Verifying a compiler: Why? How? How far?

Xavier Leroy

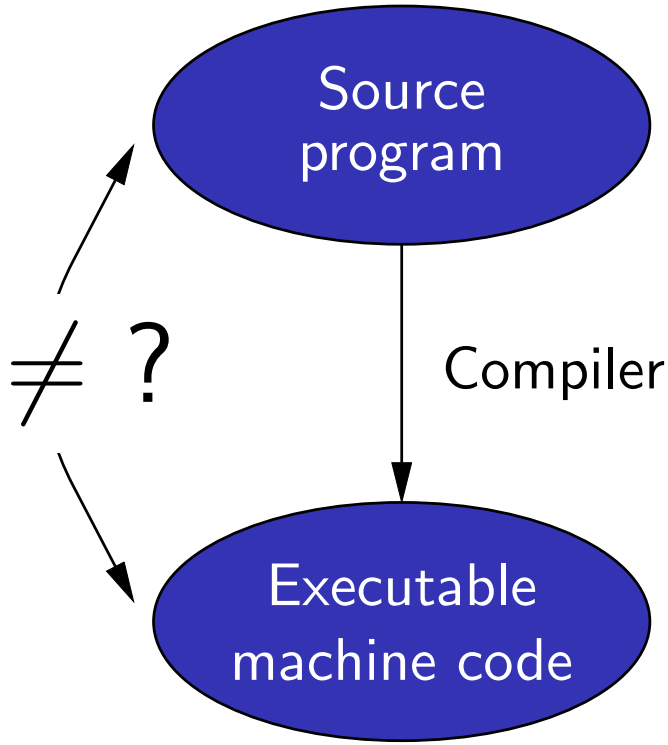
INRIA Paris-Rocquencourt

CGO 2011

Compiler verification:

Why?

Can you trust your compiler?



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

Miscompilation happens

NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated.

`http://www.nullstone.com/htmls/category/divide.htm`

We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables.

E. Eide & J. Regehr, EMSOFT 2008

To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input.

X. Yang, Y. Chen, E. Eide & J. Regehr, PLDI 2011

Exhibit A: GCC bug #323

Title: optimized code gives strange floating point results.

```
#include <stdio.h>

void test(double x, double y)
{
  double y2 = x + 1.0; // computed in 80 bits, not rounded to 64 bits
  if (y != y2) printf("error_n");
}

void main()
{
  double x = .012;
  double y = x + 1.0;  // computed in 80 bits, rounded to 64 bits
  test(x, y);
}
```

Why it is a bug: C99 allows intermediate results to be computed with excess precision, but requires them to be rounded at assignments.

Exhibit A: GCC bug #323

Reported in 2000.

Dozens of duplicates.

More than 150 comments.

Still not acknowledged as a bug.

"Addressed" in 2009 (in GCC 4.5) via flag `-fexcess-precision=standard`.

Responsible for PHP's `strtod()` function not terminating on some inputs...

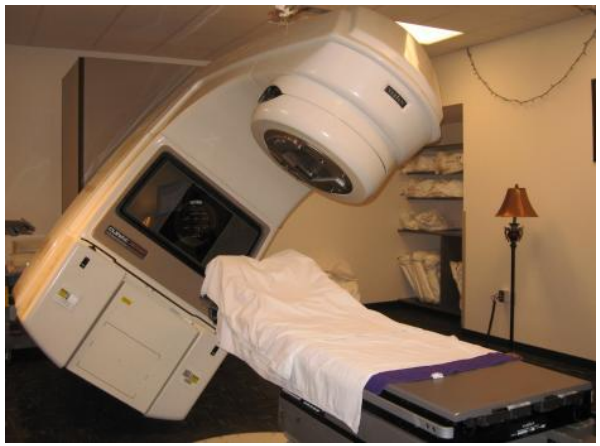
...causing denial of service on many Web sites.

Are miscompilation bugs a problem?

For ordinary software:

- Compiler-introduced bugs are negligible compared with the bugs in the program itself.
- Programmers rarely run into them.
- When they do, debugging is very hard.

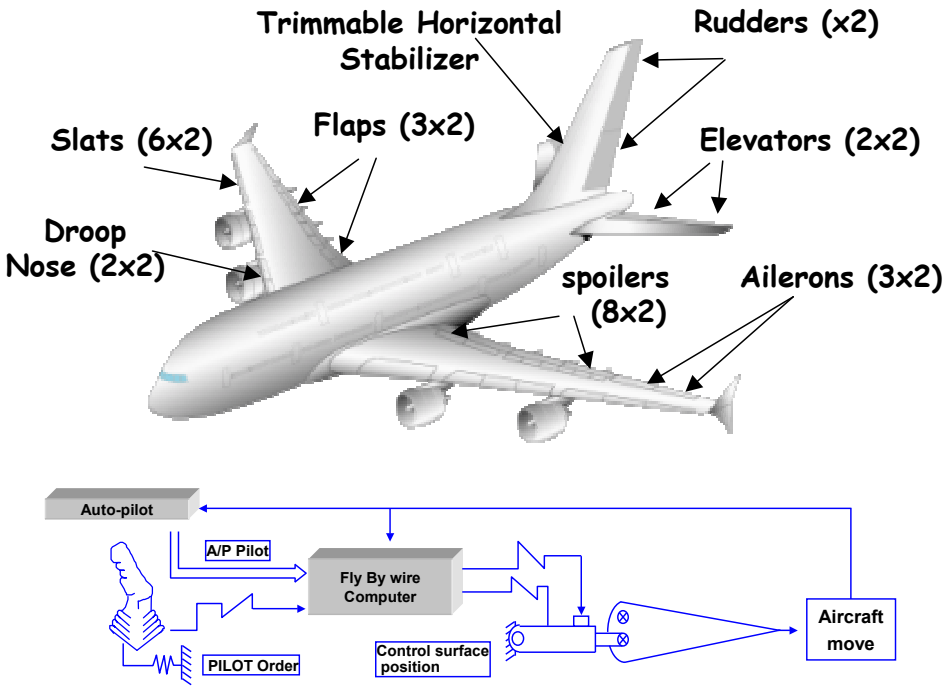
Are miscompilation bugs a problem?



For critical software validated by testing only:

- Good testing should find all bugs, even those compiler-introduced.
- Optimizations can complicate test plans.

Are miscompilation bugs a problem?

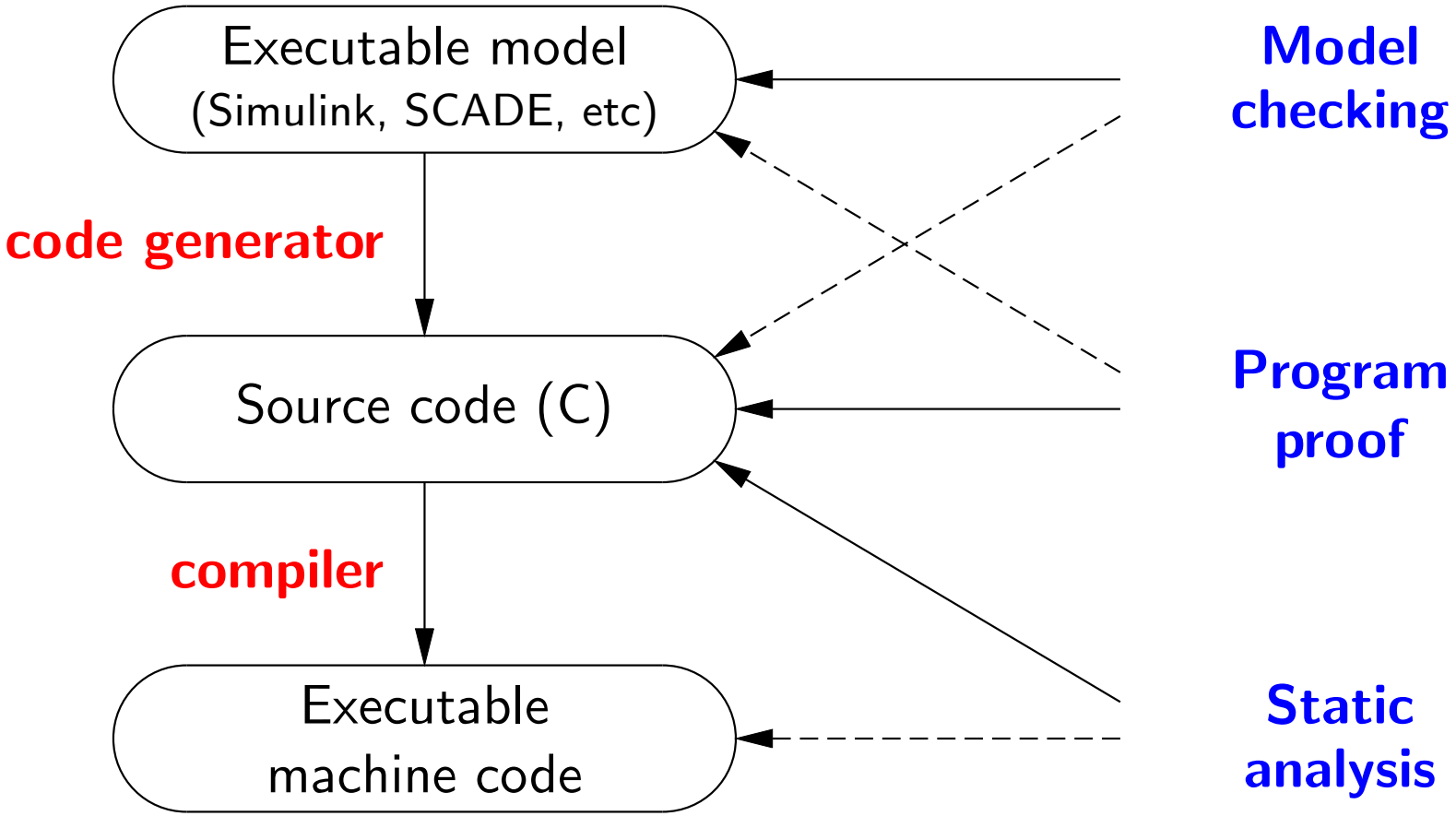


For critical software validated by review, analysis & testing:

(e.g. DO-178 in avionics)

- Manual reviews of (representative fragments of) generated assembly.
- Turning all optimizations off to get traceability.
- Reduced usefulness of formal verification.

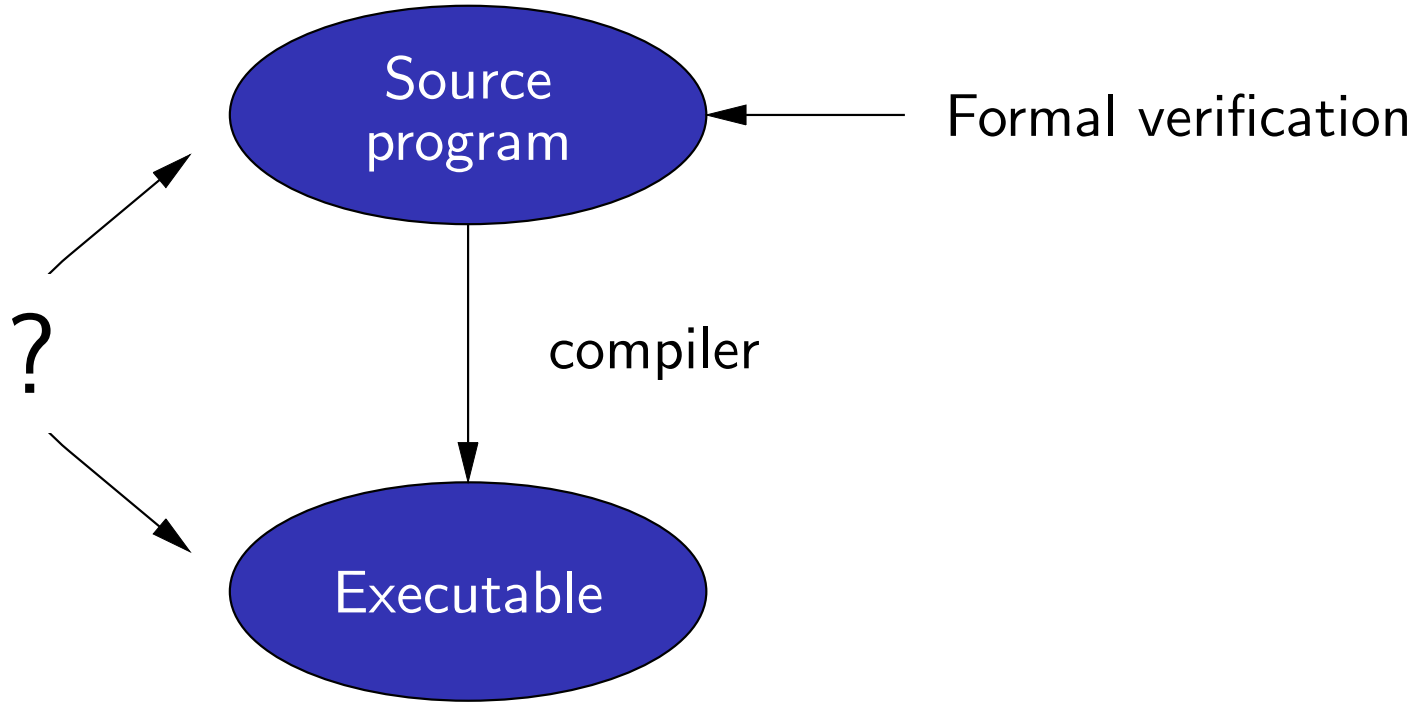
Miscompilation and formal verification



The guarantees obtained (so painfully!) by source-level formal verification may not carry over to the executable code ...

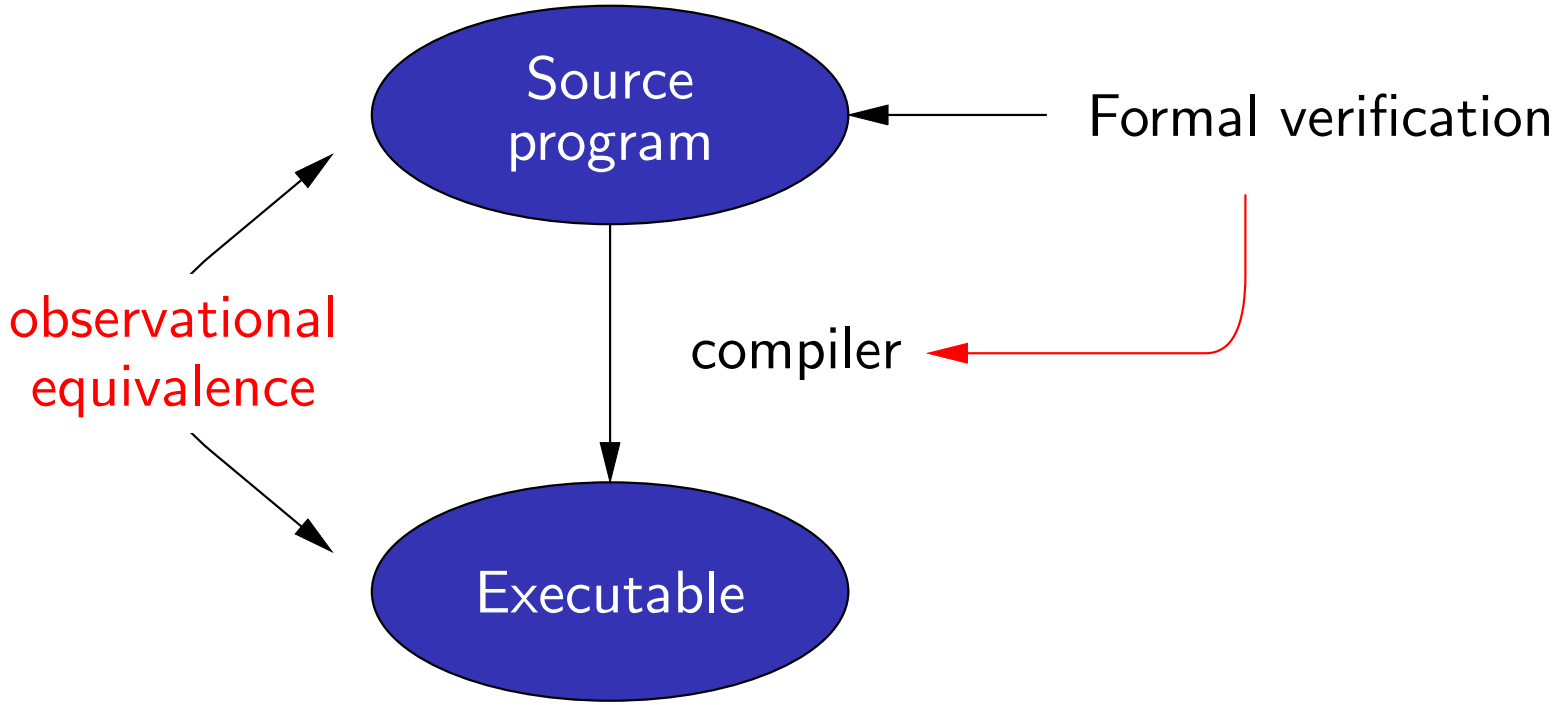
A solution? Verified compilers

With a regular compiler:



A solution? Verified compilers

With a formally verified compiler:



The properties formally established on the source program carry over to the executable.

Formal verification of compilers

A radical solution to the miscompilation problem:

Apply program proof to the compiler itself to prove that it preserves the semantics of the source code.

After all, compilers are complicated programs with a simple specification:

> *If compilation succeeds, the generated code should behave as prescribed by the semantics of the source program.*

An old idea...

John McCarthy
James Painter[1]

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS[2]

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

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

Mathematical Aspects of Computer Science, 1967

An old idea...

3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch

Computer Science Department
Stanford University

Abstract

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

Machine Intelligence (7), 1972.

Compiler verification:

How far are we today?

(X. Leroy, *Formal verification of a realistic compiler*, CACM 07/2009)

The CompCert project

(X.Leroy, S.Blazy, et al)

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

- Source language: a very large subset of C.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code
  $\Rightarrow$ careful code generation; some optimizations.

Note: compiler written from scratch, along with its proof; not trying to prove an existing compiler.

The subset of C supported

Supported natively:

- Types: integers, floats, arrays, pointers, `struct`, `union`.
- Expressions: all of C, including pointer arithmetic.
- Control: `if`/`then`/`else`, loops, `goto`, regular `switch`.
- Functions, including recursive functions and function pointers.
- Dynamic allocation (`malloc` and `free`).
- Volatile accesses.

The subset of C supported

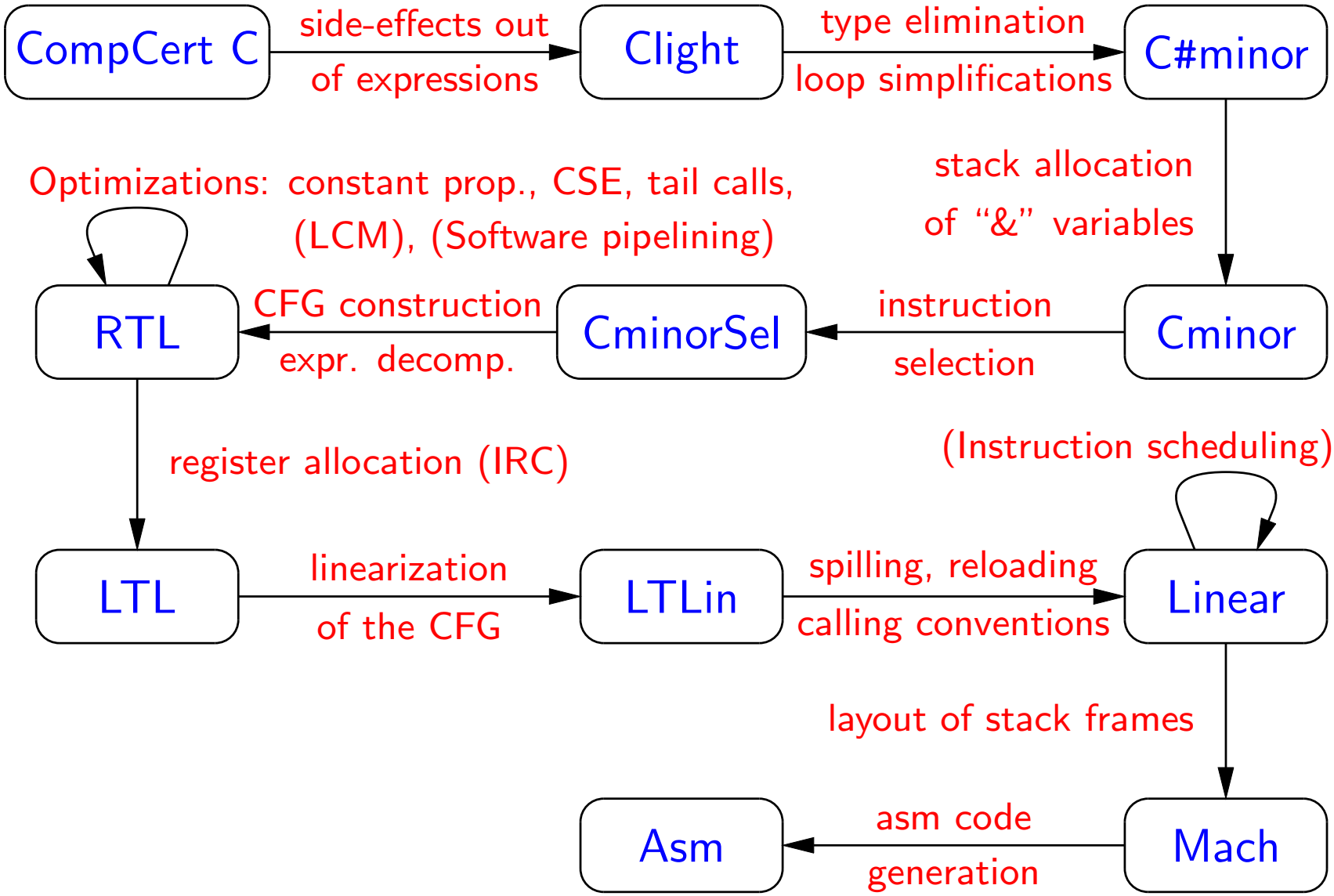
Not supported at all:

- The `long long` and `long double` types.
- Unstructured `switch` (Duff's device), `longjmp`/`setjmp`.
- Variable-arity functions.

Supported through (unproved!) expansion after parsing:

- Block-scoped variables.
- `typedef`.
- Bit-fields.
- Assignment between `struct` or `union`.
- Passing `struct` or `union` by value.

The formally verified part of the compiler



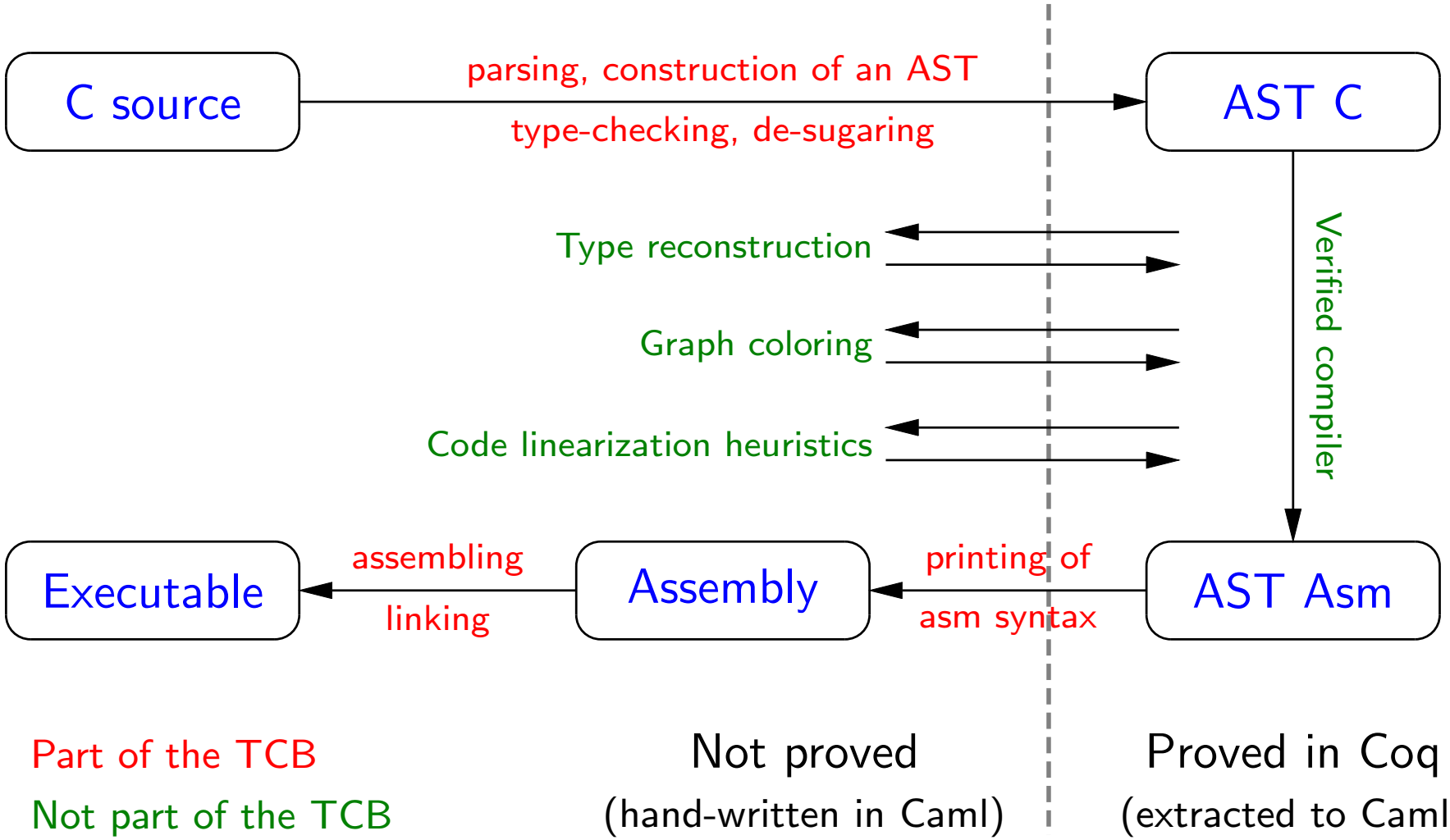
Formally verified in Coq

After 50 000 lines of Coq and 4 person.years of effort:

```
Theorem transf_c_program_is_refinement:
  forall p tp,
  transf_c_program p = OK tp ->
  (forall beh, exec_C_program p beh -> not_wrong beh) ->
  (forall beh, exec_asm_program tp beh -> exec_C_program p beh).
```

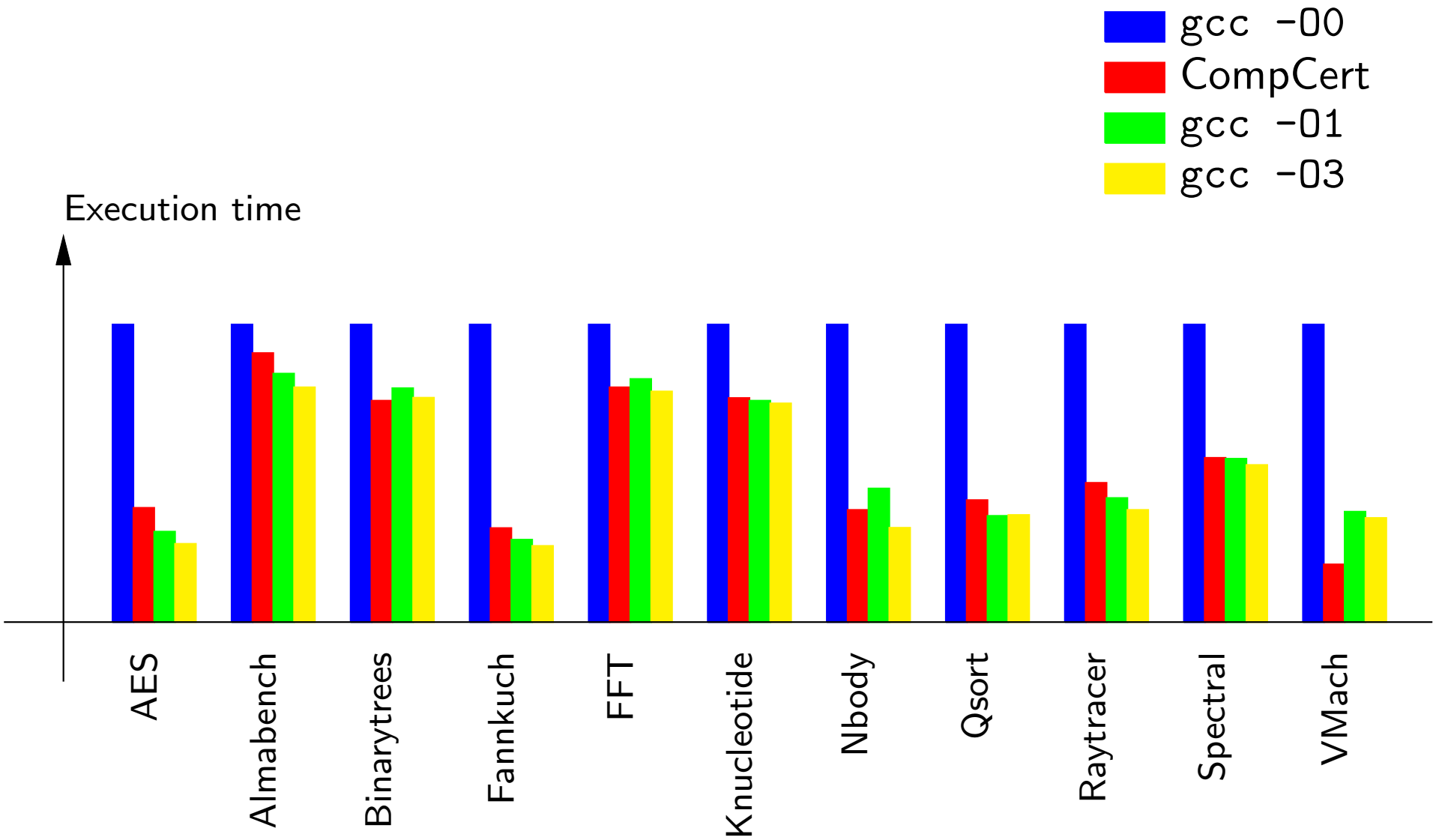
Behaviors `beh` = termination / divergence / going wrong
+ trace of I/O operations (syscalls, volatile accesses).

The whole CompCert compiler



Performance of generated code

(On a PowerPC G5 processor)



Status

Complete source & proofs available for evaluation and research purposes:

http://compcert.inria.fr/

(+ research papers)

Compiler runs on / produces code for
{Linux,MacOSX,Cygwin} / {PPC, ARM, x86}.

Tested on small benchmarks (up to 3000 LOC), real-world avionics codes, and by random testing.

As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

X. Yang, Y. Chen, E. Eide & J. Regehr, PLDI 2011

Compiler verification:

How?

Verification patterns

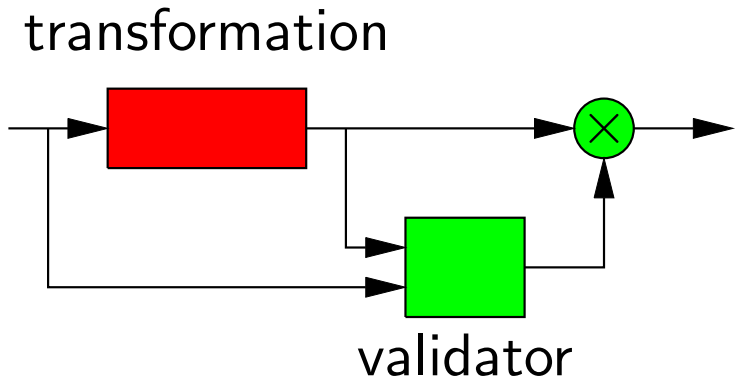
(for each compilation pass)

Verified transformation

= formally verified

= not verified

Verified translation validation



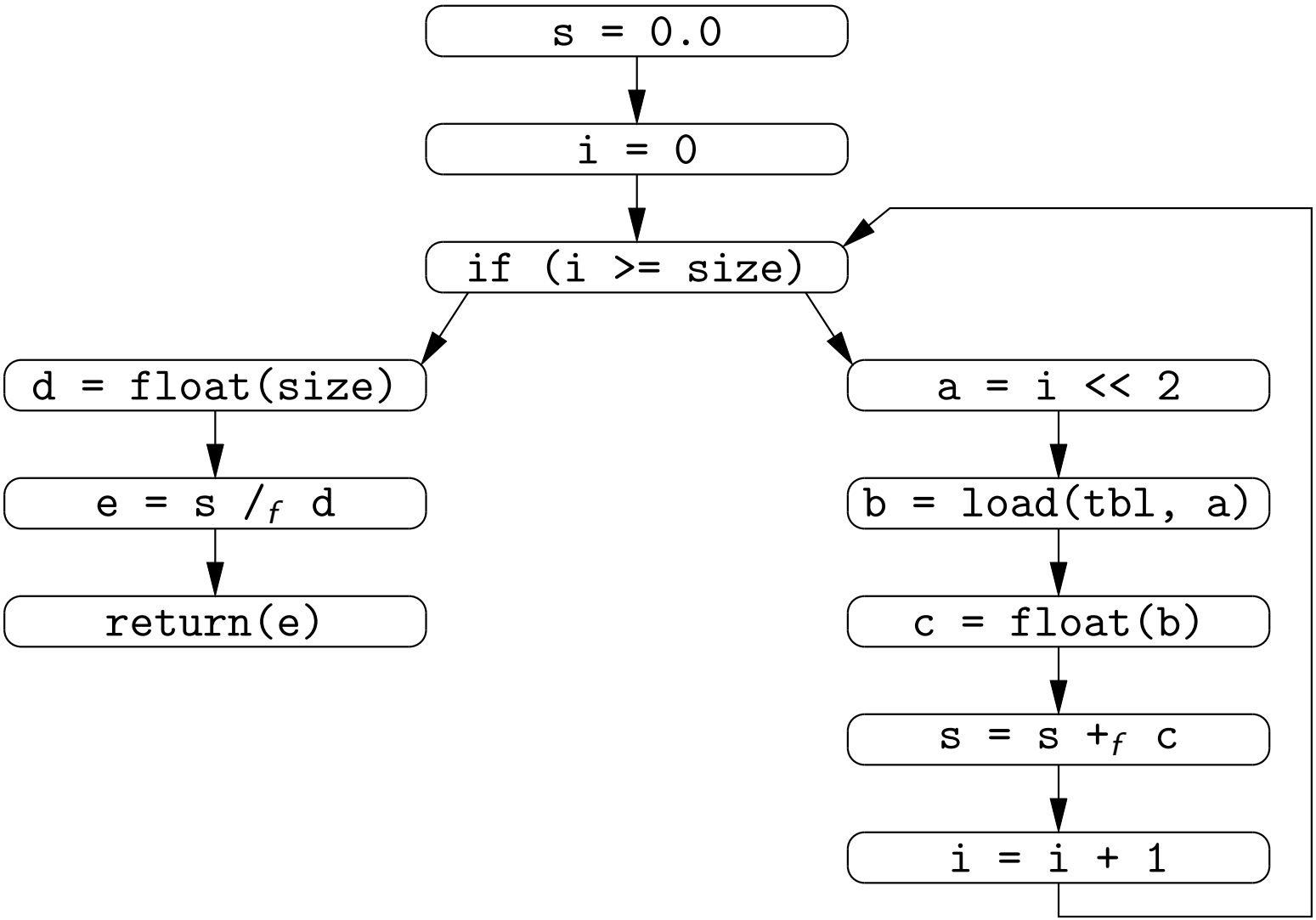
Verified translation validation:

- Less to prove (if validator simpler than transformation).
- Strong soundness guarantees, but no completeness in general.
- Validator reusable for several variants of an optimization.

An example of a verified transformation: register allocation by graph coloring

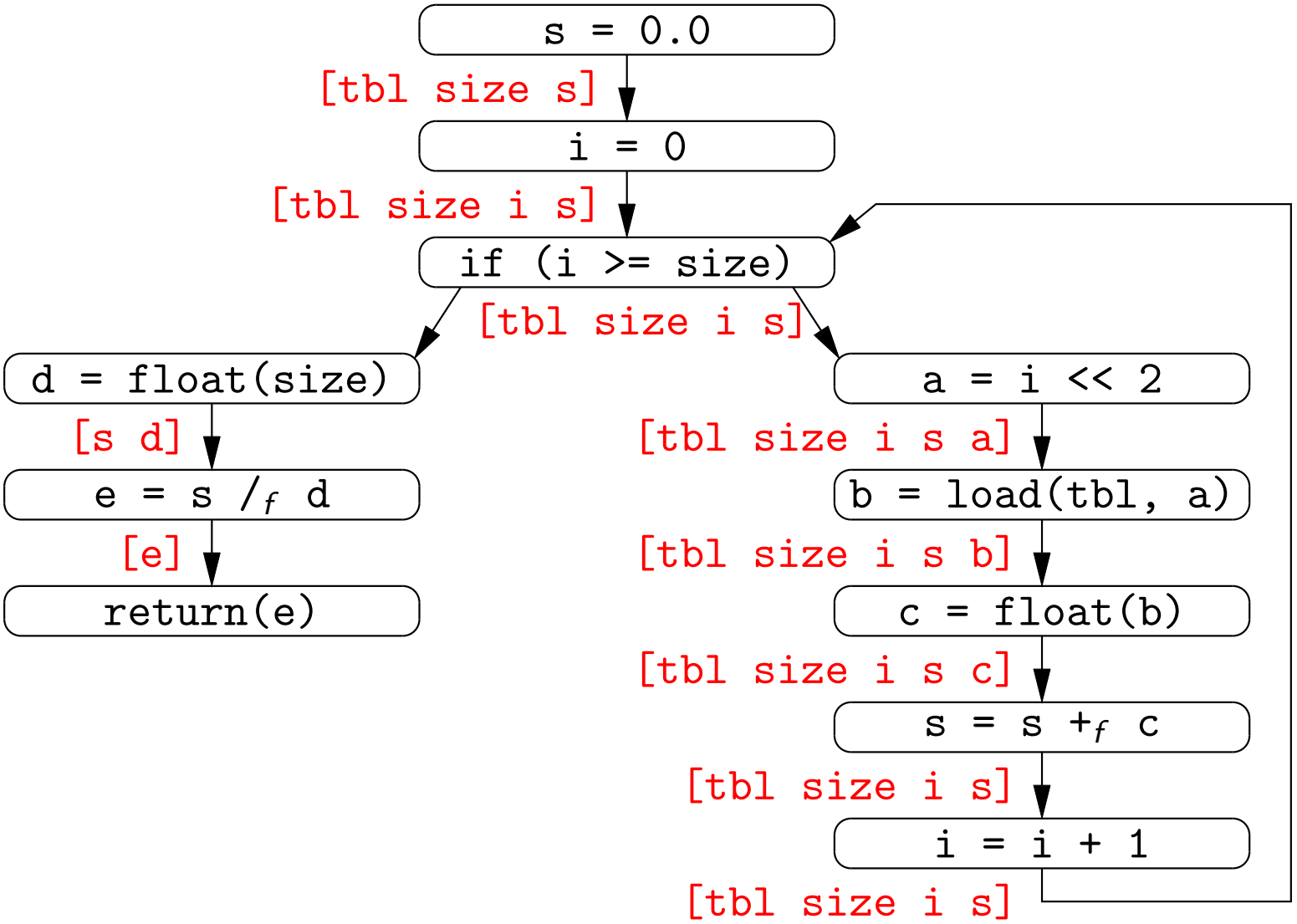
(X. Leroy, *A formally verified compiler back-end*, §8, J. Autom. Reasoning 43(4))
(S. Blazy, B. Robillard, A. W. Appel, *Formal verification of coalescing graph-coloring register allocation*, ESOP 2010)

Starting point: a CFG for a Register Transfer Language



Algorithm 1: liveness analysis (backward dataflow analysis)

$$L_{out}(p) = \bigcup \{ \texttt{transf}(L_{out}(s), \texttt{instr-at}(s)) \mid s \text{ successof of } p \}$$

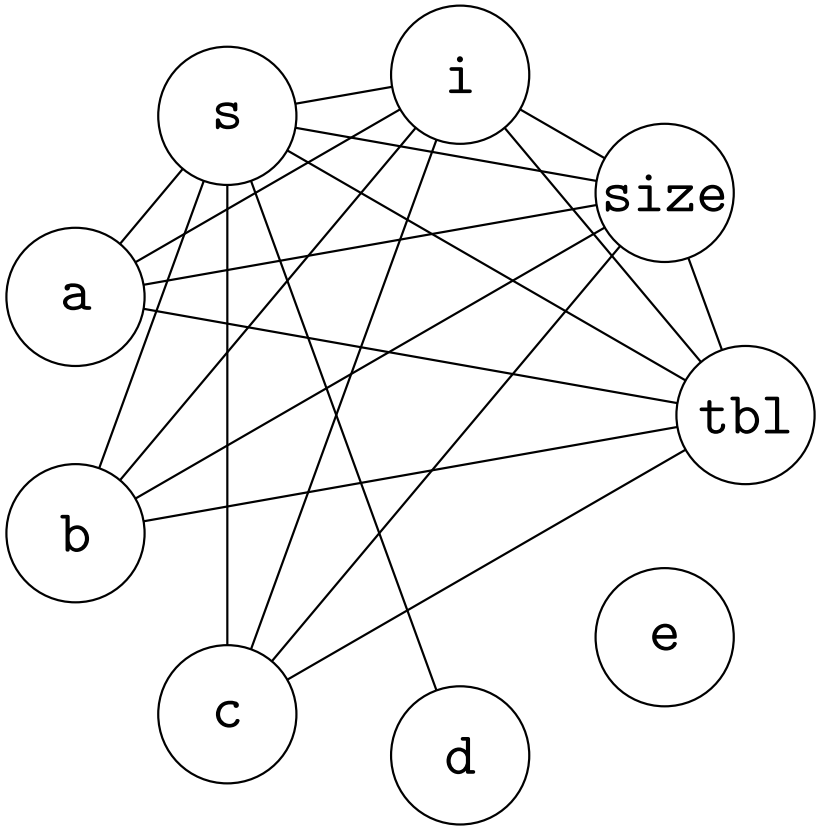


Algorithm 2: construct interference graph

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

(+ Chaitin's special case for moves.)

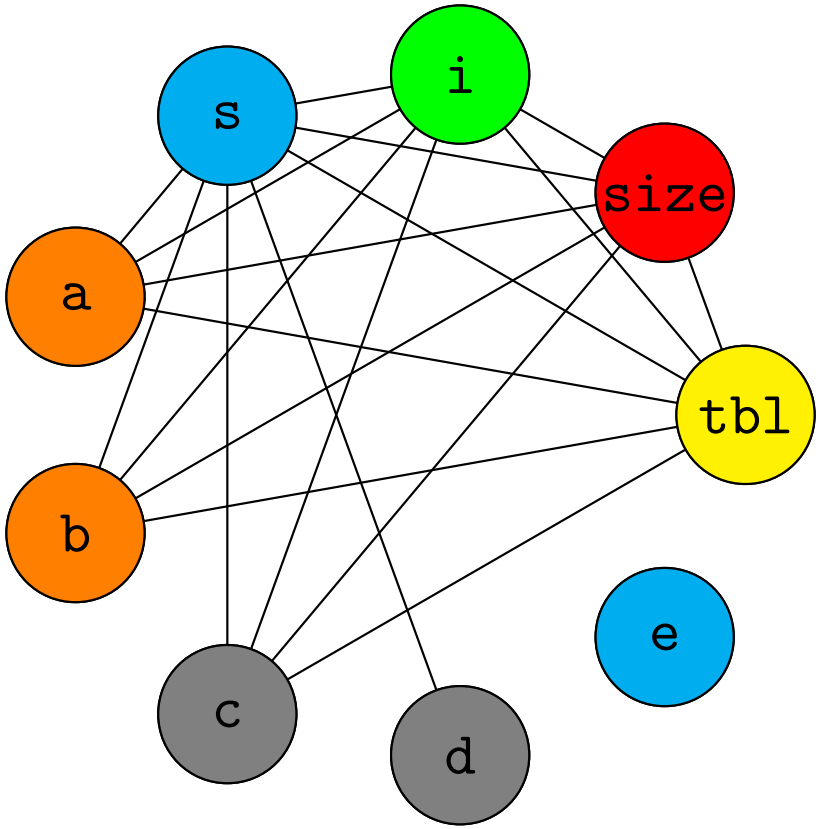
(+ Recording of preferences.)



Algorithm 3: Coloring of the interference graph

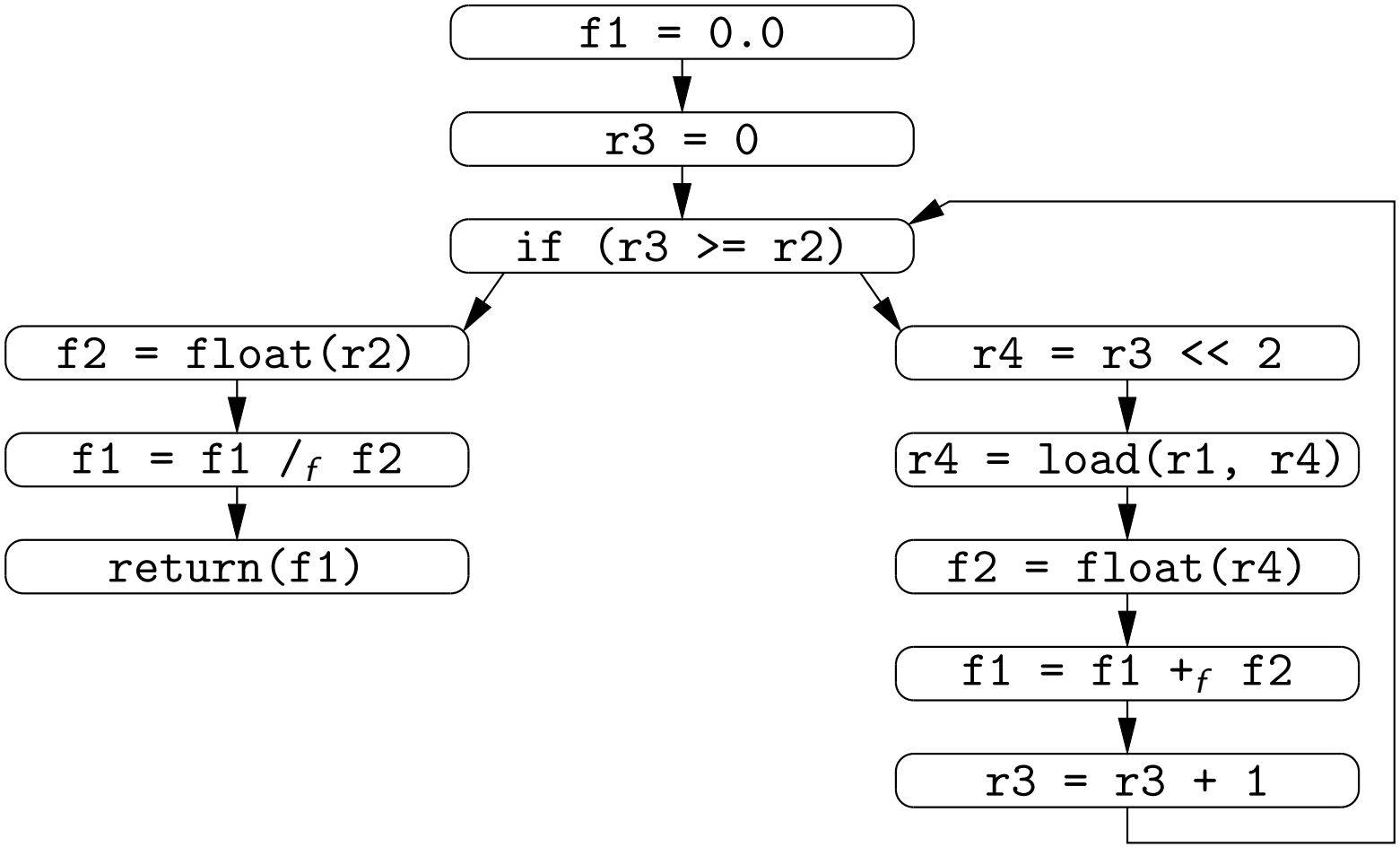
Construct a function $\phi : \mathit{Variable} \to \mathit{Register} + \mathit{Stackslot}$ such that $\phi(x) \neq \phi(y)$ if x and y interfere.

We use the Iterated Register Coalescing heuristic by George & Appel.



Algorithm 4: Rewriting the code

Replace all variables x by their color $\phi(x)$.
(Spilling & reloading are done in a later pass.)



What needs to be proved? Part 1: proofs of algorithms

Liveness analysis: show that the mapping L_{out} computed by Kildall's fixpoint 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 $(x, y) \in G \Longrightarrow \phi(x) \neq \phi(y)$

Either by direct proof (Blazy, Robillard, Appel) or by 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

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

Hmmm ...

What needs to be proved? Part 2: semantic preservation

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.

What needs to be proved? Part 2: semantic preservation

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.

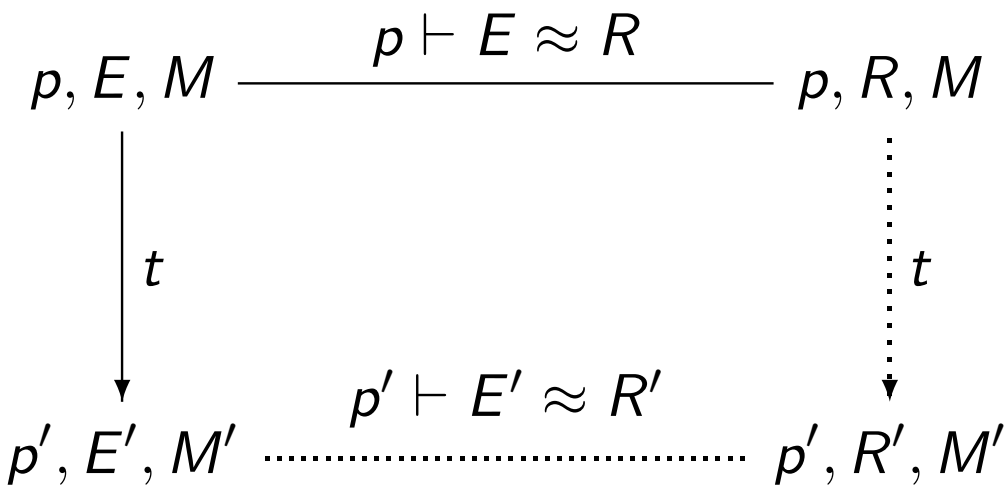
Invariant

Let E : variable $\to$ value be the values of variables at point p in the original program. Let R : location $\to$ 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 semantic preservation

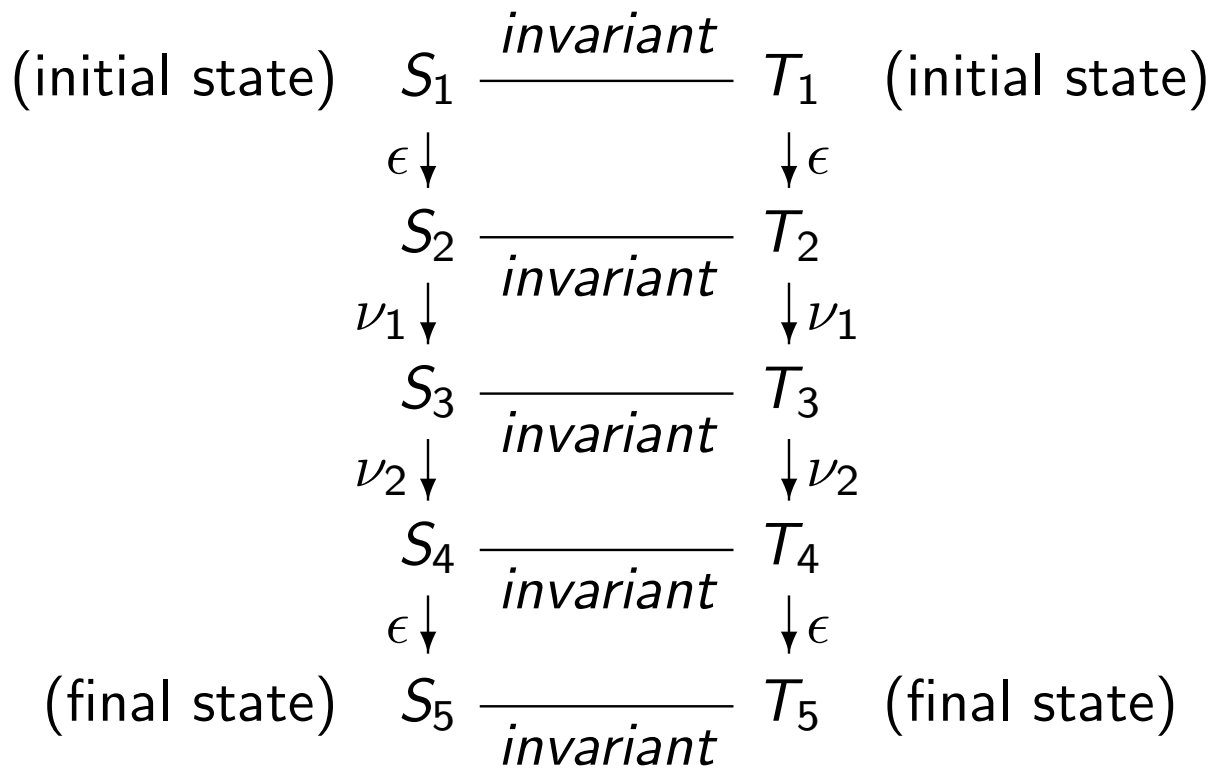
Show a simulation diagram of the form



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



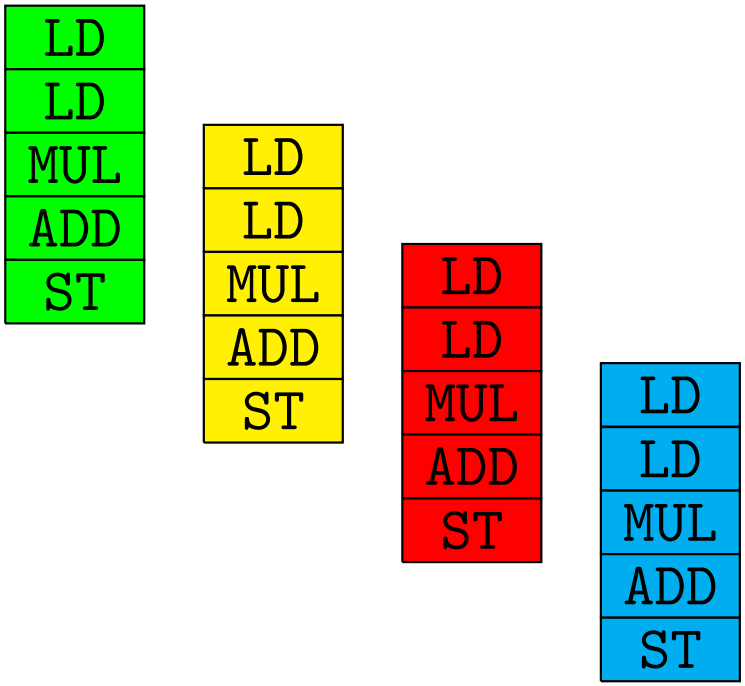
Proves that the original program and the transformed program have the same behavior (the trace $t = \nu_1.\nu_2$).

An example of verified translation validation:

software pipelining

(J.-B. Tristan and X. Leroy, *A simple, verified validator for software pipelining*, POPL 2010)

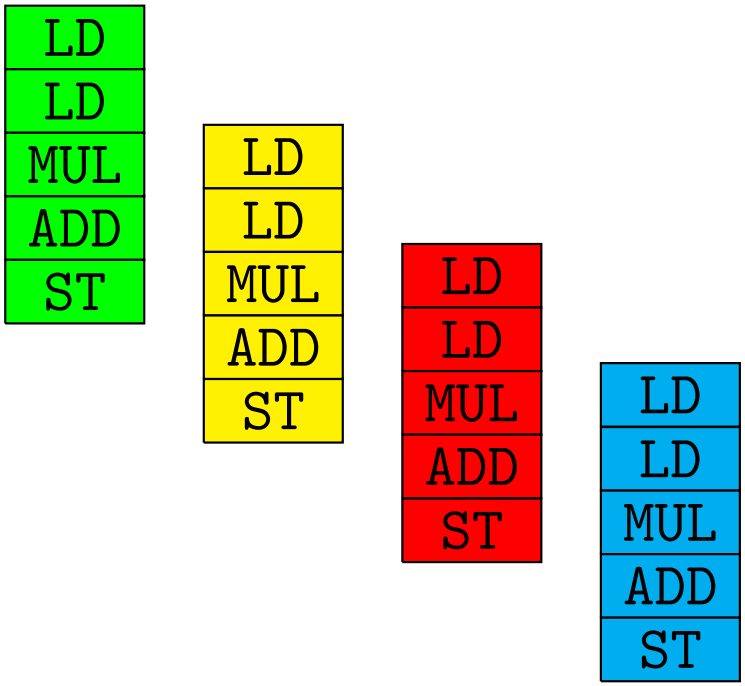
Software pipelining

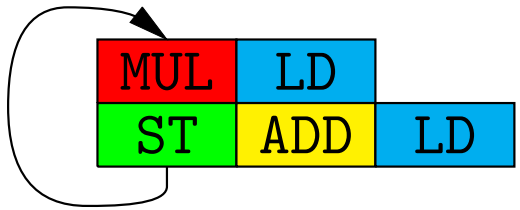


Original loop (4 iterations)

Pipelined loop

Software pipelining



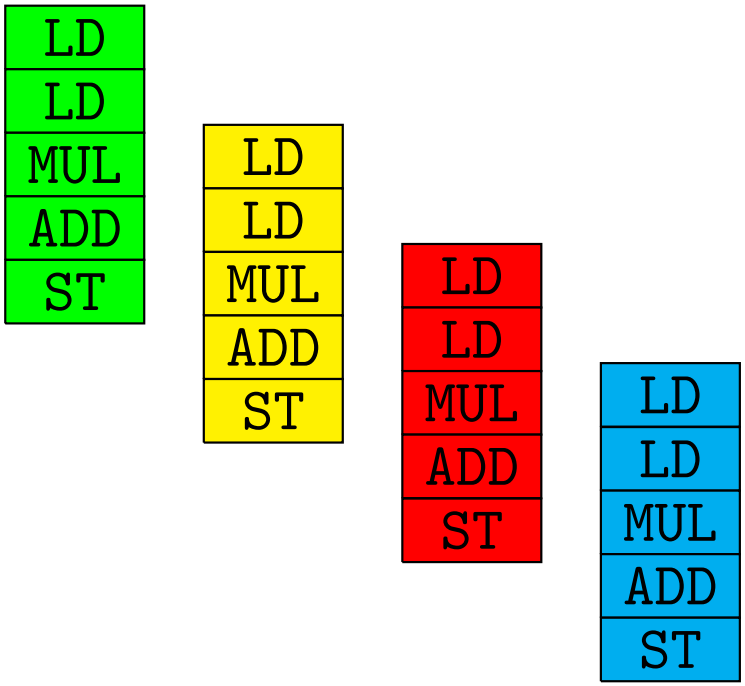


Steady state

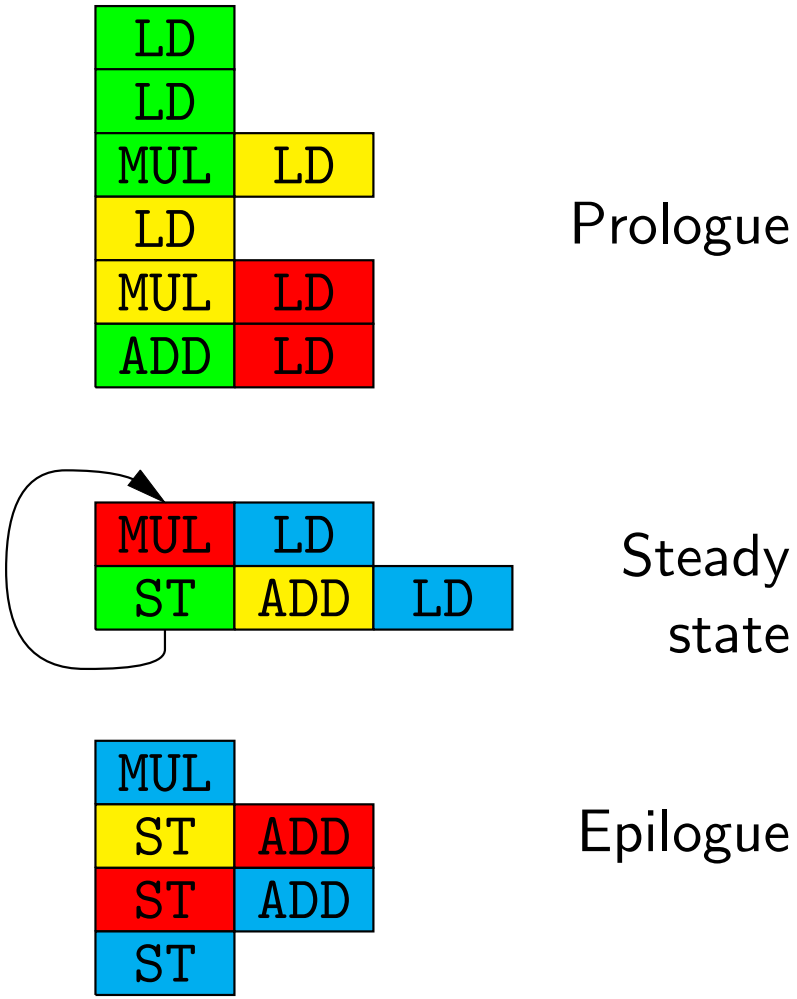
Original loop (4 iterations)

Pipelined loop

Software pipelining



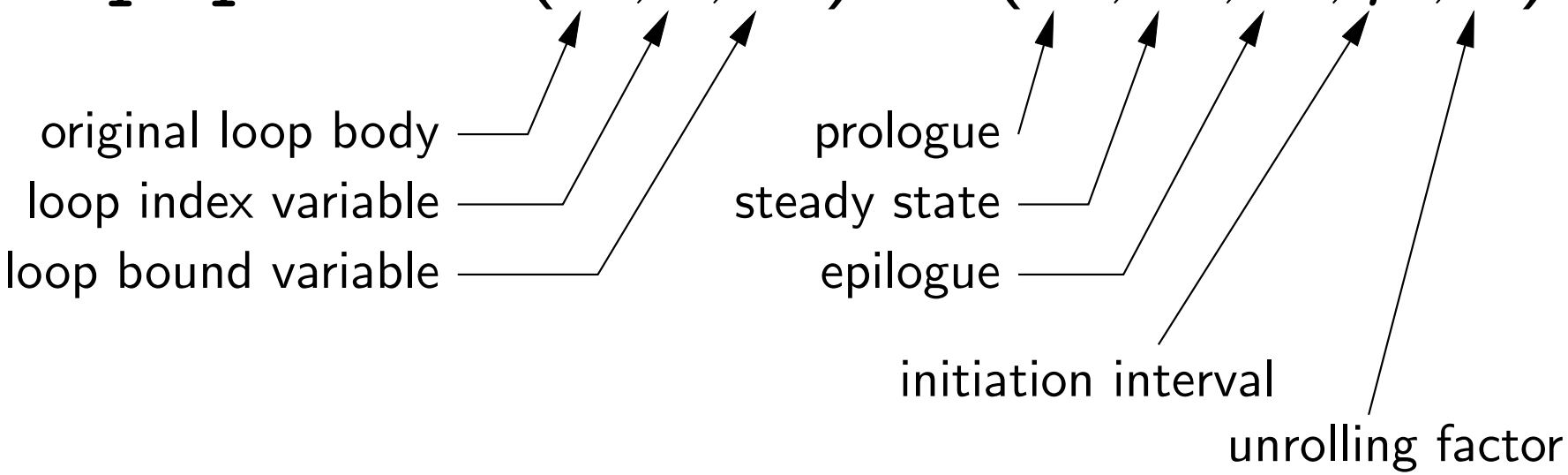
Original loop (4 iterations)



Pipelined loop

The pipeliner as a black box

$$\texttt{pipeline}(\mathcal{B}, i, N) = (\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$$

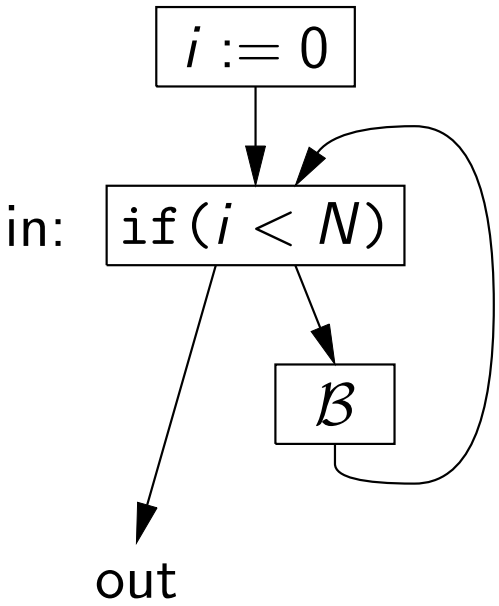


Effect on the loop variable i:

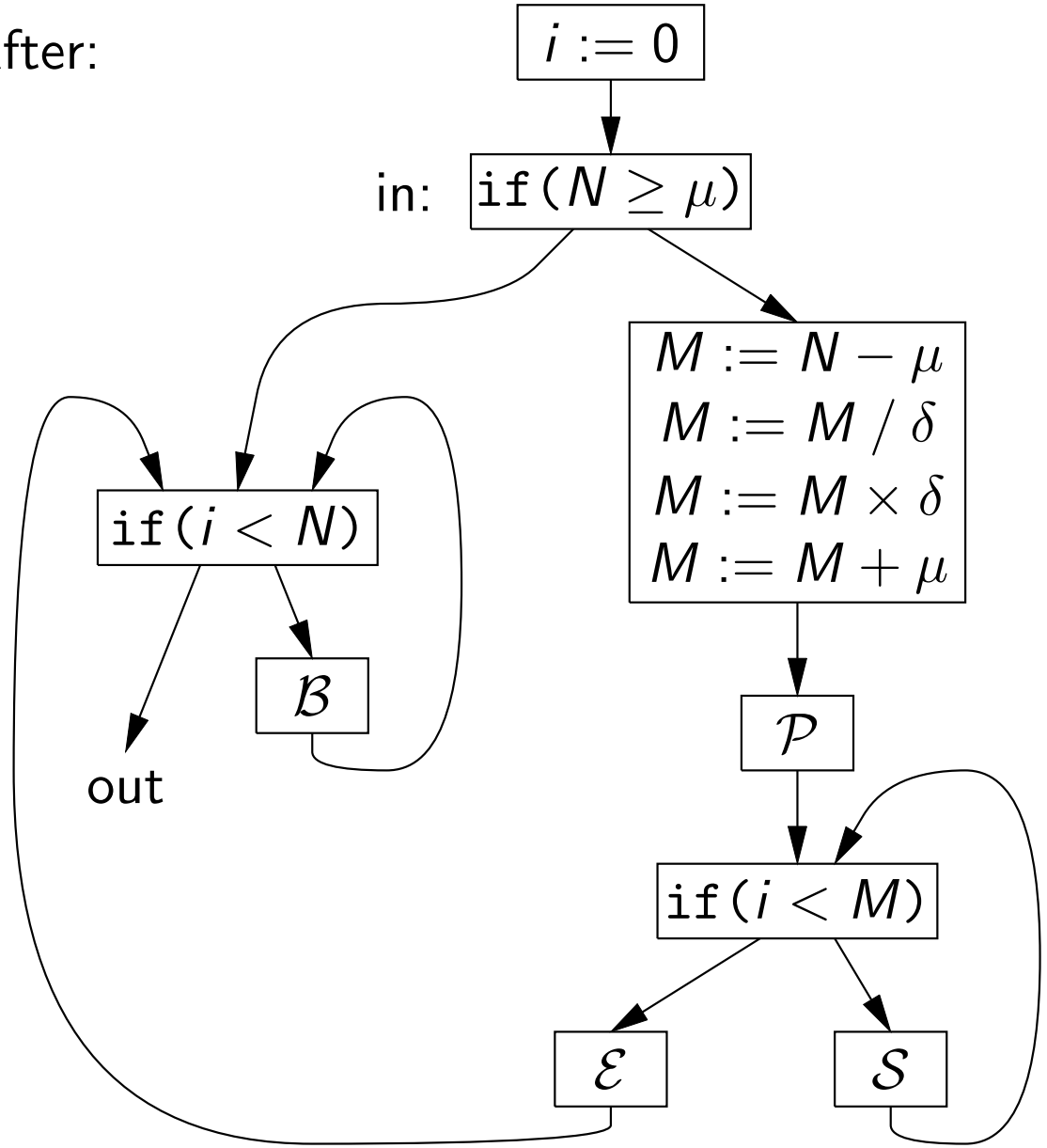
$\mathcal{B} : +1 \qquad \mathcal{P} : +\mu \qquad \mathcal{S} : +\delta \qquad \mathcal{E} : +0$

The corresponding loop transformation

Before:



After:



How to validate it?

For a fixed number of iterations $N = \mu + n \times \delta + m$, consider the unrollings of the two loops:

$$\begin{array}{lll} \text{Original loop:} & \mathcal{B}^N & (= N \text{ copies of } \mathcal{B}) \\ \text{Pipelined loop:} & \mathcal{P}; \mathcal{S}^n; \mathcal{E}; \mathcal{B}^m & \end{array}$$

If n, m are known at compile-time, we can use symbolic evaluation to show semantic equivalence between these two basic blocks.

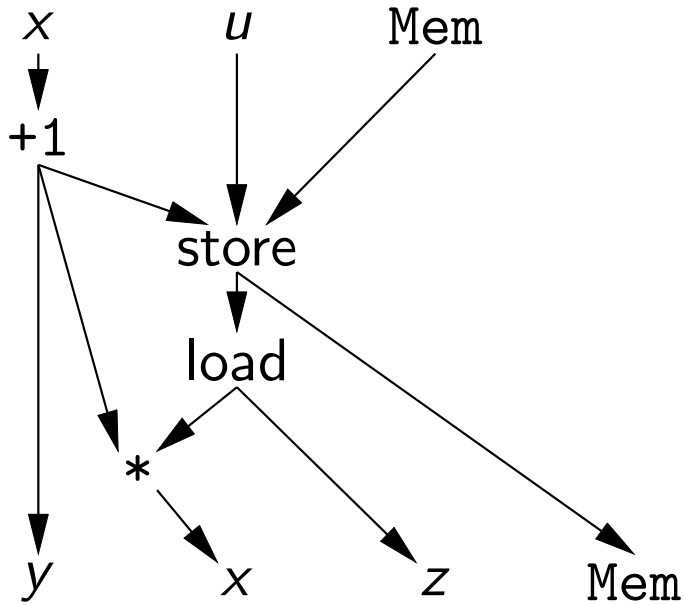
Basics of symbolic evaluation

Interpret instructions as substitutions:

$$\alpha(x := y + z) \quad = \quad x \mapsto y + z$$

Interpret basic blocks by composing substitutions:

$$\alpha \left(\begin{array}{l} y := x + 1; \\ \texttt{store}(u, y); \\ z := \texttt{load}(u); \\ x := y * z; \end{array} \right) \quad =$$



Validation by comparison of symbolic evaluations

Fundamental property: two basic blocks B_1 and B_2 that have the same symbolic evaluation ($\alpha(B_1) = \alpha(B_2)$) are semantically equivalent.

Special care must be taken for:

- Comparing modulo algebraic laws and `load`/`store` properties:

$$\begin{aligned} (e + 1) + 2 &= e + 3 \\ \texttt{load}(\texttt{store}(m, e_1, e_2), e_3) &= \texttt{load}(m, e_3) \text{ if } e_1 \neq e_3 \end{aligned}$$

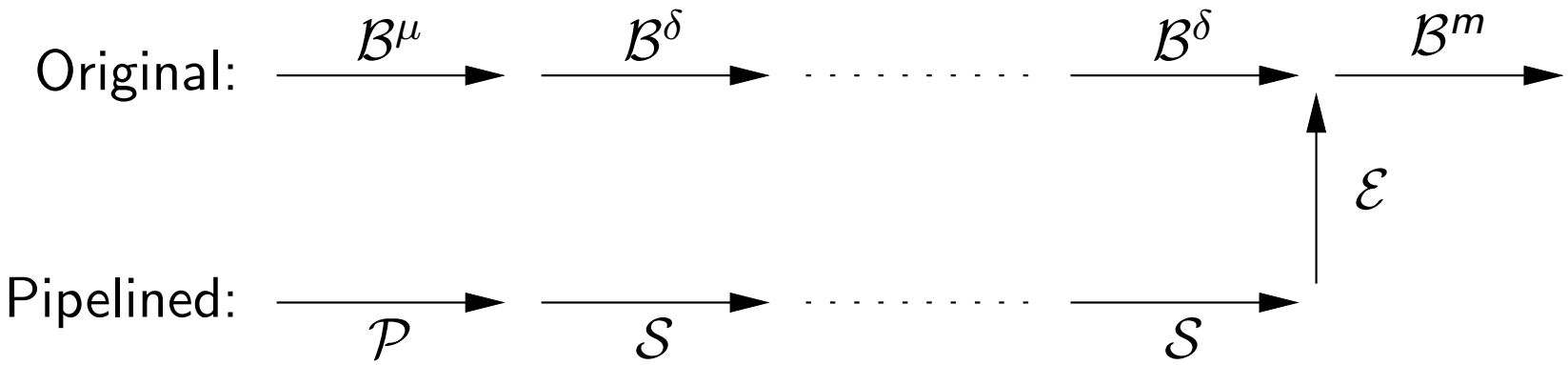
- Tracking operations that can fail at run-time (integer division).
- Excluding fresh temporaries (introduced by Modulo Variable Expansion) from the comparison.

Checking semantic equivalence of the two loops

Original loop: $\mathcal{B}^{\mu+n\times\delta+m}$

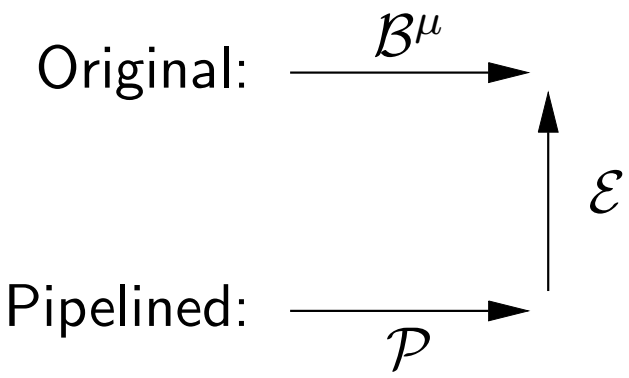
Pipelined loop: $\mathcal{P};\mathcal{S}^n;\mathcal{E};\mathcal{B}^m$

How to check semantic equivalence *for all counts* n, m ?



Checking semantic equivalence

Consider the case $n = m = 0$ and $N = \mu$:



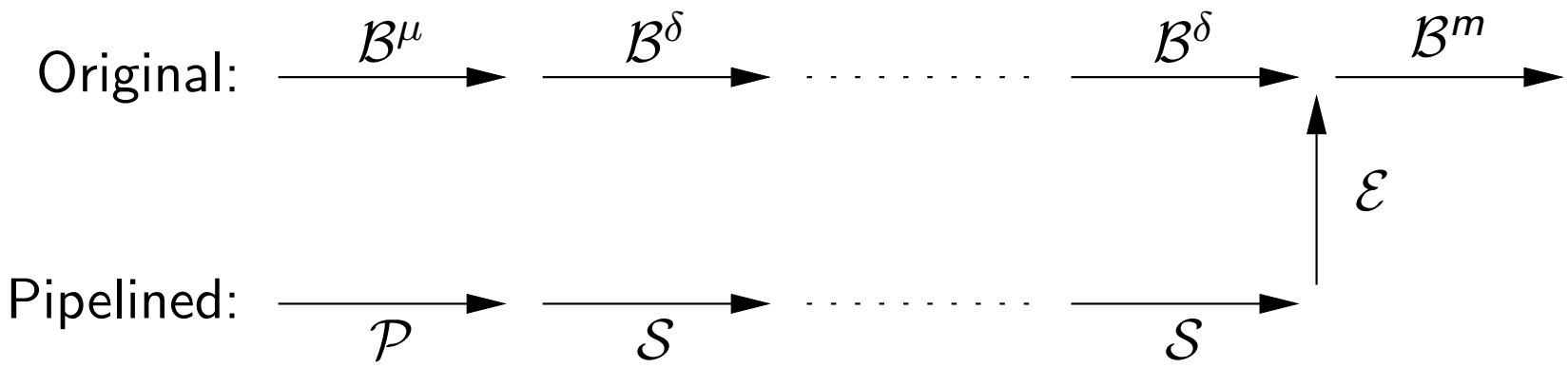
First (necessary) condition: $\mathcal{P}; \mathcal{E} \approx \mathcal{B}^\mu$

(with $\approx$ denoting semantic equivalence).

Checking semantic equivalence

What if we exit the pipelined loop one step earlier?

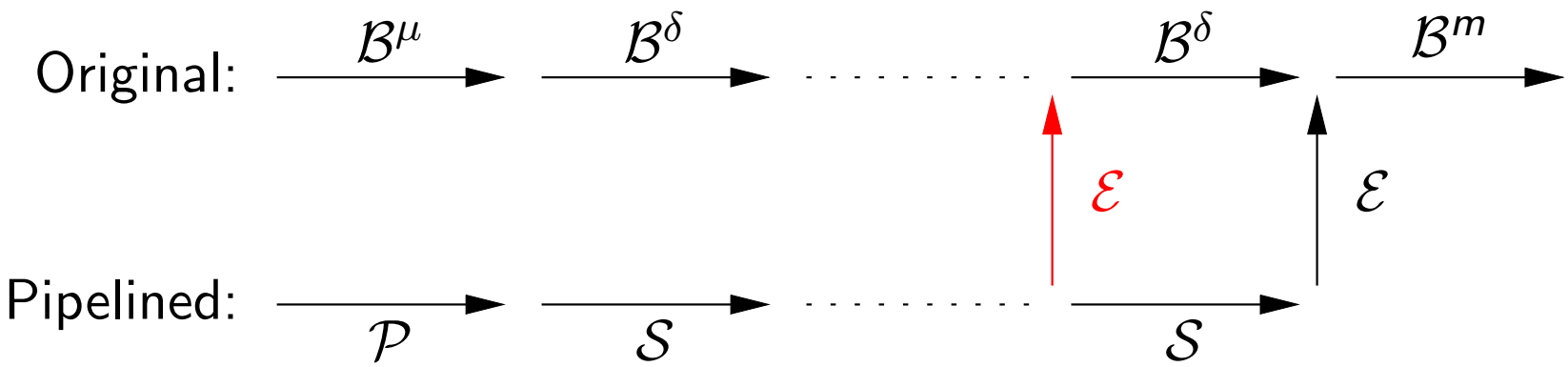
(I.e. $n \mapsto n - 1$ and $m \mapsto m + \delta$ and N unchanged.)



Checking semantic equivalence

What if we exit the pipelined loop one step earlier?

(I.e. $n \mapsto n-1$ and $m \mapsto m+\delta$ and N unchanged.)



Second condition: $\mathcal{S};\mathcal{E} \approx \mathcal{E};\mathcal{B}^{\delta}$

The validation algorithm

$$
\begin{aligned}
&\texttt{validate}\ (i, N, \mathcal{B})\ \ (\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)\ \ \theta = \\
&\qquad \alpha(\mathcal{B}^{\mu}) \approx_{\theta} \alpha(\mathcal{P}; \mathcal{E}) && \text{(A)} \\
&\quad \wedge\ \alpha(\mathcal{E}; \mathcal{B}^{\delta}) \approx_{\theta} \alpha(\mathcal{S}; \mathcal{E}) && \text{(B)} \\
&\quad \wedge\ \alpha(\mathcal{B}) \sqsubseteq \theta && \text{(C)} \\
&\quad \wedge\ \alpha(\mathcal{B})(i) = i + 1 && \text{(D)} \\
&\quad \wedge\ \alpha(\mathcal{P})(i) = i + \mu && \text{(E)} \\
&\quad \wedge\ \alpha(\mathcal{S})(i) = i + \delta && \text{(F)} \\
&\quad \wedge\ \alpha(\mathcal{E})(i) = i && \text{(G)} \\
&\quad \wedge\ \alpha(\mathcal{B})(N) = \alpha(\mathcal{P})(N) = \alpha(\mathcal{S})(N) = \alpha(\mathcal{E})(N) = N && \text{(H)}
\end{aligned}
$$

(Where θ is the set of observed variables, α is symbolic evaluation, $\approx_{\theta}$ a syntactic equivalence, and $\sqsubseteq$ a condition on free variables.)

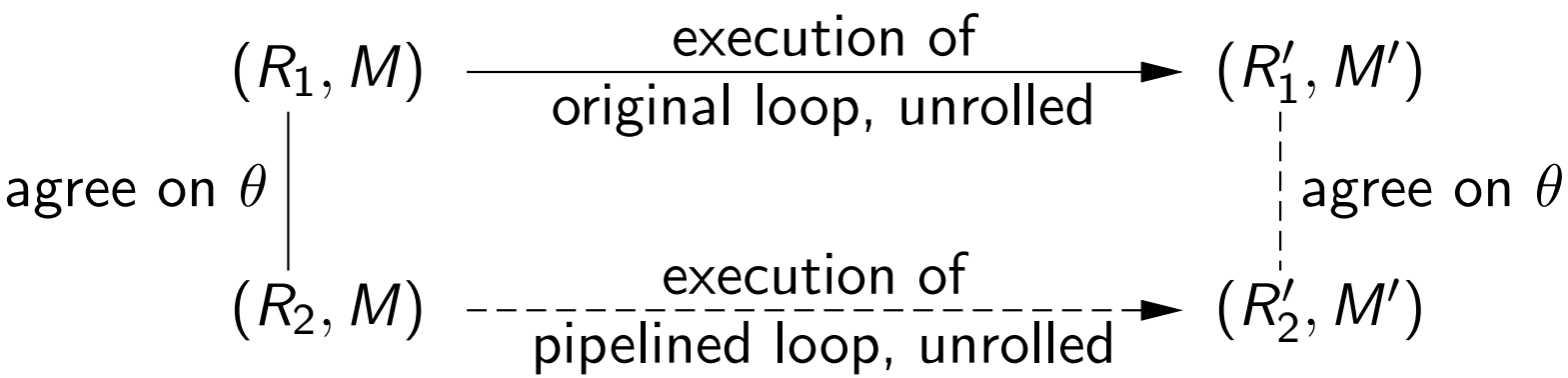
Soundness of the validator

$$\texttt{validate}\ (i, N, \mathcal{B})\ \ (\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)\ \ \theta = \texttt{true}$$

$$\Downarrow$$

$$\forall n, m,\ \ \alpha(\mathcal{B}^{\mu+\delta\times n+m}) \approx_\theta \alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E}; \mathcal{B}^m) \sqsubseteq \theta$$

$$\Downarrow$$



Summary

$$\alpha(\mathcal{B}^{\mu}) \approx_{\theta} \alpha(\mathcal{P};\mathcal{E}) \quad \wedge \quad \alpha(\mathcal{E};\mathcal{B}^{\delta}) \approx_{\theta} \alpha(\mathcal{S};\mathcal{E})$$

A surprisingly simple validator for a difficult optimization.

So simple it could go into regular, non-verified compilers.

The validator uses completely different concepts from the optimization.
(No RAW/WAR/WAW dependencies; no modulo variable expansion; etc)

The validator is incomplete in theory, but appears complete against published modulo-scheduling algorithms.

Compiler verification:

How far can we go?

Current status

At this stage of the CompCert experiment, the initial goal – proving correct a nontrivial compiler – appears feasible.

(Within the limitations of today's proof assistants such as Coq.)

This opens up many directions for future work.

Directions for future work

Higher assurance:

- Prove more of the unproved parts (e.g. parsing, bit-field emulation).
- Formal connections with verification tools.
- Formal connections with hardware (microarchitecture) verification.

Other source languages:

- Reactive languages (M. Pouzet, M. Pantel, M. Strecker).
- Functional languages (Z. Dargaye, A. Tolmach).
- Elements of C++ (T. Ramananandro, G. Dos Reis).
- Connections with run-time system verification (A. Tolmach et al).

Directions for future work

Shared-memory concurrency:

- Verified Software Toolchain (A. Appel et al).
- CompCertTSO (P. Sewell et al).

Shorter, easier proofs:

- More proof automation.
- Generic execution engine for optimizations (S. Lerner & Z. Tatlock)

More optimizations!

Done in CompCert and related experiments:

Verified transformations	Verified translation validation
Constant propagation	Lazy Code Motion
CSE over ext. basic blocks	Trace scheduling
	Software pipelining
Register allocation	Register allocation
w/ trivial spilling	w/ advanced spilling & splitting

Much remains to be done, in particular:

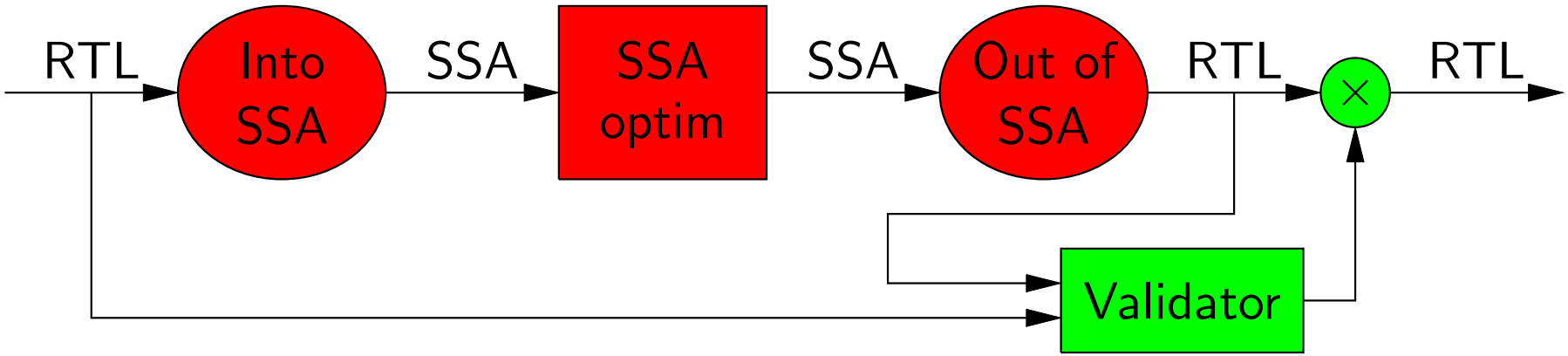
- Loop optimizations.
- SSA-based optimizations.

Verified translation validation as the path of least resistance?

SSA and what to prove about it

Plan A: formalize and reason about SSA semantics.

Plan B: use SSA in untrusted implementations; validate over RTL.

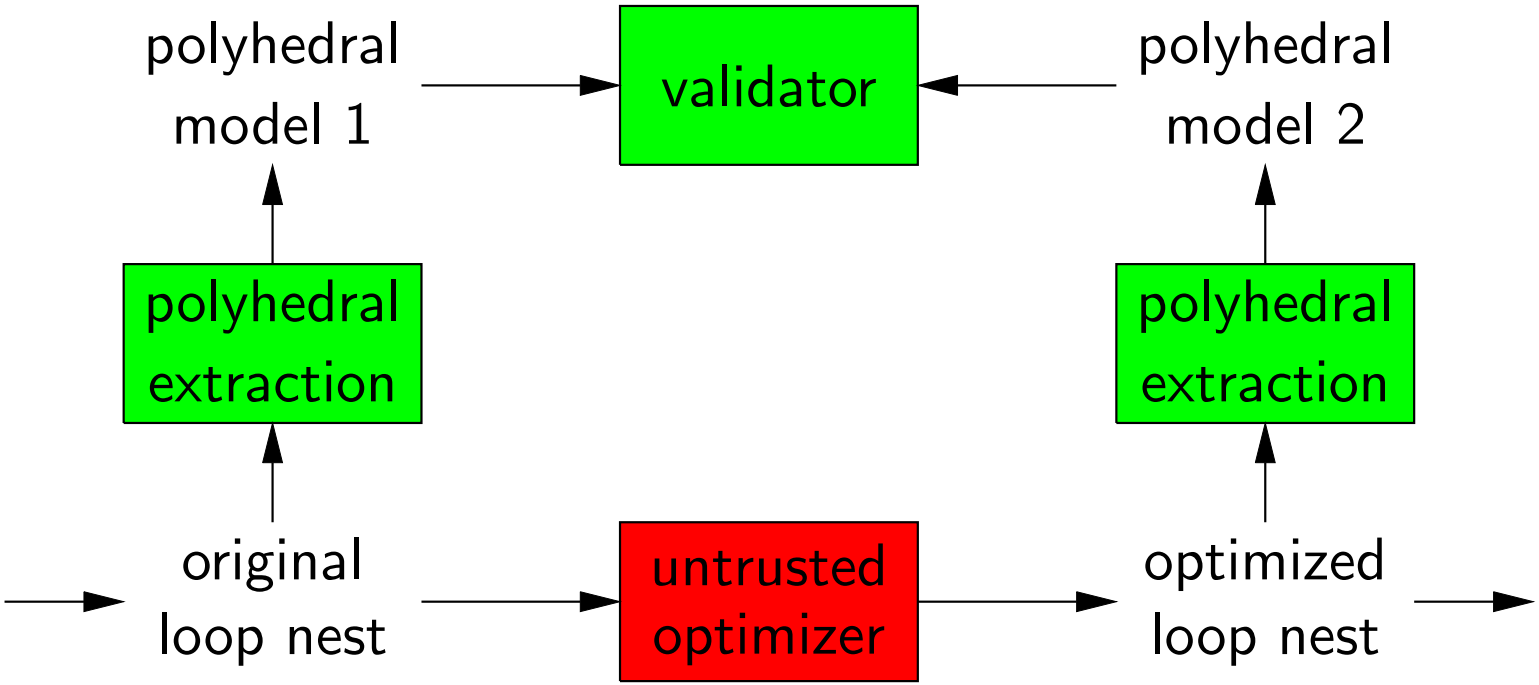


Is SSA just an algorithmic device? (faster, simpler, more powerful optims)
Or does SSA have deep semantic properties as well?

Loop optimizations

Optimizations such as loop interchange or blocking appear difficult to prove by elementary simulation arguments.

A promising approach: validation a posteriori with the polyhedral model.
(Work in progress by A. Pilkiewicz; see also Ph. Clauss et al)



In closing...

The formal verification of compilers and other development and verification tools

... is a fascinating challenge,

... appears within reach,

... and has practical importance for critical software.

Much remains to be done. *Feel free to join!*