Formal verification of a compiler front-end for mini-ML

Zaynah Dargaye, Xavier Leroy, Andrew Tolmach

INRIA Paris-Rocquencourt

WG 2.8, july 2007

Formal verification of compilers

Apply formal methods to a compiler. Prove a semantic preservation property:

Theorem

For all source codes S,
if the compiler generates machine code C from source S,
without reporting a compilation error,
and if S has well-defined semantics,
then C has well-defined semantics
and S and C have the same observable behaviour.

Formal verification of compilers

Motivations:

- Useful for high-assurance software, verified (at the source level) using formal methods.
- A challenge for mechanized program proof.
- For fun!
  (compilers + pure F.P. + mechanized proof,
  all in one easy-to-explain project).

The Compcert effort
INRIA/CNAM/Paris 7, since 2003

Develop and prove correct a realistic compiler, usable for critical embedded software.

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

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

We use the Coq proof assistant to conduct the proof of semantic preservation and to write most of the compiler.

The Compcert effort – status

A prototype compiler that executes (under MacOS X).

From Clight AST to PowerPC assembly AST:

- entirely verified in Coq (40000 lines);
- entirely programmed in Coq, then automatically extracted to executable Caml code.
  Uses monads, persistent data structures, etc.

Performances of generated code: better than `gcc -O0`, close to `gcc -O1`.

Compilation times: comparable to those of `gcc -O1`.

References: X. Leroy, POPL 2006 (back-end); S. Blazy, Z. Dargaye, X. Leroy, Formal Methods 2006 (C front-end).

Front-ends for other source languages

Clight ⟶ Cminor ⟶ PPC

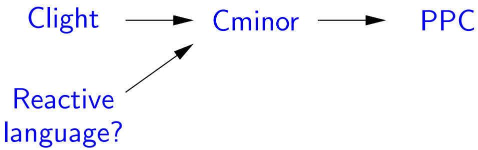
Cminor could be a reasonable I.L. for other source languages.

A flavor of Cminor

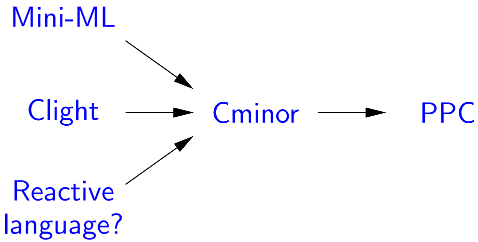
```
"quicksort"(lo, hi, a): int -> int -> int -> void
{
  var i, j, pivot, temp;
  if (! (lo < hi)) return;
  i = lo; j = hi; pivot = int32[a + hi * 4];
  block { loop {
    if (! (i < j)) exit;
    block { loop {
      if (i >= hi || int32[a + i * 4] > pivot) exit;
      i = i + 1;
    } }
    /* ... */
  } }
  temp = int32[a + i * 4];
  int32[a + i * 4] = int32[a + hi * 4];
  int32[a + hi * 4] = temp;
  "quicksort"(lo, i - 1, a) : int -> int -> int -> void;
  tailcall "quicksort"(i + 1, hi, a) : int -> int -> int -> void;
}
```

Front-ends for other source languages

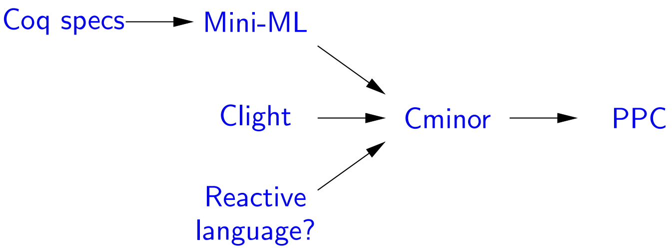
Front-ends for other source languages



Front-ends for other source languages



Front-ends for other source languages



Towards a trusted execution path for programs written and proved in Coq.

This includes the Compcert compiler itself . . . (bootstrap!)

mini-ML: syntax

Pure, call-by-value, datatypes + shallow pattern matching.

Terms:	$a ::= \underline{n}$	variable (de Bruijn)
	$\mid \lambda.a \mid a_1\ a_2$	
	$\mid \mu.\lambda.a$	recursive function
	$\mid \texttt{let}\ a_1\ \texttt{in}\ a_2$	
	$\mid C(a_1, \ldots, a_n)$	data constructor
	$\mid \texttt{match}\ a\ \texttt{with}\ p_1 \rightarrow a_1 \ldots p_n \rightarrow a_n$	
Patterns:	$p ::= C^n$	i.e. $C(\underline{n}, \ldots, \underline{1})$

Also: constants and arithmetic operators.

More or less the output language for Coq's extraction, minus mutually-recursive functions.

mini-ML: dynamic semantics

Big-step operational semantics with environments

$$e \vdash a \Rightarrow v$$

with $v ::= C(v_1, \ldots, v_n) \mid (\lambda.a)[e] \mid (\mu.\lambda.a)[e]$ and $e = v_1 \ldots v_n$.

Entirely standard.

Big-step semantics with substitutions also used in some of the proofs.

mini-ML: (no) type system

Our Mini-ML is untyped:

- Makes it easier to translate various typed F.P.L. to mini-ML, e.g. Coq with its extremely powerful type system.
- We are doing semantic-preserving compilation, which subsumes all the guarantees that type-preserving compilation provides.

Exception: we demand that constructors are grouped into "datatype declarations" to facilitate pattern-matching compilation (see example).

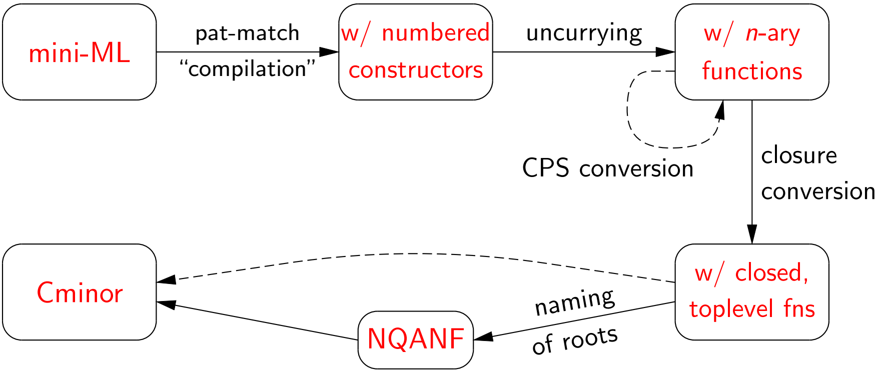
Example of mini-ML

```
type list = Nil | Cons

program

let map = μmap. λx.
    match x with
      | Nil -> Nil
      | Cons(hd, tl) -> Cons(f hd, map f tl)
    in
       map (λx. Cons(x, Nil)) Nil
```

Overview of the compiler



Uncurrying

`let`-bound curried functions are turned into *n*-ary functions.

```
let f = λx.λy. ... in Pair(f 1 2, f 1)
```

$$\Downarrow$$

```
let f = λ(x, y). ... in
Pair(f(1, 2), ((λx.λy.f(x,y))(1)))
```

Generation of Cminor code

Quite straightforward if Cminor had dynamic memory allocation with garbage collection.

(Mostly, represent constructor applications and closures as pointers to appropriately-filled memory blocks.)

But Cminor has no memory allocator, no GC, and no run-time system of any kind. . .

Run-time systems: the bane of high-level languages

Run-time systems are big (e.g. 50000 lines), messy, written in C, system-dependent, often buggy, . . .

Yet, the run-time system must be proved correct in the context of a verified compiler for a high-level language.

What needs to be done

For the memory allocator and (tracing) garbage collector:

- The algorithms must be proved correct.
  (Mostly routine.)
- The actual implementation (typically in Cminor) must be proved correct.
  (Painful, like all proofs of imperative programs.)
- This proof must be connected to that of the compiler:
  - Compiler-generated code must respect GC contract
    (Data representation conventions, don't touch block headers, etc)
  - GC must be able to find the memory roots
    (among the compiler-managed registers, call stack, etc)

What needs to be done

For the memory allocator and (tracing) garbage collector:

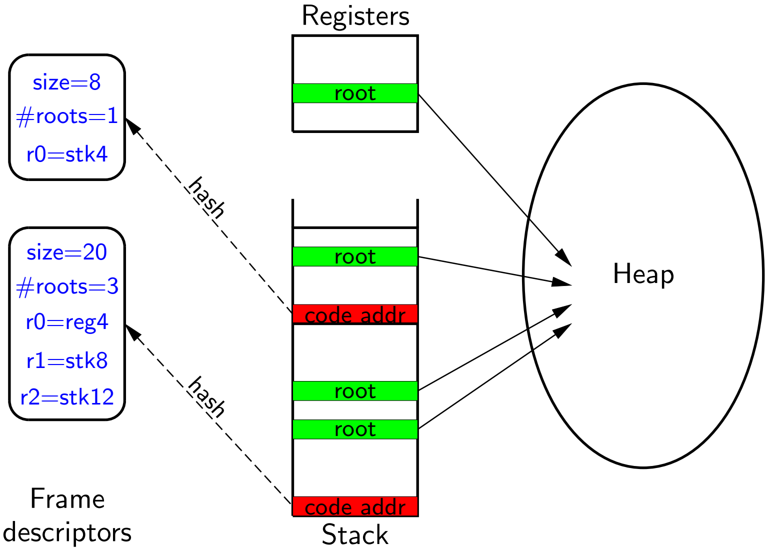
- The algorithms must be proved correct.
  (Mostly routine.)
- The actual implementation (typically in Cminor) must be proved correct.
  (Painful, like all proofs of imperative programs.)
- This proof must be connected to that of the compiler:
  - Compiler-generated code must respect GC contract
    (Data representation conventions, don't touch block headers, etc)
  - GC must be able to find the memory roots
    (among the compiler-managed registers, call stack, etc)

What needs to be done

For the memory allocator and (tracing) garbage collector:

- The algorithms must be proved correct.
  (Mostly routine.)
- The actual implementation (typically in Cminor) must be proved correct.
  (Painful, like all proofs of imperative programs.)
- This proof must be connected to that of the compiler:
  - Compiler-generated code must respect GC contract
    (Data representation conventions, don't touch block headers, etc)
  - GC must be able to find the memory roots
    (among the compiler-managed registers, call stack, etc)

Example: finding roots using frame descriptors



Possible approaches

- Plan A: prove the "frame descriptor" approach.
  Extensive work needed on the back-end: tracking of roots through compiler passes, proving preservation of the GC contract, etc
- Plan B: revert to "lesser" GC technology.
  Conservative tracing collection.
  Or even reference counting.
- Plan C: explicit root registration.
  Instrument generated Cminor code to keep track of memory roots and to communicate them to the allocator.
  A good match for a GC and allocator written in Cminor themselves.

Possible approaches

- Plan A: prove the "frame descriptor" approach.
  Extensive work needed on the back-end: tracking of roots through compiler passes, proving preservation of the GC contract, etc
- Plan B: revert to "lesser" GC technology.
  Conservative tracing collection.
  Or even reference counting.
- Plan C: explicit root registration.
  Instrument generated Cminor code to keep track of memory roots and to communicate them to the allocator.
  A good match for a GC and allocator written in Cminor themselves.

Possible approaches

- Plan A: prove the "frame descriptor" approach.
  Extensive work needed on the back-end: tracking of roots through compiler passes, proving preservation of the GC contract, etc
- Plan B: revert to "lesser" GC technology.
  Conservative tracing collection.
  Or even reference counting.
- Plan C: explicit root registration.
  Instrument generated Cminor code to keep track of memory roots and to communicate them to the allocator.
  A good match for a GC and allocator written in Cminor themselves.

An example of root registration

```
ptr f(ptr x) {
    ptr y, z, t;
    ...
    /* Assume x, y are roots (must survive next allocation) */
    { struct { int nroots; ptr roots[2]; } rb;
      rb.nroots = 2;
      rb.roots[0] = x;
      rb.roots[1] = y;
      t = alloc(&rb, size);
      x = rb.roots[0];
      y = rb.roots[0];
    }
    ...
}
```

Root passing style

If in direct style, need to chain root blocks for all active function invocations.

```
ptr f(rootblock * roots, ptr x) {
    ptr y, z, t;
    ...
    /* Assume x, y are roots (must survive next call) */
    { struct { rootblock * next; int nroots; ptr roots[2]; } rb;
      rb.next = roots;
      rb.nroots = 2;
      rb.roots[0] = x;
      rb.roots[1] = y;
      t = g(&rb, z);
      x = rb.roots[0];
      y = rb.roots[0];
    }
    ...
}
```

Generating Cminor code with explicit root registration

Easier done from an I.L. where evaluation order is explicit and potential roots are named (`let`-bound).

Inconvenient:

$$f(C(x), C(y), z)$$

More convenient:

$$\texttt{let}\ t_1 = C(x)\ \texttt{in}\ \texttt{let}\ t_2 = C(y)\ \texttt{in}\ f(t_1, t_2, z)$$

Candidate intermediate languages:

- CPS (plus: no need for root-passing style)
- ANF
- Not-Quite-ANF

Roots in CPS

CPS with `let`-binding of allocations (of closures or constructors):

Atoms: $a ::= x \mid \texttt{field}_n(a)$

Allocations: $c ::= \texttt{clos}(f, a_1, \ldots, a_n) \mid C(a_1, \ldots, a_n)$

Terms: $t ::= a \mid a(a_1, \ldots, a_n)$
$\mid \texttt{let}\ x = c\ \texttt{in}\ t$
$\mid \texttt{match}\ a\ \texttt{with}\ p_i \to t_i$

The roots for the allocation c in $\texttt{let}\ x = c\ \texttt{in}\ b$ are

$$FV(c) \cup (FV(b) \setminus \{x\}) = FV(\texttt{let}\ x = c\ \texttt{in}\ b)$$

Roots in ANF

Atoms: $a ::= x \mid \texttt{field}_n(a)$

Computations: $c ::= \texttt{clos}(f, a_1, \ldots, a_n) \mid C(a_1, \ldots, a_n) \mid a(a_1, \ldots, a_n)$

Terms:
$$
\begin{aligned}
t ::= &\ c \\
\mid &\ \texttt{let}\ x = c\ \texttt{in}\ t \\
\mid &\ \texttt{match}\ a\ \texttt{with}\ p_i \rightarrow t_i
\end{aligned}
$$

The roots for the allocation c in $\texttt{let}\ x = c\ \texttt{in}\ b$ are

$$R(c) \cup (FV(b) \setminus \{x\})$$

where

$$
\begin{aligned}
R(\texttt{clos}(f, a_1, \ldots, a_n)) &= FV(\texttt{clos}(f, a_1, \ldots, a_n)) \\
R(C(a_1, \ldots, a_n)) &= FV(C(a_1, \ldots, a_n)) \\
R(a(a_1, \ldots, a_n)) &= \emptyset
\end{aligned}
$$

Is ANF necessary?

Important feature: arguments to calls and allocations are atoms, i.e. computations that never trigger a GC.

Disadvantage: prohibits left-nested `let` and `match`

$$\texttt{let}\ x = (\texttt{let}\ y = a\ \texttt{in}\ b)\ \texttt{in}\ c$$

$$\texttt{match}\ (\texttt{match}\ a\ \texttt{with}\ \ldots)\ \texttt{with}\ \ldots$$

Requires `match`-of-`match` normalization, which can duplicate code.

Conjecture: can track roots just as easily over "Not-Quite ANF", i.e. ANF where left-nested `let` and `match` are allowed.

Status

A Caml prototype of the mini-ML $\rightarrow$ Cminor chain
$+$ two GC in Cminor (mark-and-sweep, stop-and-copy).
Performances: 3 $\times$ slower than native OCaml, 3 $\times$ faster than bytecode OCaml.

Coq formalizations and proofs of mini-ML $\rightarrow$ NQANF.

In progress:

- Coq mechanization of NQANF $\rightarrow$ Cminor.
- Coq proof of the GC.

Mostly open: connecting the two proofs . . .