CompCert, a Coq-verified C compiler

Sandrine Blazy

IRISA - INRIA and Université Rennes 1

2nd Asian-Pacific summer school on formal methods, 2010-08-27

joint work with Xavier Leroy

Formal semantics of programming languages

Provide a mathematically-precise answer to the question

What does this program do, exactly?

What does this program do, exactly?

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l                ];D[l
++]-=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.)

Why indulge in formal semantics?

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

Is this program transformation correct?

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

Compiled for the Alpha processor with all optimizations and manually decompiled back to C...

```
double dotproduct(int n, double * a, double * b)
{
     double dp, a0, a1, a2, a3, b0, b1, b2, b3;
     double s0, s1, s2, s3, t0, t1, t2, t3;
     int i, k;
     dp = 0.0;
     if (n <= 0) goto L5;
     s0 = s1 = s2 = s3 = 0.0;
     i = 0; k = n - 3;
     if (k <= 0 || k > n) goto L19;
     i = 4; if (k <= i) goto L14;
     a0 = a[0];  b0 = b[0];  a1 = a[1];  b1 = b[1];
     i = 8; if (k <= i) goto L16;
L17: a2 = a[2];  b2 = b[2];  t0 = a0 * b0;
     a3 = a[3];  b3 = b[3];  t1 = a1 * b1;
     a0 = a[4];  b0 = b[4];  t2 = a2 * b2;  t3 = a3 * b3;
     a1 = a[5];  b1 = b[5];
     s0 += t0; s1 += t1;  s2 += t2;  s3 += t3;
     a += 4; i += 4; b += 4;
     prefetch(a + 20); prefetch(b + 20);
     if (i < k) goto L17;
L16: s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1];
L18: s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     dp = s0 + s1 + s2 + s3;
     if (i >= n) goto L5;
L19: dp += a[0] * b[0];
     i += 1; a += 1; b += 1;
     if (i < n) goto L19;
L5:  return dp;
L14: a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1]; goto L18;
}
```

Proof assistants

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

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.
- Prove the correctness of representative program transformations that can be found in compilers.
- Introduce the CompCert compiler.

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

Contents

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

1. Operational semantics.
2. Compilation to virtual machine code and its correctness proof.

3. The last part will be devoted to the CompCert compiler.

Lecture material :

- A Coq development (files `Sequences.v` and `SemCompil.v`)
- These slides

Part I

Operational semantics

Operational semantics

1. Warm-up: expressions and their denotational semantics

2. The IMP language and its reduction semantics

Warm-up: symbolic expressions

A language of expressions comprising

- variables x, y, ...
- 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 (see the lecture on inductive data types).

```
Definition ident := nat.

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

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

All terms of type `expr` are finitely generated by these 4 functions
$\rightarrow$ 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:

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

Mechanizing the denotational semantics

In Coq, the denotational semantics is presented as a recursive function ($\approx$ a definitional interpreter).

```
Definition state := ident -> Z.

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

Using the denotational semantics (1/3)

As an interpreter, to evaluate expressions.

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

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

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

Coq prints `= 13 : Z`.

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

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

```
Extraction eval_expr.
```

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

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

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

```
Recursive Extraction eval_expr.
```

```
...
type expr = Evar of ident | Econst of z
          | Eadd of expr * expr | Esub of expr * expr
...
let zplus x y = ...
...

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

Using the denotational semantics (2/3)

To reason symbolically over expressions.

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

Using the denotational semantics (3/3)

To prove "meta" properties of the semantics. For example: the denotation of an expression is insensitive to values of variables not mentioned in the expression.

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

where the predicate `occurs_in` is defined by

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

Variant 1: interpreting arithmetic differently

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

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

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

```
Definition normalize (x : Z) : Z :=
  let y := x mod 4294967296 in
  if Z_lt_dec y 2147483648 then y else y - 4294967296.
```

Variant 2: accounting for undefined expressions

In some languages, the value of an expression can be undefined:

- if it mentions an undefined variable;
- in case of arithmetic operation overflows (ANSI C);
- in case of division by zero;
- etc.

Recommended approach: use option types, with `None` meaning "undefined" and `Some` *n* meaning "defined and having value *n*".

```
Inductive option (A: Type): A -> option A :=
  | None: option A
  | Some: A -> option A.
```

Variant 2: accounting for undefined expressions

```
Definition ostate := ident -> option Z.

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

Summary

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

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

$\rightarrow$ 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 semantics

1. Warm-up: expressions and their denotational semantics

2. The IMP language and its reduction semantics

The IMP language

A prototypical imperative language with structured control.

Expressions:

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

Boolean expressions (conditions):

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

Commands (statements):

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

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

$$M \xrightarrow{\beta} M_1 \xrightarrow{\beta} M_2 \xrightarrow{\beta} \ldots$$

Each reduction $M \longrightarrow M'$ represents an elementary computation.
M' represents the residual computations that remain to be done later.

Reduction semantics for IMP

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

Reduction rule for assignments:

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

Reduction semantics for IMP

Reduction rules for sequences:

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

Example

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

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

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

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

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

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

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

Expressing inference rules in Coq

Step 1: write each rule as a proper logical formula

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

```
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 $R : A \to A \to \texttt{Prop}$:

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

Part II

Compilation to a virtual machine

Execution models for a programming language

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

Execution models for a programming language

1. Interpretation:
   the program is represented by its abstract syntax tree. The interpreter traverses this tree during execution.
2. Compilation to native code:
   before execution, the program is translated to a sequence of machine instructions. These instructions are those of a real microprocessor and are executed in hardware.

Execution models for a programming language

1. Interpretation:
   the program is represented by its abstract syntax tree. The interpreter traverses this tree during execution.
2. Compilation to native code:
   before execution, the program is translated to a sequence of machine instructions. These instructions are those of a real microprocessor and are executed in hardware.
3. Compilation to virtual machine code:
   before execution, the program is translated to a sequence of instructions. These instructions are those of a virtual machine. They do not correspond to that of an existing hardware processor, but are chosen close to the basic operations of the source language. Then,
   1. either the virtual machine instructions are interpreted (efficiently)
   2. or they are further translated to machine code (JIT).

Compilation to a virtual machine

3. The IMP virtual machine
4. Compiling IMP programs to virtual machine code
5. Notions of semantic preservation
6. Semantic preservation for our compiler

The IMP virtual machine

Components of the machine:

- The code C: a list of instructions.
- The program counter pc: an integer, giving the position of the currently-executing instruction in C.
- The 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

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

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	$\begin{matrix}1\\12\end{matrix}$	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	`var(`x`);`	`const(1);`	`add;`	`setvar(`x`);`	`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).

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

Compilation scheme for expressions

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

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

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

(= translation to "reverse Polish notation".)

Compilation scheme for conditions

The code $\texttt{comp\_b}(b, \delta)$ 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.

$$\begin{aligned}
\texttt{comp_b}(e_1 = e_2,\ \delta) &= \texttt{comp_e}(e_1); \texttt{comp_e}(e_2); \texttt{bne}(\delta) \\
\texttt{comp_b}(e_1 < e_2,\ \delta) &= \texttt{comp_e}(e_1); \texttt{comp_e}(e_2); \texttt{bge}(\delta)
\end{aligned}$$

Example

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

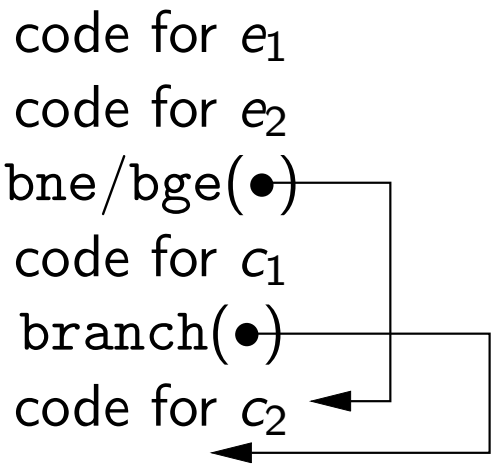
$\texttt{var}(x); \texttt{const}(1); \texttt{add};$	(compute $x + 1$)
$\texttt{var}(y); \texttt{const}(2); \texttt{sub};$	(compute $y - 2$)
$\texttt{bge}(\delta)$	(branch if $\geq$)

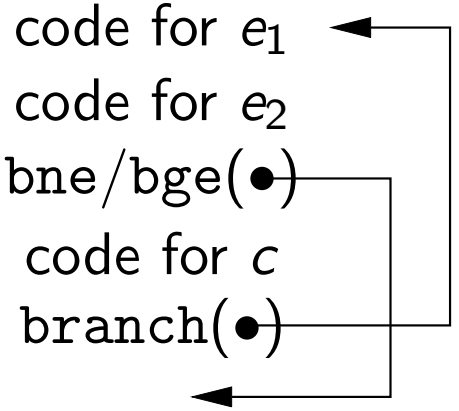
Compilation scheme for commands

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

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

Compilation scheme for commands





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

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

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

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

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

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

Compiler verification

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

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

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

Observable behaviors

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

Example

```
x := 1; x := 2;      ≈      x:=2;
print(1); print(2);  ≉      print(2);
```

Bisimulation (equivalence)

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

The source and transformed programs are completely undistinguishable.

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

Restrict ourselves to source programs that cannot go wrong:

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

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

`x := 1 / y; x := 42`	optimized to	`x := 42`
(goes wrong if $y = 0$)		(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. $\mathcal{B}(P_2) \subseteq Spec$).

The pains with backward simulation

Backward simulation for correct programs looks like the semantic preservation property we expect from a correct compiler.
It is however rather difficult to prove:

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

Forward simulations

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

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

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

P_2 has all the good behaviors of P_1, but could have additional bad behaviors.

Determinism to the rescue

Lemma

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

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

Utterly trivial result: follows from $\emptyset \subset X \subseteq \{y\} \Rightarrow X = \{y\}$.

Our plan for verifying a compiler

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

Note: (1) + (2) imply that the source langage has deterministic semantics. If this isn't naturally the case (e.g. for C), start by determinizing its semantics (e.g. fix an evaluation order a priori).

Compilation to a virtual machine

Compilation of expressions

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

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

More formally: $\texttt{comp\_e}(e) : \ (0, \sigma, s) \xrightarrow{*} (|\texttt{comp\_e}(e)|, (\llbracket e \rrbracket\ s).\sigma, s)$.

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

Compilation of expressions

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

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

Historical remark: the first published proof of correctness for a compiler (McCarthy & Painter, 1967).

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),

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

Combining with an add transition, we obtain:

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

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

Compilation of conditions

The code comp_b(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 ++ comp_b e ofs ++ C2))
       (pc, stk, s)
       (pc + length (comp_b 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.

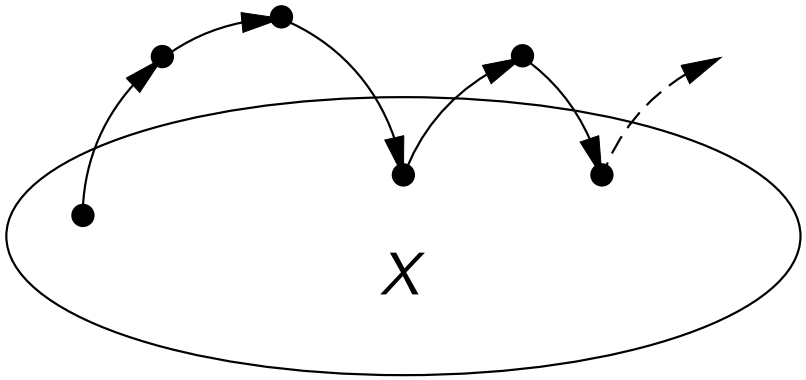
```
Lemma compile_cmd_correct_terminating:
  forall c s s',
  terminates c s s' ->
  forall stk pc C1 C2,
  pc = length C1 ->
  star (transition (C1 ++ comp c ++ C2))
       (pc, stk, s)
       (pc + length (comp c), stk, s').
```

(By induction on a derivation tree representing the execution of c and following the structure of commands).

Compilation of commands, diverging case

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

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 c s, diverges c s ->
  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.

Further reading

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

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

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

Part III

The CompCert verified C compiler

The CompCert project

compcert.inria.fr, conducted by Xavier Leroy

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

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

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

Used Coq to mechanize the proof of semantic preservation and also to implement most of the compiler.

The subset of C supported

Supported:

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

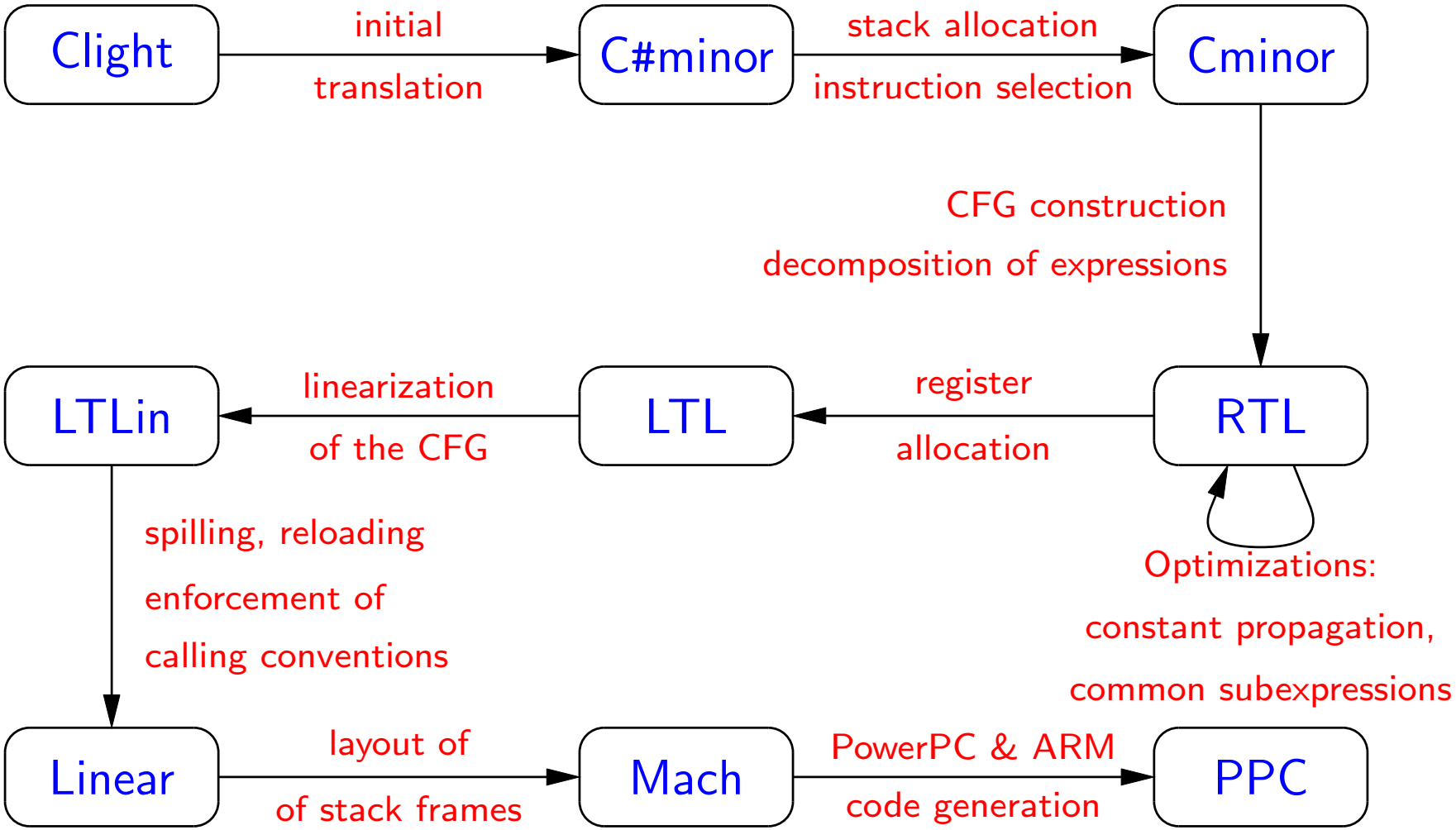
Not supported at all:

- The `long long` and `long double` types.
- `longjmp`/`setjmp`.

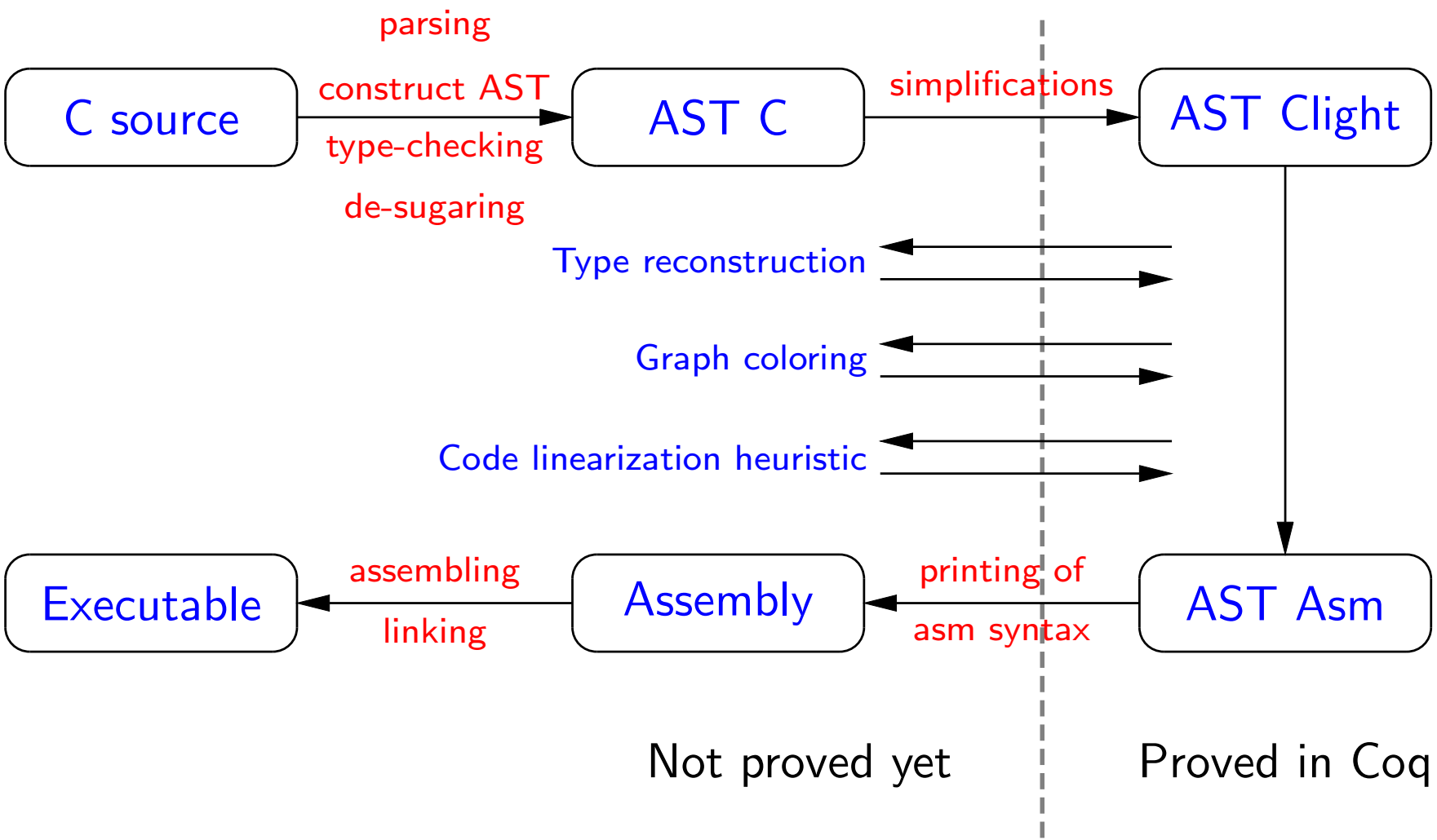
Supported through de-sugaring after parsing (not proved !):

- Side-effects within expressions.
- Block-scoped variables.
- Variable-arity functions.
- Assignment & pass-by-value of `struct` and `union`.
- Bit-fields.

The formally verified part of the compiler



The whole CompCert compiler



Refining control

From C-style structured loops (Clight) . . .

```
double average(int * tbl, int size)
{
double s; int i;
s = 0;
for (i = 0; i < size; i++)
    s += tbl[i];
return s / size;
}
```

Refining control

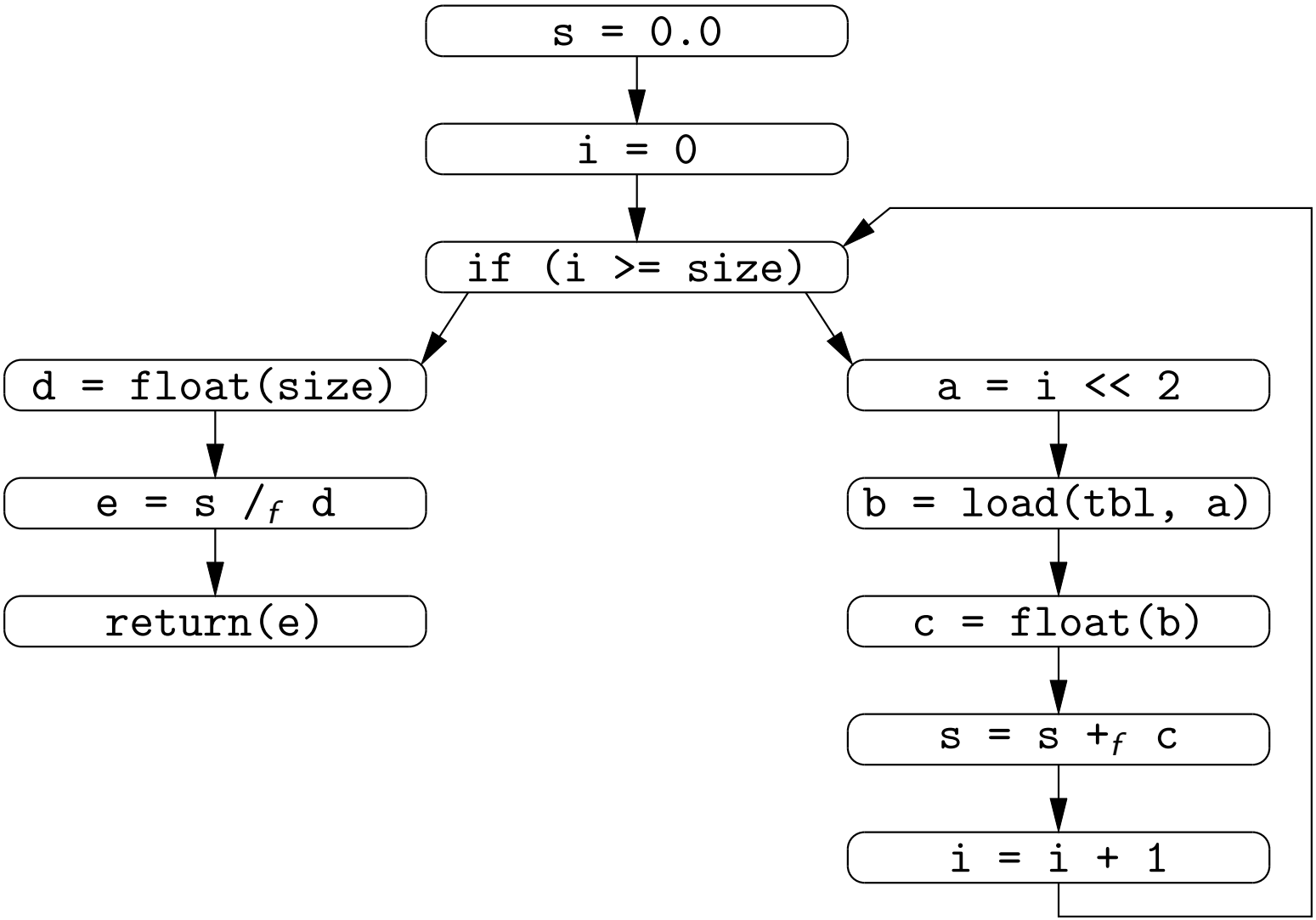
. . . to infinite loops + blocks + block exits (Cminor) . . .

```
average(tbl, size)
{
  var s, i;
  s = 0.0;
  i = 0;
  block {
      loop {
         if (i >= size) exit;
         s = s +f floatofint(load(int32, tbl + i << 2));
         i = i + 1;
      }
  }
  return s /f floatofint(size);
}
```

Also: materialization of all type-dependent behaviors (overloaded operators, implicit casts, address computations).

Refining control

. . . to a control-flow graph (RTL) . . .



Refining control

. . . to a list of instructions with labels, jumps, and conditional jumps. (LTLin and below.)

```
      F1 = 0.0
      R5 = 0
lbl1: if (R5 >= R4) goto lbl2
      R6 = R5 << 2
      R6 = load_int32(R3, R6)
      F2 = floatofint(R6)
      F1 = F1 + F2
      R5 = R5 + 1
      goto lbl1
      lbl2: F2 = floatofint(R4)
      F1 = F1 / F2
      return F1
```

Optimizations

Instruction selection:

```
x = (y & 0xFF00) >> 8; x = rlwinm(y, 24, 0xFF);
z = int32[tbl + (x + 1) * 4]; z = int32[(tbl + 4) + x << 2]
```

Recognition of processor-specific combined operations and addressing modes.
(By bottom-up rewriting of expressions.)

Optimizations

Tail call optimization:

```
x = f(...);
y = x;
return y;
```

```
tailcall f(...);
```

Tail-calls reduce stack consumption (stack frame is freed before the call, not after).

Optimizations

Constant propagation:

```
x = 0;
...
y = x + 1;
z = z + x;
if (y > 2)
    goto L1;
else
    goto L2;
```

```
x = 0;
...
y = 1;
z = z + 1;
goto L2;
```

Builds on the results of a forward data flow analysis.

Optimizations

Common subexpression elimination:

```
t1 = x;                          t1 = x;
t2 = t1 << 2;                    t2 = t1 << 2;
t3 = ptr + t2;                   t3 = ptr + t2;
y = load(t3);                    y = load(t3);
... ...
t4 = x;                          t4 = x; // eliminated later
t5 = t4 << 2;                    t5 = t2; // eliminated later
t6 = ptr + t5;                   t6 = t3; // eliminated later
z = load(t6);                    z = y;
```

Optimizations

As part of register allocation:

- Dead code elimination 1
  (instructions whose result is unused $\longrightarrow$ `nop`)
- Coalescing of redundant moves
  (`x := x` $\longrightarrow$ `nop`)

As part of linearization:

- Dead code elimination 2
  (`nop` and unreachable instructions are deleted)

Verified in Coq

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

A composition of 14 forward simulation proofs.

Observable behaviors

The observable behavior of a program P is:

- $P \Downarrow \texttt{terminates}(t, n)$:
  termination, with trace t of input-output events, and return code n (return value of the `main` function).
- $P \Uparrow \texttt{diverges}(T)$:
  divergence, with trace T of input-output events (finite or infinite).
- $P \Downarrow \texttt{wrong}(t)$: going wrong (undefined behavior).

Some functions of the program are declared as "system calls".
Every call to such a function produces an event in the trace:
`name(arguments, result)`.

All semantics are internally deterministic: the only source of non-determinism is the results of system calls, provided by the outside world.

The Coq proof

4 person-years of work + many experiments

Size of proof : 50 000 lines of Coq

Size of program proved : 8 000 lines

Code : 13% of the proof
Semantics : 8%
Proof scripts : 55%
Misc. : 7%

Programmed in Coq

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

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

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

Claim: pure functional programming is the shortest path between an executable program and its proof.

Example of error handling

```
Inductive res (A: Type) : Type :=
| OK: A -> res A
| Error: errmsg -> res A.

Fixpoint transl_expr (a: Csyntax.expr) struct a : res expr :=
  match a with
  | Expr (Csyntax.Econst_int n) _ => OK(make_intconst n)
  | Expr (Csyntax.Evar id) ty => var_get id ty
  | Expr (Csyntax.Eunop op b) _ =>
      do tb <- transl_expr b;
      transl_unop op tb (typeof b)
  | Expr (Csyntax.Ebinop op b c) _ =>
      do tb <- transl_expr b;
      do tc <- transl_expr c;
      transl_binop op tb (typeof b) tc (typeof c)
  ...
  end
```

Monadic inversion

```
Lemma transl_Eunop_correct : ...
Proof.
intros.  ...
```

Current subgoal

```
...
H : Csem.eval_expr ge e m a v1
H1 : sem_unary_operation op v1 (typeof a) = Some v
TR : transl_expr (Expr (Csyntax.Eunop op a) ty) = OK ta
------------------------------------
eval_expr tgve te m ta v
```

Monadic inversion : `monadInv TR.`

```
...
EQ : transl_expr a =  OK x
EQ0 : transl_unop op x (typeof a) =  OK ta
------------------------------------
eval_expr tgve te m ta v
```

Coq's extraction mechanism

```
Recursive Extraction Library Compiler.
```

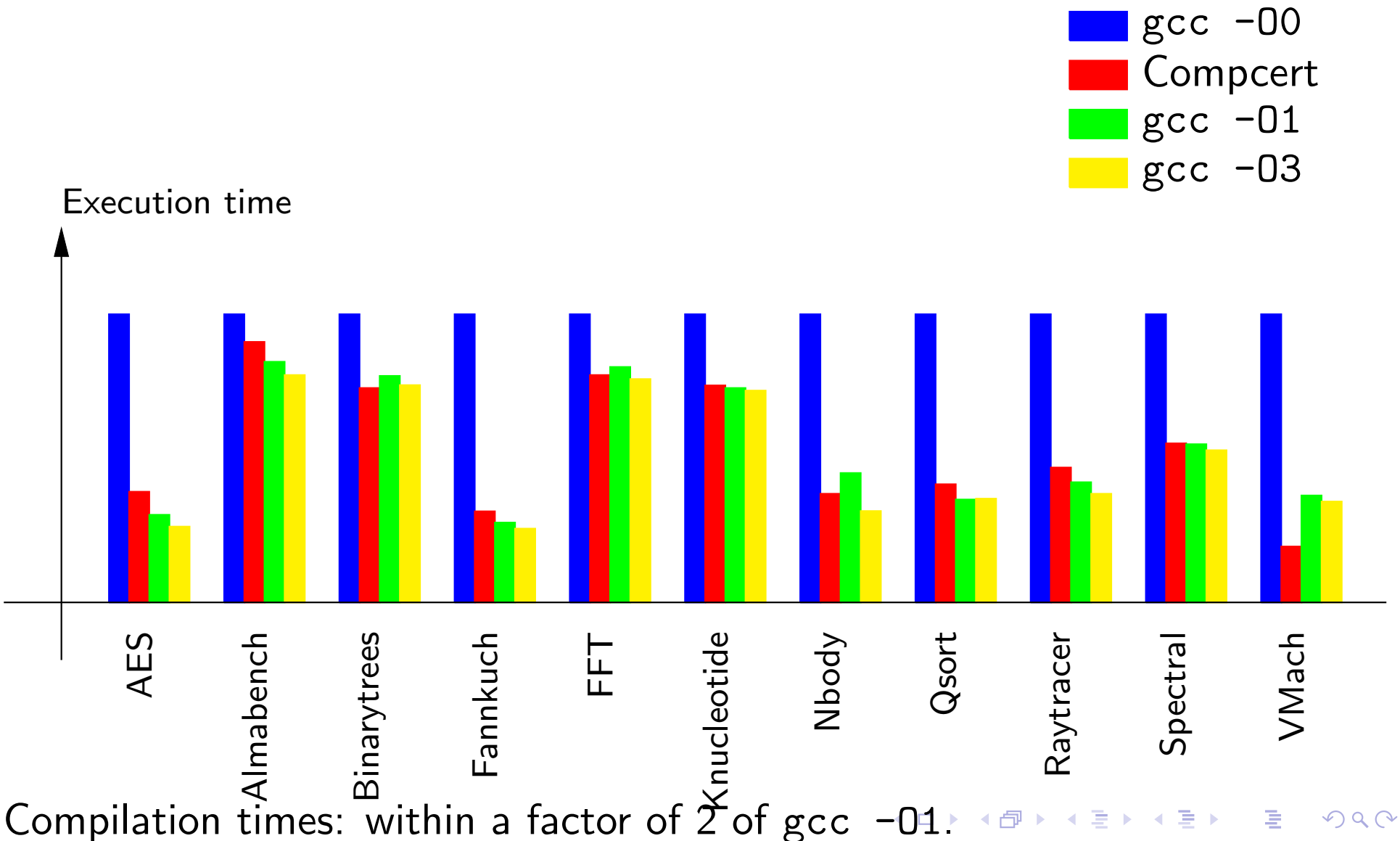
```
let rec transl_expr a = match a with
| Expr (e, ty) -> (match e with
    | Econst_int n -> OK (make_intconst n)
    | Evar id -> var_get id ty
    | Eunop (op, b) -> (match transl_expr b with
        | OK x -> transl_unop op x (typeof b)
        | Error msg0 -> Error msg0)
    | Ebinop (op, b, c) -> (match transl_expr b with
        | OK x -> (match transl_expr c with
            | OK x0 -> transl_binop op x (typeof b) x0 (typeof c)
            | Error msg0 -> Error msg0)
        | Error msg0 -> Error msg0)
    | _ -> Error (msg (...)))
```

but also

```
Extract Constant Color.graph_coloring => "Coloraux.graph_coloring".
```

Performances of the generated code

(On a PowerPC G5 processor)



Compilation times: within a factor of 2 of gcc -O1.

Preliminary conclusions

At this stage of the Compcert experiment, the initial goal - proving correct a realistic compiler - appears feasible.

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

What next?

Enhancements to CompCert

Upstream :

- Prove correct some of the C $\longrightarrow$ Clight simplifications.
- Is there anything to prove about the C parser? preprocessor??

Downstream :

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

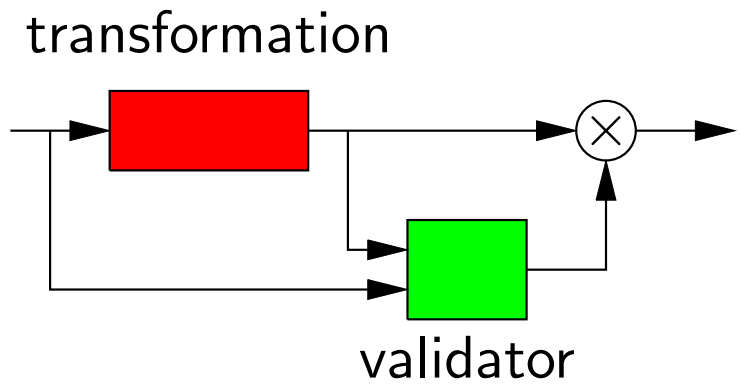
Enhancements to CompCert

In the middle :

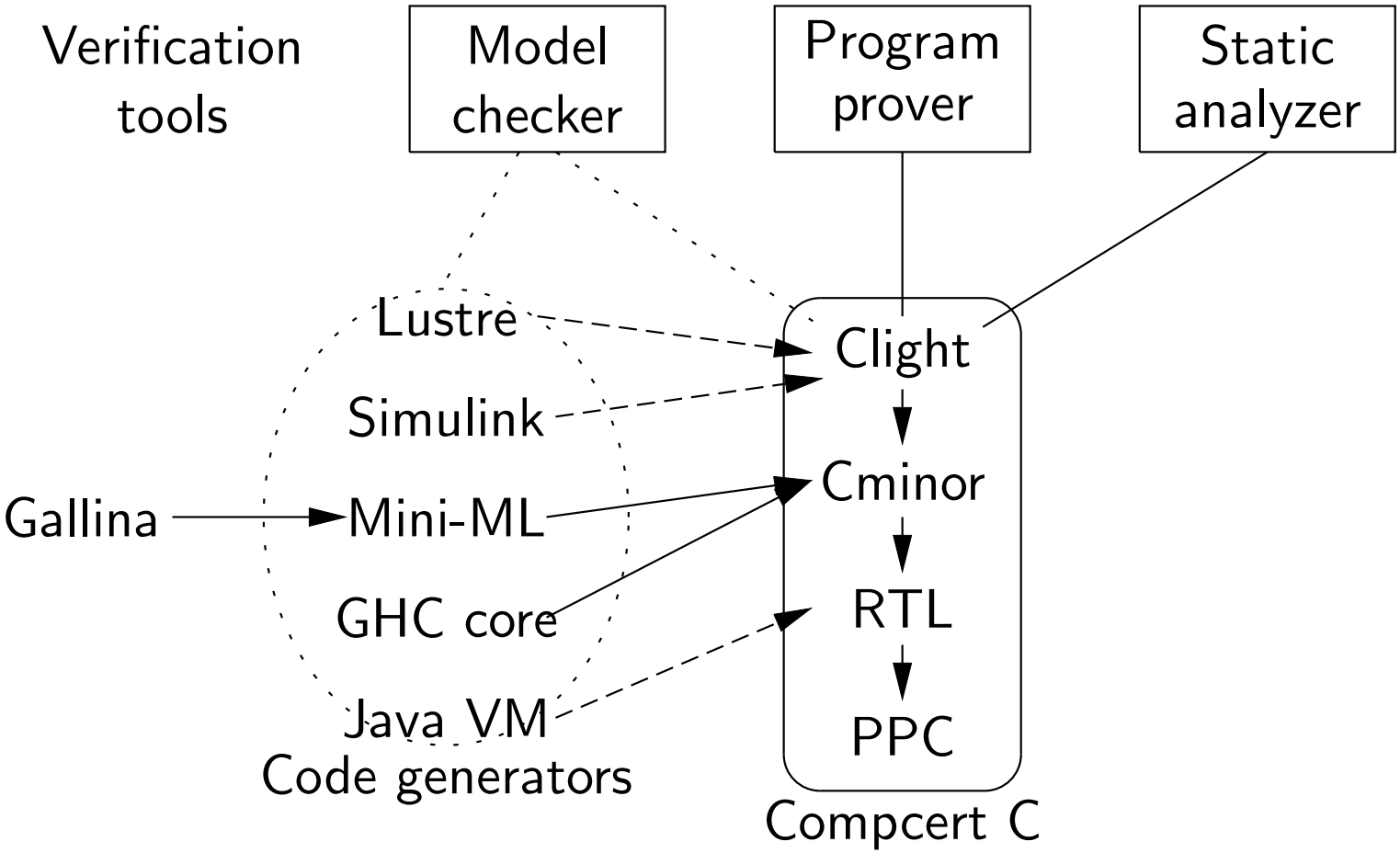
- More static analyses, esp. for nonaliasing.
- Formal verification of the Appel and George heuristic for register allocation.
- More optimizations? Possibly using verified translation validation (e.g. instruction scheduling, lazy code motion, and software pipelining) ?
- Other target architectures, e.g. x86.

Verified transformation

Verified translation validation



Extensions on the "input" side



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

Connections with verification tools

Consider other C-related tools involved in the production and verification of critical software: code generators, static analyzers, model checkers, program provers, ...

- (Long term) Formally verify these tools as well?
  (E.g. verification condition generators, abstract interpreters, abstract domains, etc)
- (Medium term) Validate the operational semantics used in CompCert against the other semantics used in these tools?
  (E.g. axiomatic semantics, collecting semantics, etc)
- (More modestly) Agree on a common subset of the C language?

Towards shared-memory concurrency

Concurrent Cminor project, Andrew W. Appel et al.

Programs containing data races are generally compiled in a non-semantic-preserving manner.

One solution : Restrict ourselves to data-race free source programs . . .
. . . as characterized by concurrent separation logic.

For parallel programs provable in concurrent separation logic, we can restrict ourselves to "quasi-sequential" executions

Verifying a compiler for data-race free programs

"Just" have to show that quasi-sequential executions are preserved by compilation:

- Easy ?? extensions of the sequential case.
- Can still use forward simulation arguments.
- Most classic sequential optimizations remain valid.
- The only "no-no": moving memory accesses across lock and unlock operations.

Work in progress, stay tuned . . .