Formal verification of an optimizing compiler

Xavier Leroy

INRIA Rocquencourt

RTA 2007

INRIA

The compilation process

General definition: any automatic translation from a computer language to another.

Restricted definition: efficient ("optimizing") translation from a source language (understandable by programmers) to a machine language (executable in hardware).

A mature area of computer science:

- Already 50 years old! (Fortran I: 1957)
- Huge corpus of code generation and optimization algorithms.
- Many industrial-strength compilers that perform subtle transformations.

An example of optimizing compilation

```
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;
}
```

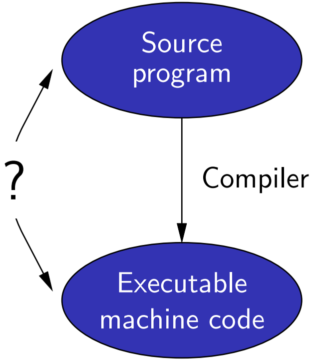
Compiled for the Alpha processor and manually decompiled back to C. . .

```
double dotproduct(int n, double a[], double b[]) {
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
    dp += f28;  dp += f1; dp += f10; dp += f11;
    if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
    dp += f18;
    if (r1 < n) goto L19;
L5:  return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
```

```
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
```

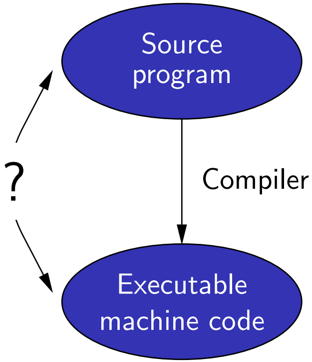
```
double dotproduct(int n, double a[], double b[]) {
     dp = 0.0;
     if (n <= 0) goto L5;
     r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
     if (r2 > n || r2 <= 0) goto L19;
     prefetch(a[16]); prefetch(b[16]);
     if (4 >= r2) goto L14;
     prefetch(a[20]); prefetch(b[20]);
     f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
     r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
     f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
     f12 = a[4]; f16 = f18 * f16;
     f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
     f11 += f17; r1 += 4; f10 += f15;
     f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
     f1 += f16; dp += f19; b += 4;
     if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
     f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
     f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
     a += 4; b += 4; f14 = a[8]; f15 = b[8];
     f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
     f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
     a += 4; f28 = f29 * f28; b += 4;
     f10 += f14; f11 += f12; f1 += f26;
     dp += f28;  dp += f1; dp += f10; dp += f11;
     if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
     dp += f18;
     if (r1 < n) goto L19;
L5:  return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
```

Can you trust your compiler?



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

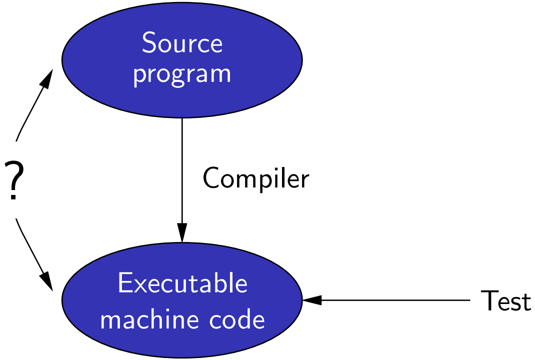
Can you trust your compiler?



Non-critical sofware:

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

Can you trust your compiler?

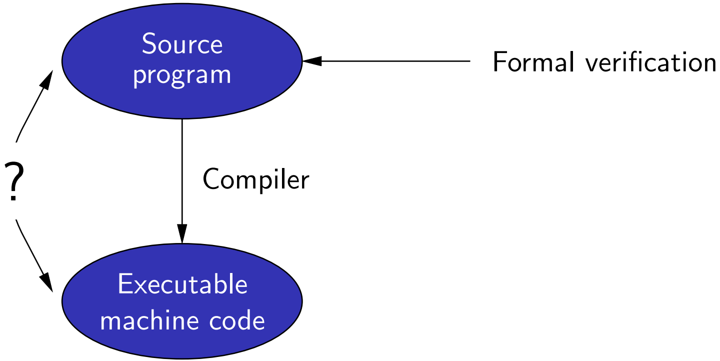


Critical software certified by systematic testing:

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

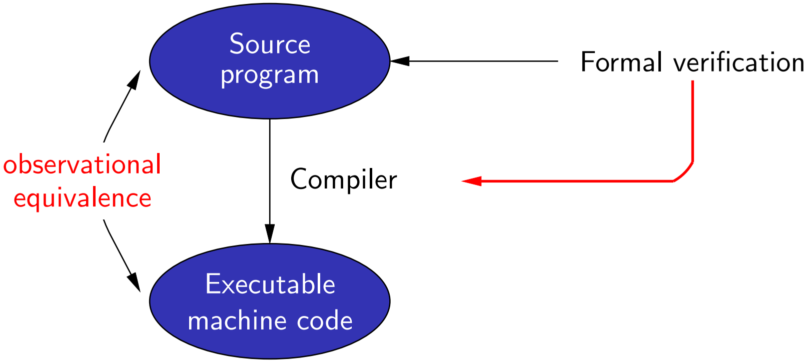
Compiler bugs are detected along with those of the program.

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.

Outline

Formal verification of compilers

Apply formal methods to the compiler itself to prove that it preserves the property of interest *Prop* of the source code:

Theorem

For all source codes S,
if the compiler generates machine code C from source S,
without reporting a compilation error,
and if S satisfies Prop,
then C satisfies Prop.

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

Some properties of interest

Among the properties of programs we'd like to see preserved:

1. Observable behaviour, including "going wrong".
2. Observable behaviour if the source code does not go wrong.
   Compilers are allowed to replace undefined behaviours by more specific behaviours.
3. Satisfaction of the functional specifications for the application.
   Implied by (2) if these specs are couched in terms of observable behaviour.
4. Type- and memory-safety.
   Implied by (2).

Approach 1: proving the compiler

Model the compiler as a function

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

and prove that

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

using a proof assistant.

It then follows that for any property P of the observable behaviour,

$$\forall S, C, \quad Comp(S) = C \land S \models P \Rightarrow C \models P$$

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:

$$\begin{array}{rcl} \mathit{Comp} & : & \mathit{Source} \rightarrow \mathit{Code} + \mathtt{Error} \\ \mathit{Validator} & : & \mathit{Source} \times \mathit{Code} \rightarrow \mathtt{bool} \end{array}$$

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

It suffices to prove that the validator is correct:

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

The compiler itself need not be proved.

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

$$\begin{array}{rcl} Comp & : & Source \times Prop \rightarrow Code \times Certificate + \texttt{Error} \\ Checker & : & Prop \times Code \times Certificate \rightarrow \texttt{bool} \end{array}$$

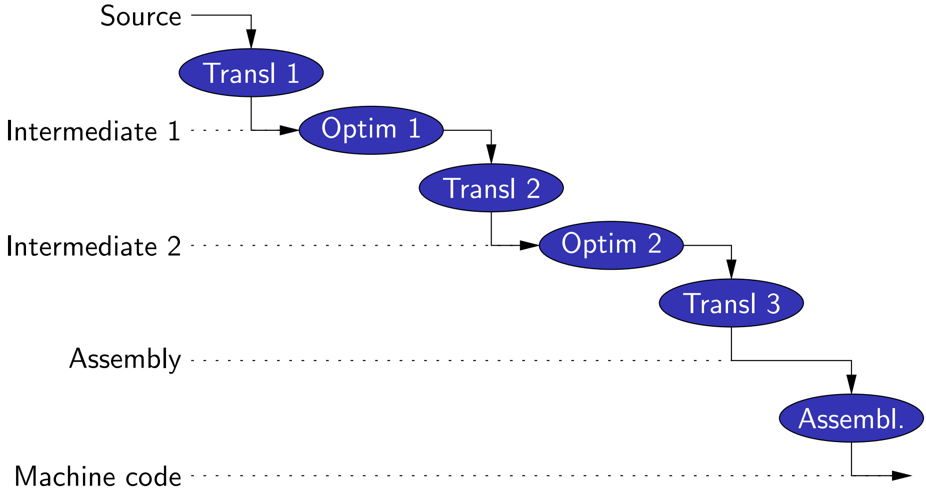
If $Comp(S, P) = (C, \pi)$ and $Validator(P, C, \pi) = \texttt{true}$, success.
Otherwise, error.

Assume that the checker is proved correct:

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

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

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
  $\Rightarrow$ 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

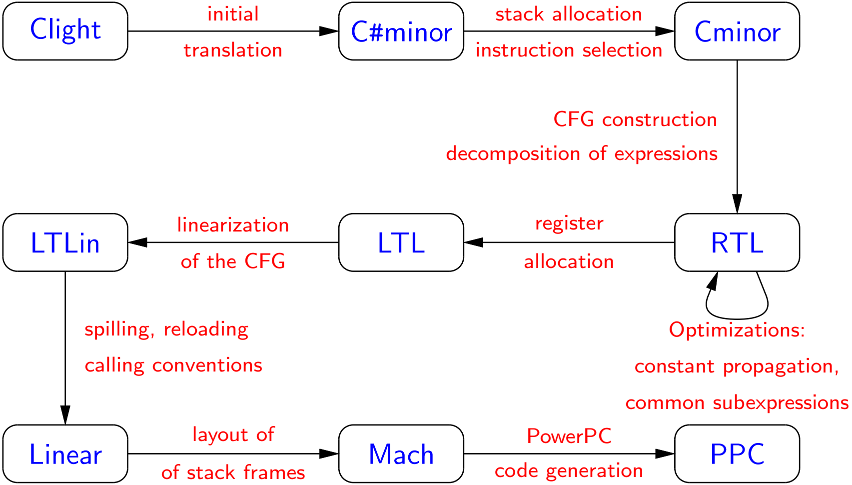
Supported:

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

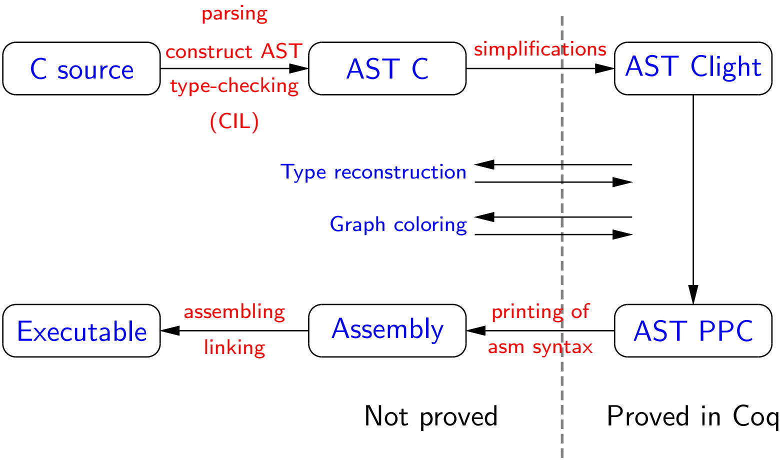
Not supported:

- The `long long` and `long double` types.
- Dynamic memory allocation `malloc`/`free`.
- `goto`, unstructured `switch`, `longjmp`/`setjmp`.
- Variable-arity functions.

The formally verified part of the compiler



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.

```
Theorem transf_c_program_correct:
    forall prog tprog trace n,
    transf_c_program prog = Some tprog ->
    Csem.exec_program prog trace (Vint n) ->
    PPC.exec_program tprog trace (Vint n).
```

What does the semantic preservation theorem says?

The formal semantics for the source and target languages associate to programs:

- a trace of input-output events (system calls);
- the integer returned by the `main` function (exit code).

The theorem guarantees that if the source program terminates and does not go wrong,

- the compiled code terminates and does not go wrong,
- performs exactly the same system calls,
- and returns the same exit code

as the source program.

Currently says nothing about source programs that do not terminate (work in progress).

The Coq proof

Approximately 2 man.years and 40000 lines of Coq:

Code	Sem.	Statements	Proof scripts	Misc
13%	8%	22%	50%	7%

Programmed in Coq

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

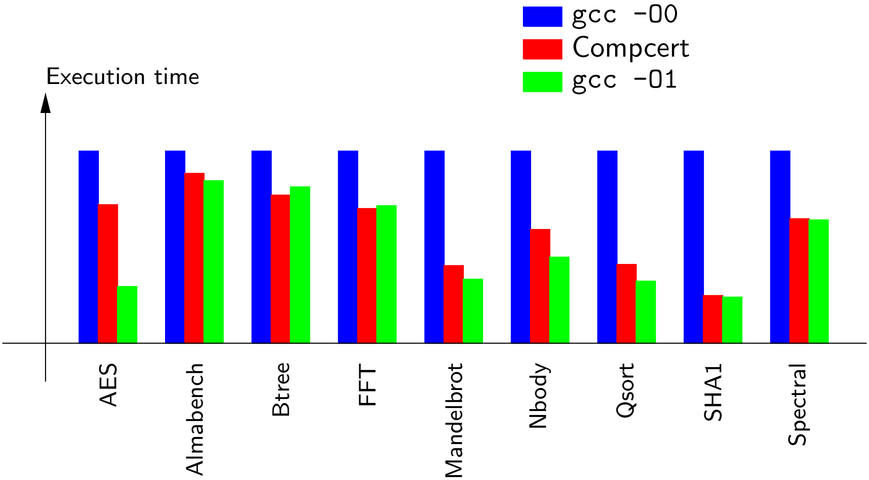
- Uses monads to deal with errors and state.
- Purely functional (persistent) data structures.

(4500 lines of Coq + 1500 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



Plan

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

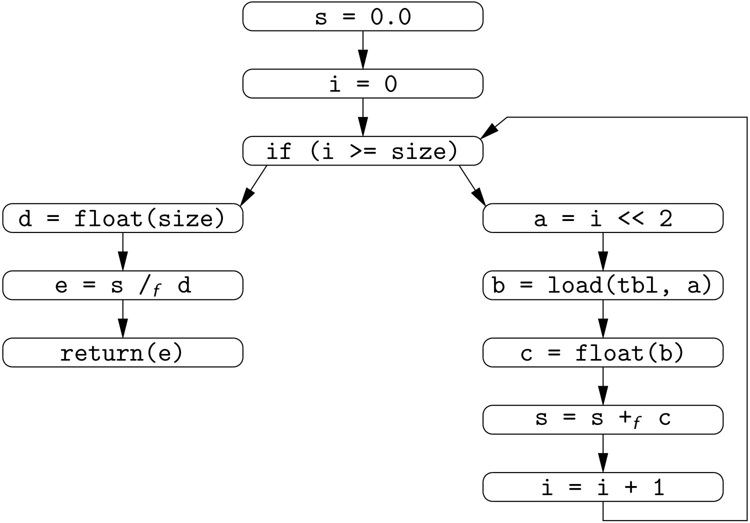
- Edge from I to J = J is a successor of I (J can execute just after I).

Example: the C source code

```
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



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.

Accessing registers is faster than accessing stack slots
$\longrightarrow$ maximize the use of registers.

Approaches to register allocation

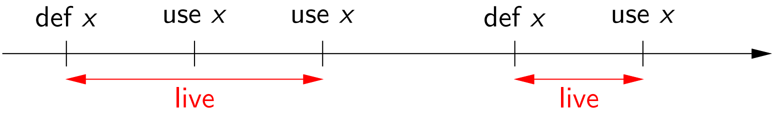
Naive approach:
Assign the N hardware registers to the N most used variables; assign stack slots to the other variables.

Finer approach:
Notice that the same hardware register can be assigned to several distinct variables, provided they are never used simultaneously.

Liveness analysis

A variable x is live at point p if an instruction reachable from p uses x, and x is not redefined in between.

In straight-line code, a variable becomes live at each definition and dies at its last uses.



If x is dead (not live) at a given point, the value of x at this point has no effect on the results of the computation.

Liveness analysis

Define

- $V(p)$ the set of variables live "before" the point p
- $V'(p)$ the set of variables live "after" the point p

Assume the instruction at p is

$$(r_1, \ldots, r_n) = instr(a_1, \ldots, a_m) \rightarrow s_1, \ldots, s_k$$

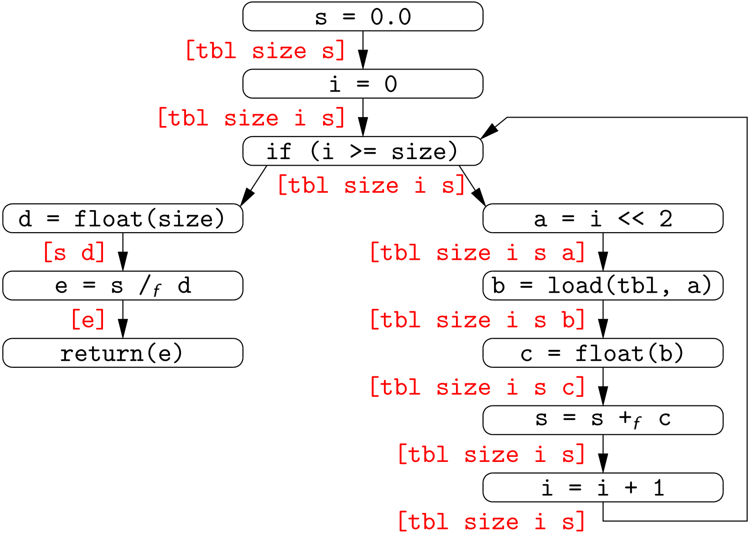
We have the following inequations over the V and V':

$$\begin{array}{rcl} V(p) & = & (V'(p) \setminus \{r_1, \ldots, r_n\}) \cup \{a_1, \ldots, a_m\} \\ V'(p) & \supseteq & V(s_1) \cup \ldots \cup V(s_k) \end{array}$$

These inequations define a backward dataflow analysis.

They can be solved easily by fixpoint iteration (Kildall's worklist algorithm).

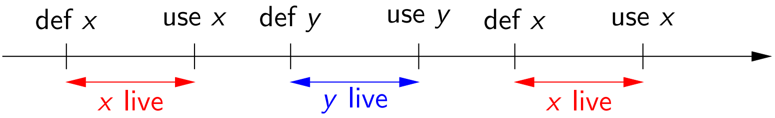
Example of liveness analysis



Using liveness information for register allocation

Two variables x and y interfere if they are both live at one point in the program.

If x and y do not interfere, they can share the same register or stack slot.

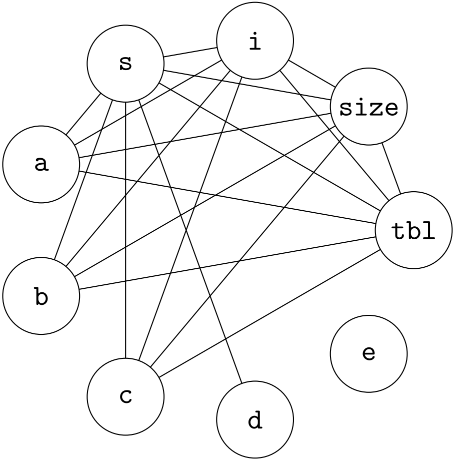


→ Determine the minimal number of registers needed by coloring of the graph representing the interference relation.

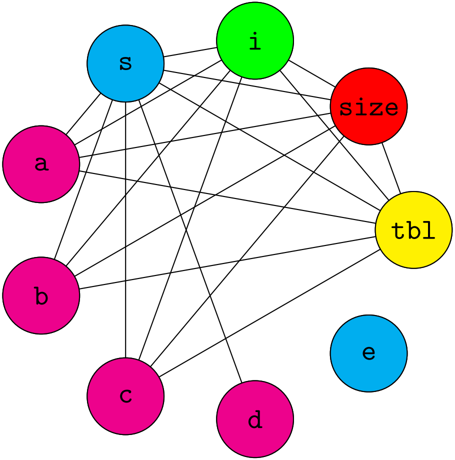
→ If this number is ≤ number of hardware registers, we obtain a perfect register allocation.

→ Otherwise, the coloring is a good starting point to determine which variables go into registers.

Example of an interference graph



Example of an interference graph



The algorithm for register allocation by graph coloring

1. Liveness analysis: compute the sets of live variables "before" $V(p)$ and "after" $V'(p)$ each program point p.
2. Construct the graph of the interference relation.
3. Coloring of this graph: construct a function

$$\phi : \mathit{Variable} \rightarrow \mathit{Register} + \mathit{Stackslot}$$

   so that $\phi(x) \neq \phi(y)$ if x and y interfere.
   (NP-hard, but good linear-time heuristics are known.)
4. Code transformation: replace each instruction

$$r := \mathit{instr}(a_1, \ldots, a_n) \rightarrow s_1, \ldots, s_k$$

   by

$$\phi(r) := \mathit{instr}(\phi(a_1), \ldots, \phi(a_n)) \rightarrow s_1, \ldots, s_k$$

The correctness proof of this algorithm

1. Liveness analysis: prove that the $V(p)$ and $V'(p)$ are indeed solutions of the dataflow inequations.
2. Interference graph construction: prove that for every instruction

   $$p: \quad r := instr(a_1, \ldots, a_n) \rightarrow s_1, \ldots, s_k$$

   the graph contains edges between r and each of the $x \in V'(p) \setminus \{r\}$.
3. Graph coloring: prove that $\phi(x) \neq \phi(y)$ if x and y interfere, either by proving directly the coloring heuristic, or by verifying a posteriori this property by edge enumeration.
4. Code transformation: next slides.

An overview of the semantics of RTL

A transition system $code \vdash (p, E, M) \rightarrow (p', E', M')$.

p, E, M: initial program point, values of variables, and memory state.

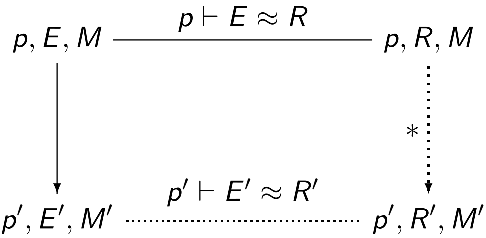
p', E', M': program point, values of variables, and memory state after executing the instruction at p.

These transitions are defined by inference rules such as

$$\frac{code(p) = (z := \texttt{add}(x, y) \rightarrow p') \qquad v = E(x) + E(y) \pmod{2^{32}}}{code \vdash (p, E, M) \rightarrow (p', E\{z \leftarrow v\}, M)}$$

Proving that the code transformation preserves semantics

Prove simulation diagrams of the form



Hypotheses: left, a transition in the original code; top, the invariant before the transition.

Conclusions: right, some transitions in the transformed code; bottom, the invariant after the transition.

The invariant $p \vdash E \approx R$ is defined by

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

Plan

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.

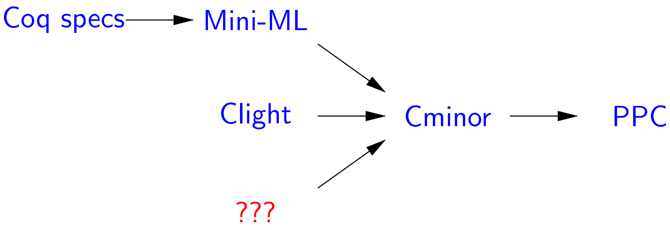
What next?

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.
  (Loop optimizations, instruction scheduling, ...)
- Prove semantic preservation for non-terminating programs (in progress); for concurrent programs? (hard!)
- Target other processors beyond the PowerPC.
- 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?

More generally: formal verification of programming tools

Besides compilers, many other programming tools are involved in the production and verification of critical software:

- Code generators (e.g. SCADE to C).
- Static analyzers, including type checkers.
- Model checkers.
- Program provers.

Formally verify these tools as well?

To finish. . .

The formal verification of compilers and other programming tools

. . . could be worthwhile,

. . . might be feasible,

. . . and is definitely exciting!

To finish. . .

The formal verification of compilers and other programming tools

. . . could be worthwhile,

. . . might be feasible,

. . . and is definitely exciting!

To finish. . .

The formal verification of compilers and other programming tools

. . . could be worthwhile,

. . . might be feasible,

. . . and is definitely exciting!

To finish. . .

The formal verification of compilers and other programming tools

. . . could be worthwhile,

. . . might be feasible,

. . . and is definitely exciting!