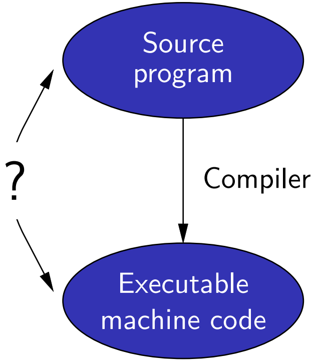
A formally verified compiler for critical embedded software

Xavier Leroy

INRIA Paris-Rocquencourt

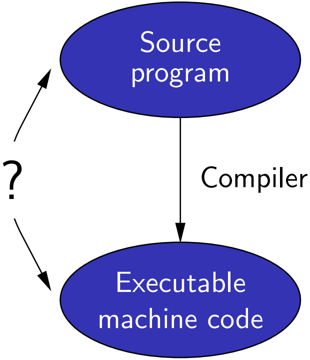
LCTES, 2008-06-12

Can you trust your compiler?



Bugs in the compiler can lead to incorrect machine code being generated from a correct source program.

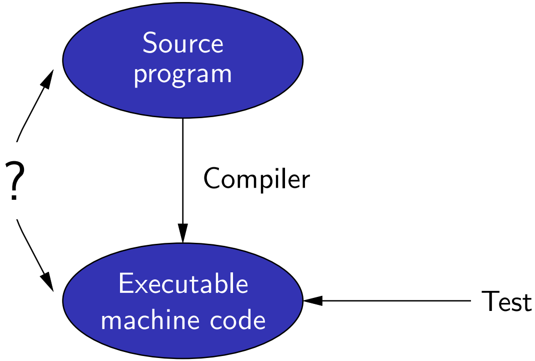
Can you trust your compiler?



Non-critical sofware:

Compiler bugs are negligible compared with those of the program itself.

Can you trust your compiler?

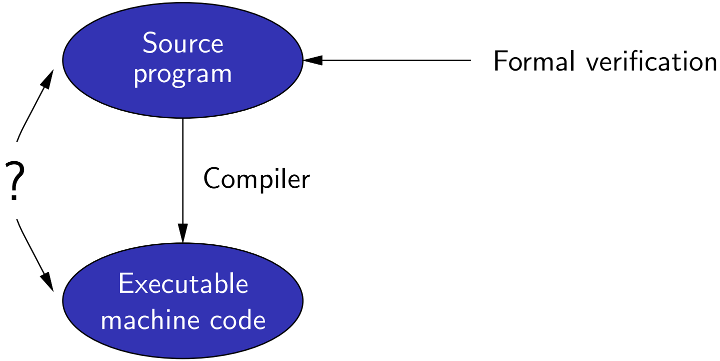


Critical software certified by systematic testing:

What is tested: the executable code generated by the compiler.

Compiler bugs are detected along with those of the program.

Can you trust your compiler?

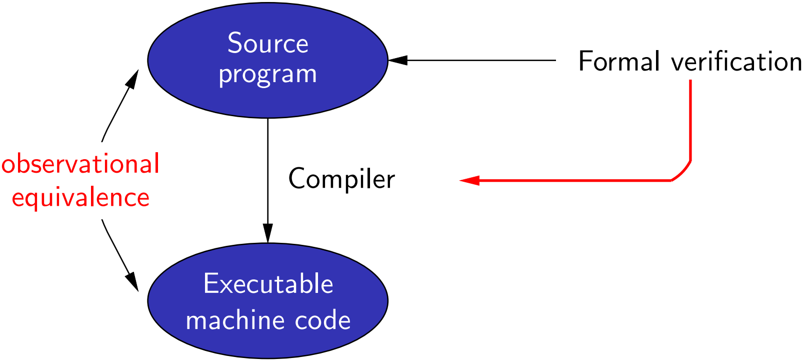


Critical software certified by formal methods::

What is formally verified: the source code, not the executable code.

Compiler bugs can invalidate the guarantees obtained by formal methods.

Can you trust your compiler?

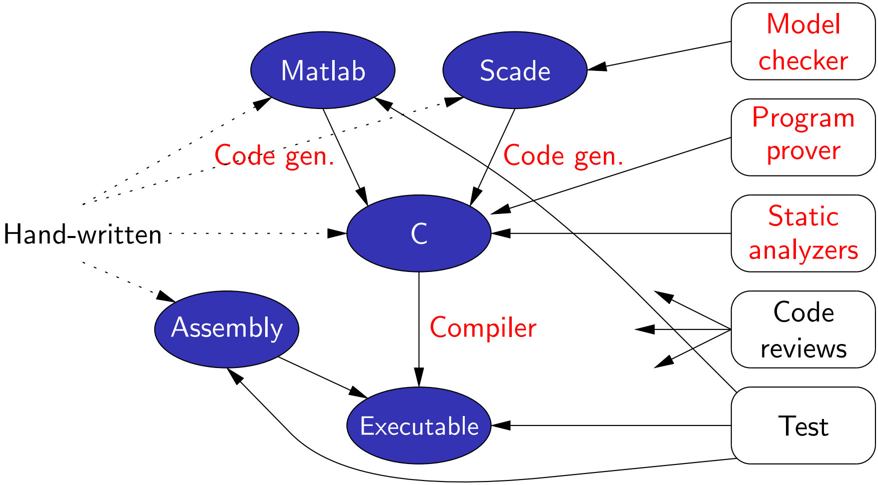


Formally verified compiler:

Guarantees that the generated executable code behaves as prescribed by the semantics of the source program.

In reality. . .

A great many tools are involved in the production and verification of critical software:



Outline

1. Introduction: Can you trust your compiler?
2. Formally verified compilers
3. The Compcert experiment
4. An example of verification: the register allocation pass
5. An example of verified translation validation: lazy code motion
6. Perspectives

Formal verification of compilers

Apply formal methods to the compiler itself to 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,
then C behaves like S.

Note: compilers are allowed to fail (ill-formed source code, or capacity exceeded).

Some preservation properties of interest

Preservation of all behaviors

The observable behaviors of the source and compiled programs are identical:

$$\forall b, \quad S \Downarrow b \iff C \Downarrow b$$

(Notation: $p \Downarrow b$ means "p does not go wrong and executes with observable behavior b".)

Some preservation properties of interest

Preservation of “not going wrong” behaviors

Compilers are allowed to refine behaviors if the source language is not deterministic.

Compilers are allowed to generate code that doesn't go wrong for a source program that goes wrong.

$$\begin{array}{ll} & \exists b,\ S \Downarrow b \\ \Longrightarrow & \\ & (\exists b,\ C \Downarrow b)\ \wedge (\forall b',\ C \Downarrow b' \implies S \Downarrow b') \end{array}$$

If the target language is deterministic, this is implied by

$$\forall b,\quad S \Downarrow b \implies C \Downarrow b$$

Some preservation properties of interest

Preservation of specifications

Let *Spec* : *behavior* → *Prop* be a functional specification for the program. If the source satisfies *Spec*, so does the compiled code.

$$
\begin{array}{ll}
 & (\exists b,\ S \Downarrow b \ \wedge\ \forall b,\ S \Downarrow b \Longrightarrow \mathit{Spec}(b)) \\
\Longrightarrow & \\
 & (\exists b,\ C \Downarrow b \ \wedge\ \forall b,\ C \Downarrow b \Longrightarrow \mathit{Spec}(b))
\end{array}
$$

Implied by the previous property (preservation of "not going wrong" behaviors).

Special case: preservation of type and memory safety.

Approach 1: proving the compiler

Model the compiler as a function

$$Comp : Source \rightarrow Code + \texttt{Error}$$

and prove that

$$\forall S, C, b, \quad Comp(S) = C \implies S \equiv C \text{ (observational equivalence)}$$

using a proof assistant.

Note: complex data structures + recursive algorithms ⇒ interactive program proof is a necessity.

Approach 2: translation validation

(A. Pnueli et al; G. Necula; X. Rival)

Validate a posteriori the results of compilation:

$$\begin{aligned} \mathit{Comp} &: \mathit{Source} \rightarrow \mathit{Code} + \texttt{Error} \\ \mathit{Validator} &: \mathit{Source} \times \mathit{Code} \rightarrow \texttt{bool} \end{aligned}$$

If $\mathit{Comp}(S) = C$ and $\mathit{Validator}(S, C) = \texttt{true}$, success.
Otherwise, error.

It suffices to prove that the validator is correct:

$$\forall S, C, \quad \mathit{Validator}(S, C) = \texttt{true} \implies S \equiv C$$

The compiler itself needs not be proved.

Approach 3: proof-carrying code
(G. Necula and P. Lee)

$$\begin{aligned} \mathit{Comp} &: \mathit{Source} \times \mathit{Prop} \times \mathit{Proof} \rightarrow \mathit{Code} \times \mathit{Certificate} + \mathtt{Error} \\ \mathit{Checker} &: \mathit{Prop} \times \mathit{Code} \times \mathit{Certificate} \rightarrow \mathtt{bool} \end{aligned}$$

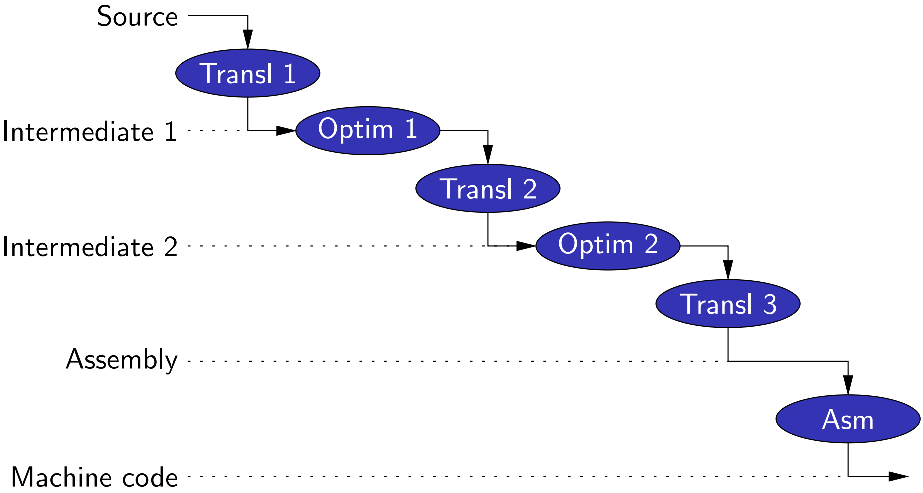
If $\mathit{Comp}(S, P) = (C, \pi)$ and $\mathit{Validator}(P, C, \pi) = \mathtt{true}$, success. Otherwise, error.

Assume that the checker is proved correct:

$$\forall P, C, \pi, \quad \mathit{Checker}(P, C, \pi) = \mathtt{true} \Rightarrow C \models P$$

Enables the code consumer to check the validity of the compiled code without trusting the code producer and without having access to the source code. (Think mobile code.)

Decomposition in multiple compiler passes



Decomposition in multiple compiler passes

If every compiler pass preserves semantics, so does their composition!

A compiler pass can generally be proved correct independently of other passes.

However, formal semantics must be given to every intermediate language (not just source and target languages).

For each pass, we can either

- prove it correct directly, or
- use validation a posteriori and just prove the correctness of the validator.

Outline

1. Introduction: Can you trust your compiler?
2. Formally verified compilers
3. The Compcert experiment
4. An example of verification: the register allocation pass
5. An example of verified translation validation: lazy code motion
6. Perspectives

The Compcert experiment

(X.Leroy, Y.Bertot, S.Blazy, Z.Dargaye, P.Letouzey, T.Moniot, L.Rideau, B.Serpette)

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
  ⇒ some optimizations.

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

The proof of semantic preservation is mechanized using the Coq proof assistant.

The subset of C supported

Supported:

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

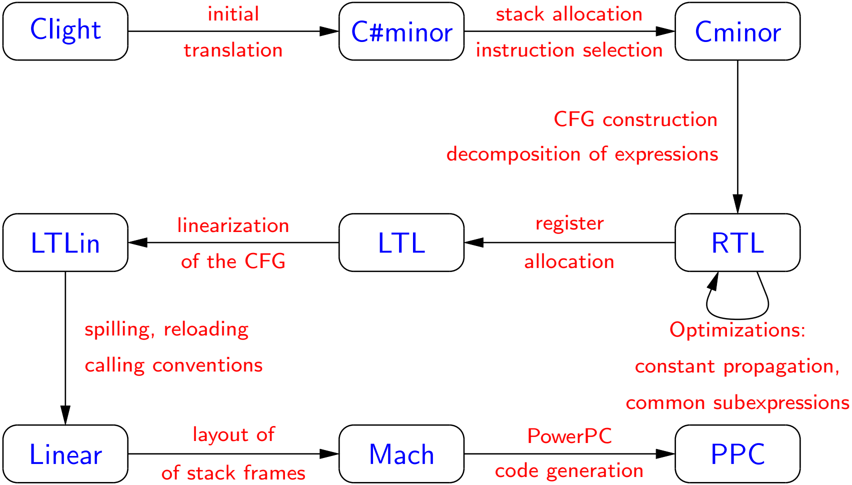
Not supported at all:

- The `long long` and `long double` types.
- `goto`, unstructured `switch`, `longjmp`/`setjmp`.
- Variable-arity functions.
- Passing `struct` and `union` by value.

Supported through de-sugaring after parsing:

- Side-effects within expressions.
- Block-scoped variables.

The formally verified part of the compiler



Clight

initial translation

C#minor

stack allocation instruction selection

Cminor

CFG construction decomposition of expressions

RTL

Optimizations: constant propagation, common subexpressions

register allocation

LTL

linearization of the CFG

LTLin

spilling, reloading calling conventions

Linear

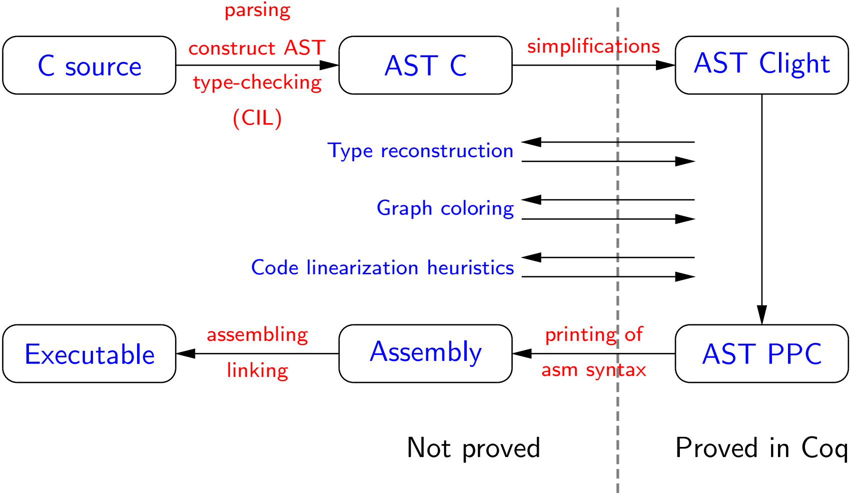
layout of of stack frames

Mach

PowerPC code generation

PPC

The whole Compcert compiler



Formally verified using Coq

The correctness proof (semantic preservation) for the compiler is entirely machine-checked, using the Coq proof assistant.
(48000 lines of Coq, 2.5 man.years.)

```
Theorem transf_c_program_correct:
    forall prog tprog behavior,
    transf_c_program prog = OK tprog ->
    Clight.exec_program prog behavior ->
    PPC.exec_program tprog behavior.
```

Observable behaviors are either

- Termination, with a finite trace of input-output events (system calls) and the integer returned by the `main` function (exit code).
- Divergence, with a finite or infinite trace of input-output events.

Programmed in Coq

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

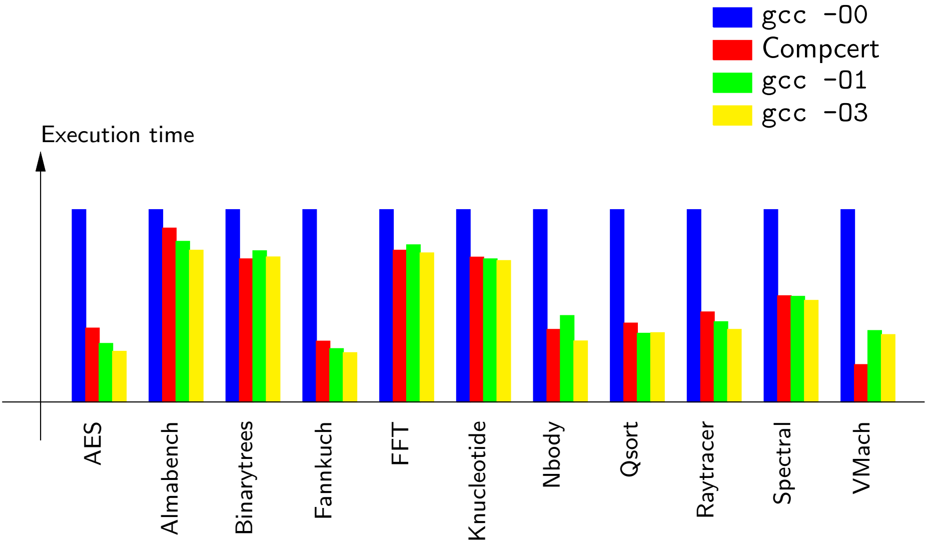
- Uses monads to deal with errors and state.
- Purely functional (persistent) data structures.

(6000 lines of Coq + 2000 lines of non-verified Caml code.)

Coq's extraction mechanism produces executable Caml code from these specifications.

Probably the biggest program ever extracted from a Coq development.

Performances of the generated code



For more information

Source distribution, commented specifications, papers:

`http://compcert.inria.fr/`

Outline

1. Introduction: Can you trust your compiler?

2. Formally verified compilers

3. The Compcert experiment

4. An example of verification: the register allocation pass

5. An example of verified translation validation: lazy code motion

6. Perspectives

The RTL intermediate language

Register Transfer Language, a.k.a. 3-address code.

The code of a function is represented by a control-flow graph:

- Nodes $=$ instructions corresponding roughly to that of the processor, operating over variables (temporaries).

`z = x +f y`	float addition
`i = i + 1`	integer immediate addition
`if (x > y)`	test and conditional branch

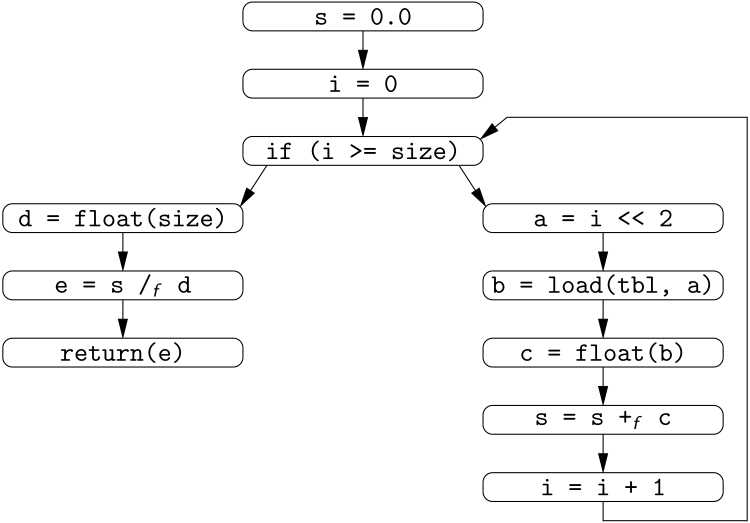
- Edge from I to J $\quad = \quad$ J is a successor of I (J can execute just after I).

Example: the C source code

```c
double average(int * tbl, int size)
{
    double s = 0;
    int i;

    for (i = 0; i < size; i++) s += tbl[i];
    return s / size;
}
```

Example: the corresponding RTL graph



s = 0.0

i = 0

if (i >= size)

d = float(size)

$e = s /_f d$

return(e)

a = i << 2

b = load(tbl, a)

c = float(b)

$s = s +_f c$

i = i + 1

Register allocation

Purpose: refine the notion of variables used as arguments and results of RTL operations.

- RTL (before register allocation):
  an unbounded quantity of variables.
- LTL (after register allocation):
  a fixed number of hardware registers;
  an unbounded number of stack slots.

(Insertion of spilling and reloading code is performed by a later pass.)

Objective: maximize the use of registers.

Technique used: coloring of an interference graph.

Algorithm, 1: Liveness analysis

Set up dataflow equations:

$$\begin{array}{rcl} L_{in}(p) & = & \texttt{transf}(L_{out}(p), \texttt{instr-at}(p)) \\ L_{out}(p) & = & \bigcup \{ L_{in}(s) \mid s \text{ successof of } p \} \end{array}$$

where, for instance,

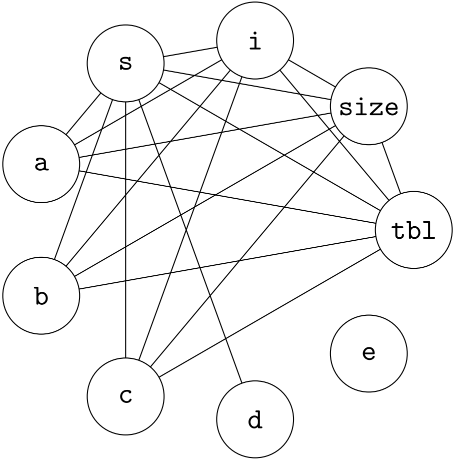
$$\texttt{transf}(X, r := op(r_1, \ldots, r_n)) \;=\; (X \setminus \{r\}) \cup \{r_1, \ldots, r_n\}$$

Solve these equations using fixpoint iteration (Kildall's algorithm).

Algorithm, 2: construct interference graph

For each instruction $p: \ r := \ldots$, add edges between r and $L_{out}(p) \setminus \{r\}$.

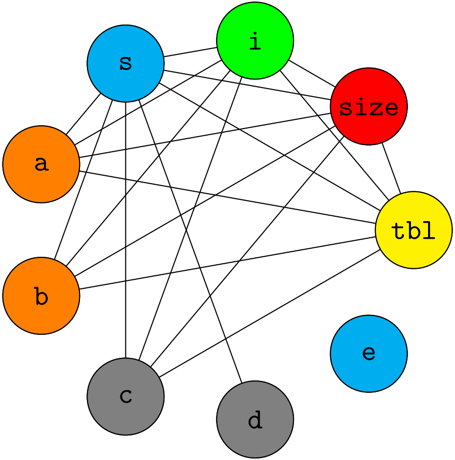
(+ Chaitin's special case for moves.) (+ Recording of preferences.)



Algorithm, 3: Coloring of the interference graph

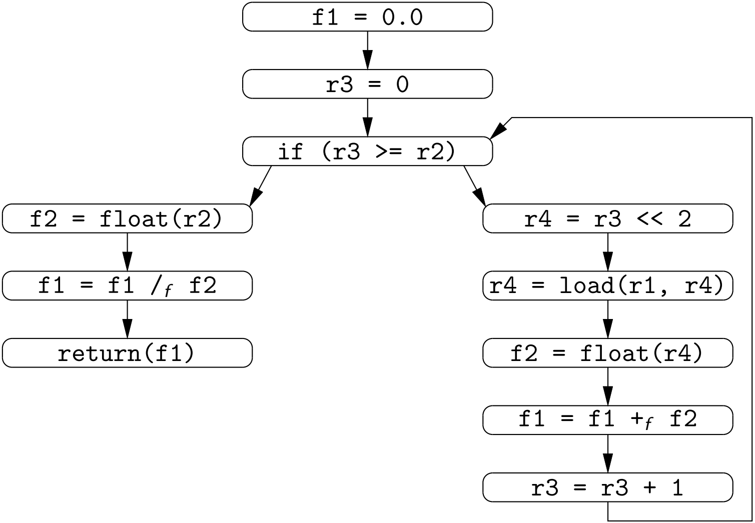
Construct a function $\phi : \mathit{Variable} \rightarrow \mathit{Register} + \mathit{Stackslot}$ such that $\phi(x) \neq \phi(y)$ if x and y interfere.

We use the Appel-George coloring heuristic.



Algorithm, 4: Rewriting the code

Replace all variables x by their color $\phi(x)$.



f1 = 0.0

r3 = 0

if (r3 >= r2)

f2 = float(r2)

f1 = f1 $/_f$ f2

return(f1)

r4 = r3 << 2

r4 = load(r1, r4)

f2 = float(r4)

f1 = f1 $+_f$ f2

r3 = r3 + 1

What needs to be proved?

Part 1: proofs of algorithms

Liveness analysis: show (by induction on the number of iterations) that the mapping L_out computed by Kildall's algorithm satisfies the inequations

$$L_{out}(p) \supseteq \texttt{transf}(L_{out}(s), \texttt{instr-at}(p)) \text{ if } s \text{ successor of } p$$

Construction of the interference graph: show that the final graph G contains all expected edges, e.g.

$$p : x := \ldots \;\; \wedge \;\; y \neq x \wedge y \in L_{out}(p) \Longrightarrow (x, y) \in G$$

Coloring of the interference graph: show that

$$(x, y) \in G \Longrightarrow \phi(x) \neq \phi(y)$$

We use verified validation:

- Validator: enumerate all edges (x, y) of G and abort if $\phi(x) = \phi(y)$
- Correctness proof for the validator: trivial.

What needs to be proved?

Part 2: semantic preservation proof

What does "x is live at p" means, semantically?

Hmmm . . .

What does "x is dead at p" means, semantically?

That the program behaves the same regardless of the value of x at point p.

Invariant

Let E : variable → value be the values of variables at point p in the original program. Let R : location → value be the values of locations at point p in the transformed program.

E and R agree at p, written $p \vdash E \approx R$, iff

$$E(x) = R(\phi(x)) \text{ for all } x \text{ live before point } p$$

What needs to be proved?

Part 2: semantic preservation proof

What does “x is live at p” means, semantically?

Hmmm . . .

What does “x is dead at p” means, semantically?

That the program behaves the same regardless of the value of x at point p.

Invariant

Let E : variable → value be the values of variables at point p in the original program. Let R : location → value be the values of locations at point p in the transformed program.

E and R agree at p, written $p \vdash E \approx R$, iff

$$E(x) = R(\phi(x)) \textit{ for all } x \textit{ live before point } p$$

What needs to be proved?

Part 2: semantic preservation proof

What does "x is live at p" means, semantically?

Hmmm . . .

What does "x is dead at p" means, semantically?

That the program behaves the same regardless of the value of x at point p.

Invariant

Let E : variable → value be the values of variables at point p in the original program. Let R : location → value be the values of locations at point p in the transformed program.

E and R agree at p, written $p \vdash E \approx R$, iff

$$E(x) = R(\phi(x)) \text{ for all } x \text{ live before point } p$$

Proving that the code transformation preserves semantics

Show a simulation diagram of the form

$$\begin{tikzcd}
p, E, M \arrow[r, dash, "p \vdash E \approx R"] \arrow[d, "t"] & p, R, M \arrow[d, dotted, "t"] \\
p', E', M' \arrow[r, dash, dotted, "p' \vdash E' \approx R'"] & p', R', M'
\end{tikzcd}$$

Hypotheses: left, a transition in the original code; top, the invariant (register agreement) before the transition.

Conclusions: one transition in the transformed code; bottom, the invariant after the transition.

Semantic preservation for whole executions

$$\begin{tikzcd}
\text{(initial state)} & S_1 \arrow[r, dash, "\textit{invariant}"] \arrow[d, "\epsilon"'] & T_1 \arrow[d, "\epsilon"] & \text{(initial state)} \\
 & S_2 \arrow[r, dash, "\textit{invariant}"'] \arrow[d, "\nu_1"'] & T_2 \arrow[d, "\nu_1"] & \\
 & S_3 \arrow[r, dash, "\textit{invariant}"'] \arrow[d, "\nu_2"'] & T_3 \arrow[d, "\nu_2"] & \\
 & S_4 \arrow[r, dash, "\textit{invariant}"'] \arrow[d, "\epsilon"'] & T_4 \arrow[d, "\epsilon"] & \\
\text{(final state)} & S_5 \arrow[r, dash, "\textit{invariant}"'] & T_5 & \text{(final state)}
\end{tikzcd}$$

Proves that the original program and the transformed program have the same behavior (the trace $t = \nu_1.\nu_2$).

Outline

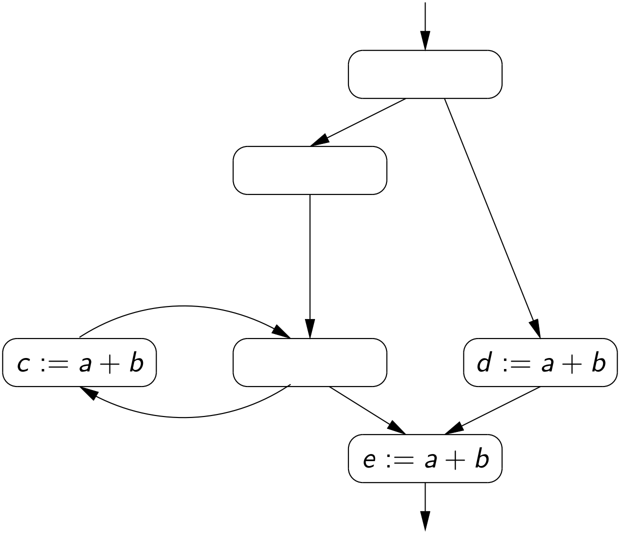
1. Introduction: Can you trust your compiler?
2. Formally verified compilers
3. The Compcert experiment
4. An example of verification: the register allocation pass
5. An example of verified translation validation: lazy code motion
6. Perspectives

Lazy code motion

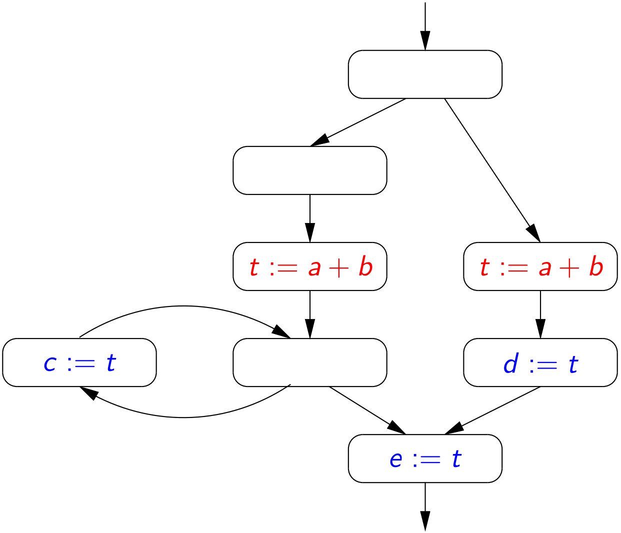
Lazy code motion (Knoop, Rüthing & Steffen, 1992) and its predecessor, partial redundancy elimination (Morel & Renvoise, 1979), perform:

- Elimination of common subexpressions, even across basic blocks.
- Loop invariant code motion.
- Factoring of partially redundant computations (i.e. computations that occur multiple times on some paths, but 0 or 1 times on others.)

An example of lazy code motion



An example of lazy code motion



Proving the correctness of lazy code motion?

A mechanized correctness proof of lazy code motion appears very difficult:

- LCM exploits the results of no less than 4 dataflow analyses.
- LCM is a highly non-local transformation: instructions are moved across basic blocks and even across loops.
- The transformation generates fresh temporaries, which adds significant bureaucratic overhead to mechanized proofs.

Alternative: verified translation validation for LCM
(J.-B. Tristan)

Unverified, untrusted implementation of the transformation (in Caml):

- Can use bitvectors, imperative data structures, etc.
- Easy to experiment with variants.

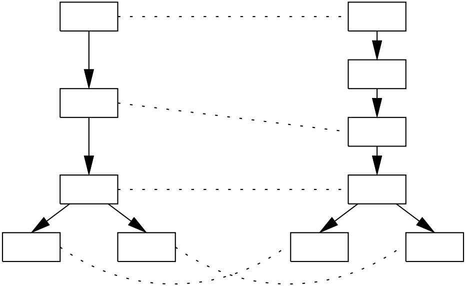
A posteriori validation with a validator written and proved correct in Coq:

- Input: the code before and after LCM.
- Output: a boolean, `true` = "semantics is preserved", `false` = "I don't know".

The validation algorithm

Pass 1:

- Define a mapping from instructions of the original program to instructions of the transformed program.
  (This mapping can be provided by the untrusted transformation.)
- Check that this mapping embeds the original control-flow graph in the transformed control-flow graph.



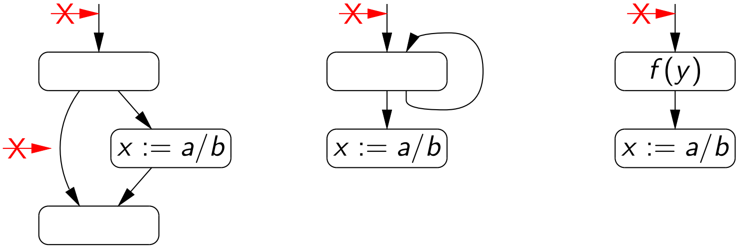
The validation algorithm

Pass 2: check each matching pairs of instructions.

Original instruction	Transformed instruction	Action
None	$t := op(y, z)$	Check that the computation $op(y, z)$ is anticipable at this point in the original program (see later).
$x := op(y, z)$	$x := t$	Check that the equality $t = op(y, z)$ holds at this point in the transformed program, based on the results of a standard reaching definition analysis.
Otherwise	Otherwise	Check that the two instructions are identical

The anticipability problem

Consider a computation that can go wrong at run-time, such as an integer division a/b.



If we place a computation of a/b at one of the X points, the transformed program can crash on a division by zero while the original program didn't.

Anticipability criterion: a computation a/b is anticipable at point p if all execution paths starting at p eventually compute a/b.

Proving the correctness of the validator

Assuming the validator returns `true`, show the simulation diagram:

$$\begin{tikzcd}
p, E, M \arrow[rr, dash, "Invariant"] \arrow[d, "t"] & & p, E_1, M \arrow[d, dotted, "t", "+"'] \\
p', E', M' \arrow[rr, dash, dotted, "Invariant"] \arrow[d] & & p', E'_1, M' \\
\text{(doesn't go wrong)} & &
\end{tikzcd}$$

The invariant includes:

- Agreement on the values of non-temporary variables: $E_1(x) = E(x)$ for all $x \in \mathrm{Dom}(E)$
- The equations inferred by reaching definition analysis are satisfied.

Assessment

The definition and correctness proof of the validator are not small (7000 lines of Coq). So, was the verified validator approach effective?

- Yes, because the proof remains conceptually simple.
  In particular, only 2 dataflow analyses are used (reaching definitions and anticipability), both of which have simple semantic characterizations.
- Yes, because the validator (possibly with extensions) could be reused for other optimizations.

Outline

1. Introduction: Can you trust your compiler?

2. Formally verified compilers

3. The Compcert experiment

4. An example of verification: the register allocation pass

5. An example of verified translation validation: lazy code motion

6. Perspectives

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.

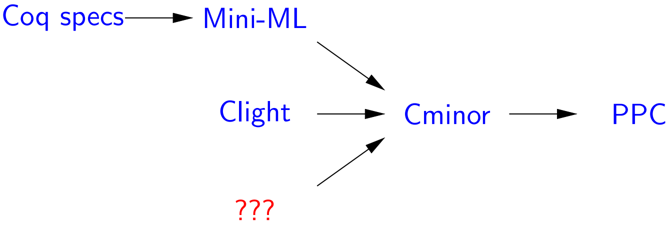
What next?

Enhancements to Compcert

Much remains to be done on the Compcert compiler:

- Handle a larger subset of C.
  (E.g. with `goto`.)
- Deploy and prove correct more optimizations.
  (E.g. global value numbering, using the "verified validator" approach.)
- Prove semantic preservation for concurrent programs.
  (Hard! Need to restrict to race-free source programs.)
- Target other processors beyond the PowerPC.
  (ARM: in progress.)
- Test usability on real-world embedded codes.

Front-ends for other source languages



An experiment in progress for a small functional language (mini-ML).

Main difficulty: proving the run-time system (allocator, GC) and interfacing this proof with that of the compiler.

What about a reactive / synchronous language, for instance?

The context on the "output" side

Bridging the gap between compiler verification and processor verification:

$$\xrightarrow{\text{Compcert}} \begin{array}{c}\text{Assembly}\\\text{language}\end{array} \xrightarrow[\text{linker}]{\text{assembler}} \begin{array}{c}\text{Machine}\\\text{code}\end{array} \xrightarrow[\text{architecture}]{\text{micro-}} \text{Circuits}$$

Some inspiring verification work in this area:

- From Piton assembly language to NDL netlist
  (J. Strother Moore et al, 1996)
- From ARM machine code to ARM6 micro-architecture
  (Anthony Fox, U. Cambridge, 2003)
- The Verisoft project
  (Wolfgang Paul et al, Germany, ongoing)

To finish. . .

The formal verification of compilers and other programming tools

. . . could be worthwhile,

. . . appears to be feasible,

. . . and is definitely exciting!

To finish. . .

The formal verification of compilers and other programming tools

. . . could be worthwhile,

. . . appears to be feasible,

. . . and is definitely exciting!

To finish. . .

The formal verification of compilers and other programming tools

. . . could be worthwhile,

. . . appears to be feasible,

. . . and is definitely exciting!

To finish. . .

The formal verification of compilers and other programming tools

. . . could be worthwhile,

. . . appears to be feasible,

. . . and is definitely exciting!