registers.)

The program transformation

Assume given a "register assignment" $f : \texttt{id} \to \texttt{id}$.

The program transformation consists of:

- Renaming variables: all occurrences of x become $f\ x$.
- Dead code elimination:

$$x := a \quad \longrightarrow \quad \texttt{skip} \quad \text{if } x \text{ is dead "after"}$$

- Coalescing:

$$x := y \quad \longrightarrow \quad \texttt{skip} \quad \text{if } f\ x = f\ y$$

Correctness conditions on the register assignment

Clearly, not all register assignments f preserve semantics.

Example: assume $f\ x = f\ y = f\ z = R$

```
x := 1;                         R := 1;
y := 2;           ---->         R := 2;
z := x + y;                     R := R + R;
```

Computes 4 instead of 3 ...

What are sufficient conditions over f? Let's discover them by reworking the proof of dead code elimination.

Agreement, revisited

```
Definition agree (L: VS.t) (s1 s2: state) : Prop :=
  forall x, VS.In x L -> s1 x = s2 (f x).
```

An expression and its renaming evaluate identically in states that agree on their free variables:

```
Lemma eval_expr_agree:
  forall L s1 s2, agree L s1 s2 ->
  forall a, VS.Subset (fv_aexp a) L ->
  eval_expr s1 a = eval_expr s2 (rename_expr a).
Lemma eval_bool_expr_agree:
  forall L s1 s2, agree L s1 s2 ->
  forall b, VS.Subset (fv_bexp b) L ->
  eval_bool_expr s1 b = eval_bool_expr s2 (rename_bool_expr b).
```

Agreement, revisited

As before, agreement is monotonic w.r.t. the set of variables L:

```
Lemma agree_mon:
  forall L L' s1 s2,
  agree L' s1 s2 -> VS.Subset L L' -> agree L s1 s2.
```

As before, agreement is preserved by unilateral assignment to a variable that is dead "after":

```
Lemma agree_update_dead:
  forall s1 s2 L x v,
  agree L s1 s2 -> ~VS.In x L ->
  agree L (update s1 x v) s2.
```

Agreement, revisited

Agreement is preserved by parallel assignment to a variable x and its renaming $f\ x$, but only if f satisfies a non-interference condition (in red below):

```
Lemma agree_update_live:
  forall s1 s2 L x v,
  agree (VS.remove x L) s1 s2 ->
  (forall z, VS.In z L -> z <> x -> f z <> f x) ->
  agree L (update s1 x v) (update s2 (f x) v).
```

A special case for moves

Consider a variable-to-variable copy $x ::= y$.
In this case, the value v assigned to x is not arbitrary, but known to be `s1` y. We can, therefore, weaken the non-interference criterion:

```
Lemma agree_update_move:
  forall s1 s2 L x y,
  agree (VS.union (VS.remove x L) (VS.singleton y)) s1 s2 ->
  (forall z, VS.In z L -> z <> x -> z <> y -> f z <> f x) ->
  agree L (update s1 x (s1 y)) (update s2 (f x) (s2 (f y))).
```

This makes it possible to assign x and y to the same location, even if x and y are simultaneously live.

The interference graph

The various non-interference constraints $f\ x \neq f\ y$ can be represented as an interference graph:

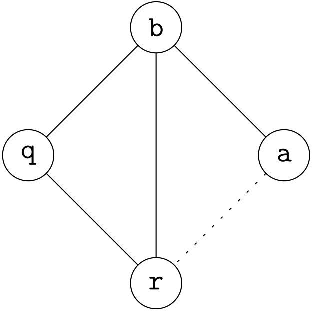
- Nodes = program variables.
- Undirected edge between x and y =
  x and y cannot be assigned the same location.

Chaitin's algorithm to construct this graph:

- For each move $x ::= y$, add edges between x and every variable z live "after" except x and y.
- For each other assignment $x ::= a$, add edges between x and every variable z live "after" except x.

Example of an interference graph

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



(Full edge = interference; dotted edge = preference.)

Register allocation as a graph coloring problem

(G. Chaitin, 1981; P. Briggs, 1987)

Color the interference graph, assigning a register or memory location to every node;

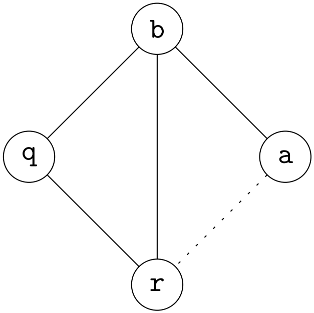
under the constraint that the two ends of an interference edge have different colors;

with the objective to

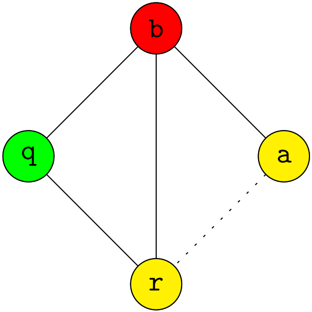
- minimize the number (or total weight) of nodes that are colored by a memory location
- maximize the number of preference edges whose ends have the same color.

(A NP-complete problem in general, but good linear-time heuristics exist.)

Example of coloring



Example of coloring




```
yellow := yellow;
green := 0;
while red <= yellow do
  yellow := yellow - red;
  green := green + 1
end
```

What needs to be proved in Coq?

Full compiler proof:
formalize and prove correct a good graph coloring heuristic.

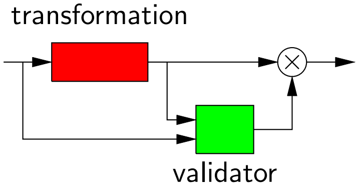
George and Appel's *Iterated Register Coalescing* $\approx 6\,000$ lines of Coq.

Validation a posteriori:
invoke an external, unproven oracle to compute a candidate allocation;
check that it satisfies the non-interference conditions;
abort compilation if the checker says `false`.

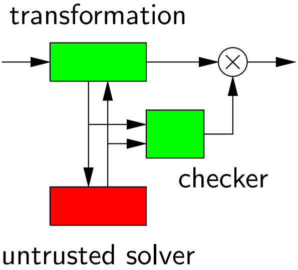
The verified transformation–verified validation spectrum

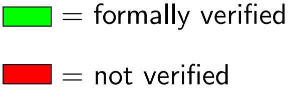
Verified transformation

Verified translation validation



External solver with verified validation





Validating candidate allocations in Coq

It is easy to write a Coq boolean-valued function

```
correct_allocation: (id -> id) -> cmd -> VS.t -> bool
```

that returns `true` only if the expected non-interference properties are satisfied.

Semantic preservation

The proofs of forward simulation that we did for dead code elimination then extend easily, under the assumption that `correct_allocation` returns true:

```
Theorem regalloc_correct_terminating:
  forall s c s', exec s c s' ->
  forall L s1, agree (live c L) s s1 ->
  correct_allocation c L = true ->
  exists s1', exec s1 (regalloc c L) s1' / agree L st' st1'.
```

Part V

State of the art and perspectives

State of the art

The approaches introduced in this lecture do scale (albeit painfully) to real-world systems (software and hardware).

A few examples where the use of

- proof assistants
- techniques firmly grounded in mathematical semantics

have produced breakthroughs in the area of formal methods

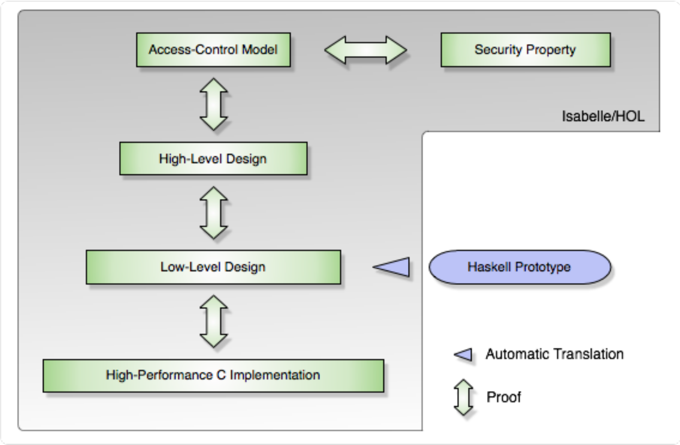
- w.r.t. the size, complexity and realism of the systems that were verified;
- w.r.t. the strength of the guarantees obtained.

(More examples in the handout.)

The L4.verified project

(G. Klein et al, NICTA)

Formal verification of the seL4 secure micro-kernel, all the way down to the actual, hand-optimized C implementation of seL4.



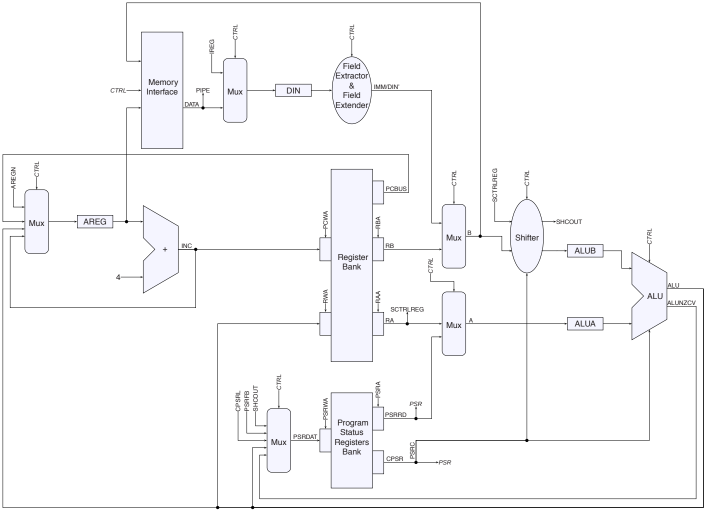
Verification of Java & Java Card components

Several academic projects on verifications of subsets of Java, the JVM and JCVM machines, bytecode verifiers, APIs and security architecture:

- Ninja (TU Munich, T. Nipkow, G. Klein et al)
- Jakarta (INRIA, G. Barthe et al)
- The Kestrel Institute project (A. Coglio et al)

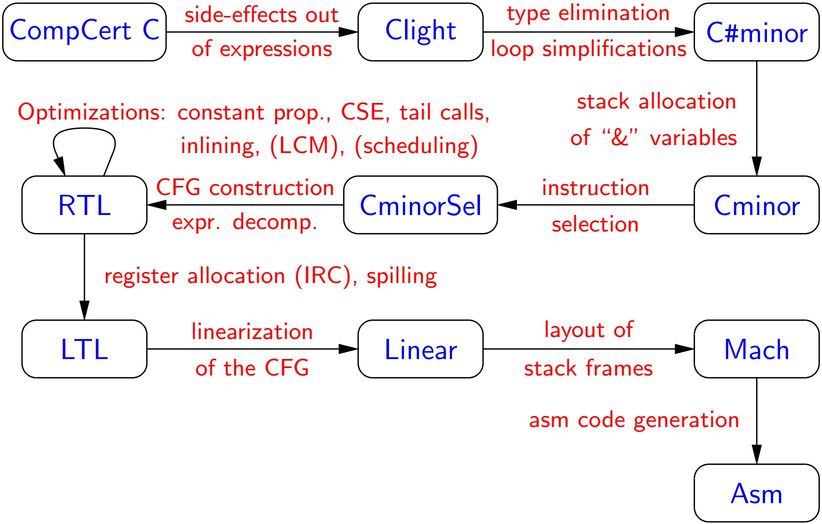
One major industrial achievement: the Common Criteria EAL7 certification of a Java Card system at Gemalto (B. Chetali et al).

Verification of the ARM6 micro-architecture
(A. Fox et al, Cambridge U.)



The CompCert verified C compiler

(X. Leroy et al, INRIA)



Some challenges and active research topics

Combining static analysis with program proof

- Static analysis as automatic generators of logical assertions.
- Static analysis as decision procedures for program proof.

Proof-preserving compilation

$$\begin{array}{c}\text{source program} + \text{logical annotation} + \text{proof in Hoare logic}\\ \downarrow \\ \text{machine code} + \text{logical annotation} + \text{proof in Hoare logic}\end{array}$$

Handling of bound variables and α-conversion

Perhaps the biggest obstacle to mechanizing high-level languages.
Cf. the "POPLmark challenge" at U. Penn.

Some challenges and active research topics

Semantics and logics of shared-memory concurrency

- Mechanizing program logics appropriate to reason on concurrent programs (rely-guarantee, concurrent separation logic, ...)
- Formalizing the "weakly consistent" memory models of today's multicore processors.
- Compiler optimizations in the presence of concurrency.

Verified development environments for critical software

Beyond compiler verification: formal assurance in code generators, program verification tools, the software-hardware interface, etc.