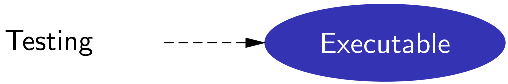
Trust in compilers, code generators, and software verification tools

Xavier Leroy

Inria Paris

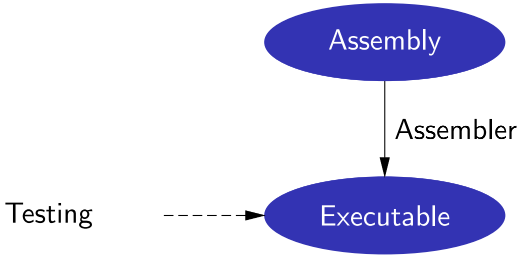
Embedded Real Time Software and Systems, 2018-02-02

Development and verification tools



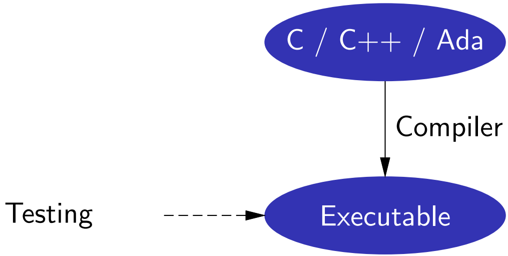
1940's: hand-written machine code

Development and verification tools



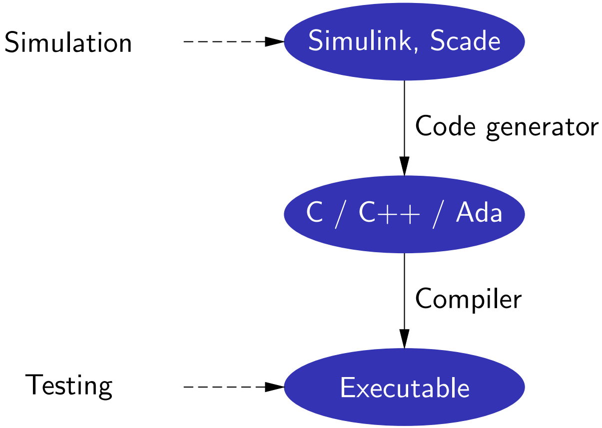
1950's: assembly languages + assemblers, linkers, autocoders

Development and verification tools



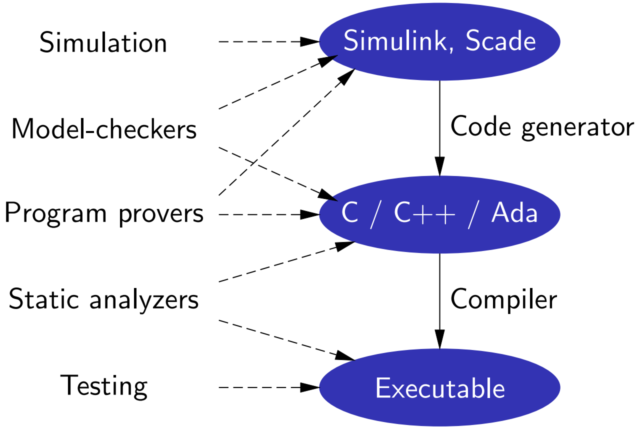
1960's: higher-level languages + compilers

Development and verification tools



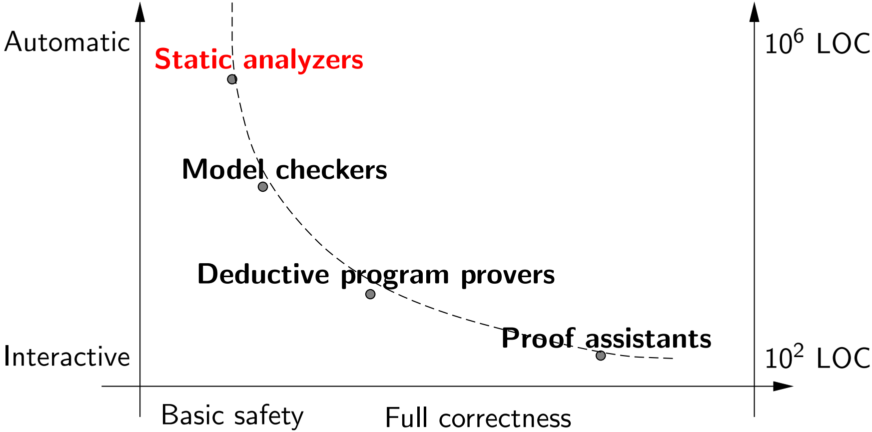
1980's: automatic code generation from models or declarative specs

Development and verification tools



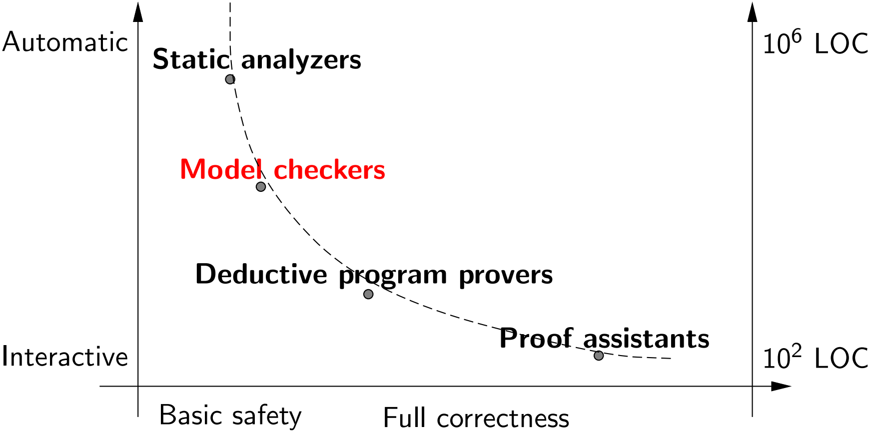
2000's: tool-assisted formal verification

A panorama of verification tools



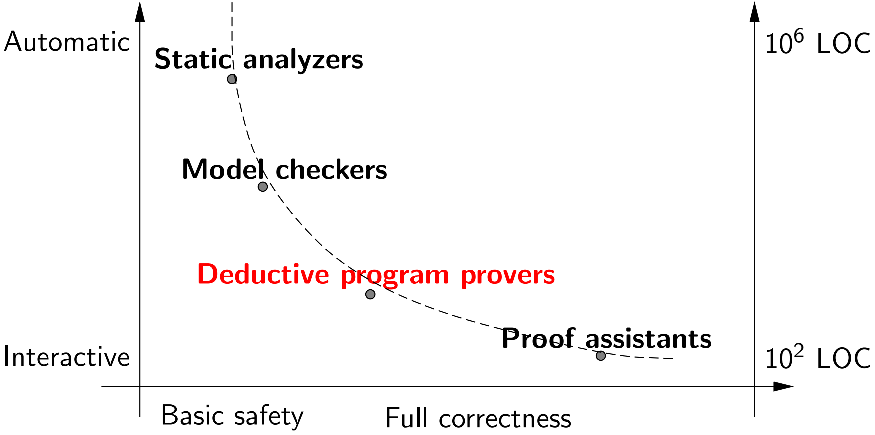
Static analysis: automatically infer simple properties of one variable ($x \in [N_1, N_2]$, p points to a, etc) or several ($x + y \leq z$).

A panorama of verification tools



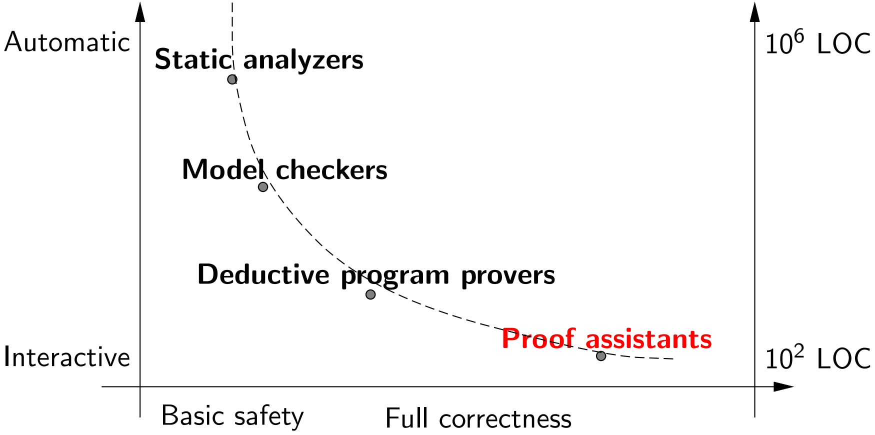
Model checking: automatically check that some “bad” program points are not reachable.

A panorama of verification tools



Program proof (Hoare logic, separation logic): show that
preconditions $\Rightarrow$ *invariants* $\Rightarrow$ *postconditions*
using automated theorem provers.

A panorama of verification tools



Proof assistants: conduct mathematical proofs in interaction with the user; re-check the proofs for correctness.

The long road to formal verification

From very early intuitions that there is something to be proved about computer programs. . .

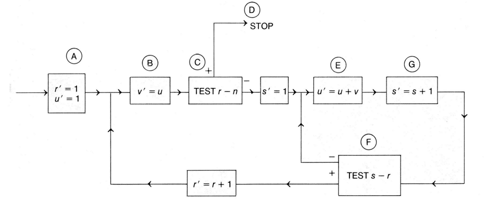


Figure 1 (Redrawn from Turing's original)

STORAGE LOCATION	(INITIAL) A $k = 6$	B $k = 5$	C $k = 4$	(STOP) D $k = 0$	E $k = 3$	F $k = 1$	G $k = 2$
27					s	$s + 1$	s
28		r	r		r	r	r
29	n	n	n	n	n	n	n
30		$\lfloor r$	$\lfloor r$		$s\lfloor r$	$(s + 1)\lfloor r$	$(s + 1)\lfloor r$
31			$\lfloor r$	$\lfloor n$	$\lfloor r$	$\lfloor r$	$\lfloor r$
	TO B WITH $r' = 1$ $u' = 1$	TO C	TO D IF $r = n$ TO E IF $r < n$		TO G	TO B WITH $r' = r + 1$ IF $s \geq r$ TO E WITH $s' = s + 1$ IF $s < r$	TO F

Figure 2 (Redrawn from Turing's original)

Alan Turing, *Checking a large routine*, 1949.

(The first known example of loop invariants.)

The long road to formal verification

From very early intuitions that there is something to be proved about computer programs...

... to fundamental formalisms ...

- Deductive verification: Floyd 1967, Hoare 1969.
- Abstract interpretation: Cousot & Cousot, 1977.
- Model checking: Clarke & Emerson, 1980; Queille & Sifakis, 1982.

The long road to formal verification

From very early intuitions that there is something to be proved about computer programs...

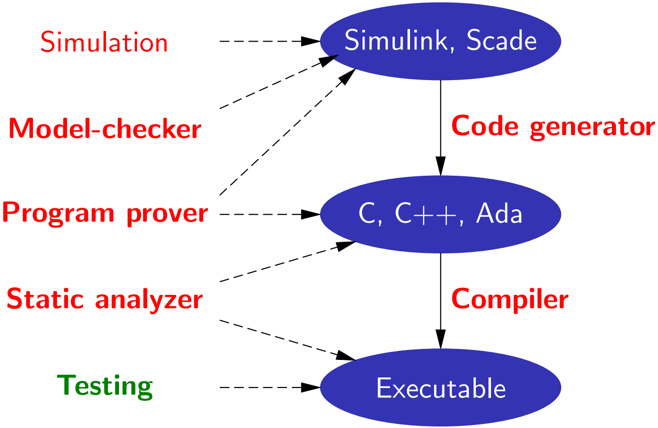
... to fundamental formalisms ...

... to verification tools that automate these ideas ...

... to actual use in the critical software industry. (50 years later)

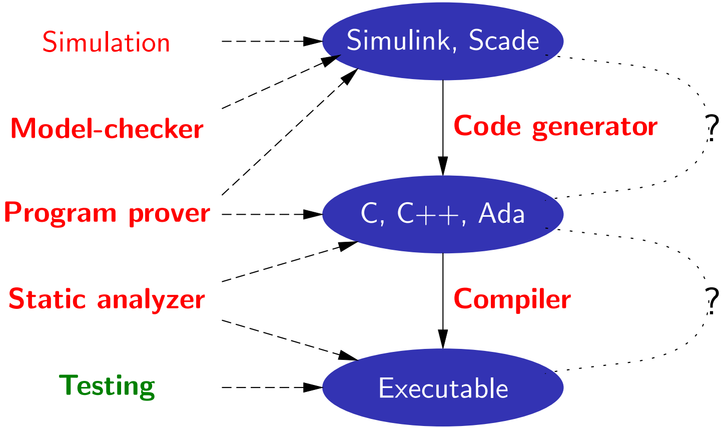
Trust in the development tools

Tool-related risks



The unsoundness risk: a verification tool could fail to account for all possible run-time states of the program, giving a false sense of safety.

Tool-related risks



The miscompilation risk: a compiler could generate bad code from a correct source program, invalidating all guarantees obtained by source-level formal verification.

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

Why is it so hard to compile and analyze correctly?

- Algorithmic complexity of compilers and analyzers.
  Ambitious optimizations; complex abstractions;
  SAT and SMT solving; etc.

Why is it so hard to compile and analyze correctly?

- Algorithmic complexity of compilers and analyzers.
  Ambitious optimizations; complex abstractions;
  SAT and SMT solving; etc.

- Structural complexity of input data (= arbitrary programs).
  Some test suites for compilers, low coverage.
  No test suites of wrong programs for analyzers.
  Random differential testing as a substitute for proper tests.

Why is it so hard to compile and analyze correctly?

- Algorithmic complexity of compilers and analyzers.
  Ambitious optimizations; complex abstractions;
  SAT and SMT solving; etc.

- Structural complexity of input data (= arbitrary programs).
  Some test suites for compilers, low coverage.
  No test suites of wrong programs for analyzers.
  Random differential testing as a substitute for proper tests.

- Misunderstandings in the definition of the source language.
  Esp. the C and C++ standards, which have many subtle points and 200+ undefined behaviors.
  Sometimes compiler writer misreads the standards.
  More often, an undefined behavior is compiled differently from expected by the programmer.

Silly compiler bugs

[Our] new method succeeded in finding bugs in the latter five (newer) versions of GCCs, in which the previous method detected no errors.

```
int main (void)
{
  unsigned x = 2U;
  unsigned t = ((unsigned) -(x/2)) / 2;
  assert ( t != 2147483647 );
}
```

It turned out that [the program above] caused the same error on the GCCs of versions from at least 3.1.0 through 4.7.2, regardless of targets and optimization options.

E. Nagai, A. Hashimoto, N. Ishiura, SASIMI 2013

Misunderstandings: 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!");
}

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: ISO C allows intermediate results to be computed with excess precision, but requires them to be rounded at assignments.

Misunderstandings: 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.

Misunderstandings: a Linux bug

```
struct sock *sk = tun->sk;
if (tun == NULL)
    return POLLERR;
/* write to address based on tun */
```

GCC removes the `tun == NULL` safety check, reasoning that if `tun` is `NULL` the memory access `tun->sk` is undefined behavior.

However, this code runs in the kernel, and the read `tun->sk` can succeed (without a kernel panic) even if `tun` is `NULL`.

Removing the `tun == NULL` check therefore opens an exploitable security hole, CVE-2009-1897.

Formal verification of tools

Testing tools to a high level of confidence is hard. Why not formally verify the compiler and the verification tools themselves? (using program proof)

After all, these tools have simple specifications:

> *Correct compiler: if compilation succeeds, the generated code behaves as prescribed by the semantics of the source program.*
>
> *Sound verification tool: if the tool reports no alarms, all executions of the source program satisfy a given safety property.*

As a corollary, we obtain:

> *The generated code satisfies the given safety property.*

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.

CompCert:
a formally-verified C compiler

The CompCert project

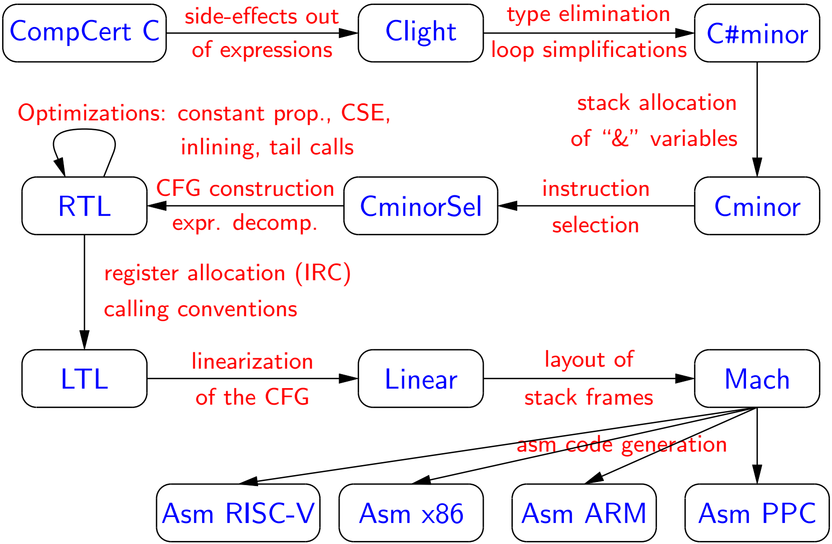
(X.Leroy, S.Blazy, et al + AbsInt Gmbh)

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

- Source language: a very large subset of C 99.
- Target language: PowerPC/ARM/RISC-V/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 formally verified part of the compiler



Formally verified using Coq

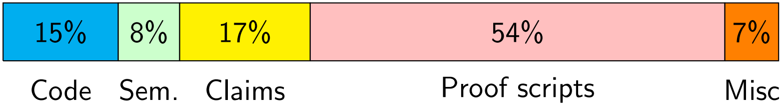
The correctness proof (semantic preservation) for the compiler is entirely machine-checked, using the Coq proof assistant.

```
Theorem transf_c_program_preservation:
  forall (p: Csyntax.program) (tp: Asm.program) (b: behavior),
  transf_c_program p = OK tp ->
  program_behaves (Asm.semantics tp) b ->
  exists b', program_behaves (Csem.semantics p) b'
             /\ behavior_improves b' b.
```

Shows refinement of observable behaviors `b`:

- Reduction of internal nondeterminism
  (e.g. choose one evaluation order among the several allowed by C)
- Replacement of run-time errors by more defined behaviors
  (e.g. optimize away a division by zero)

Proof effort



100,000 lines of Coq.

Including 15000 lines of “source code” ($\approx$ 60,000 lines of Java).

6 person.years

Low proof automation (could be improved).

Programmed (mostly) in Coq

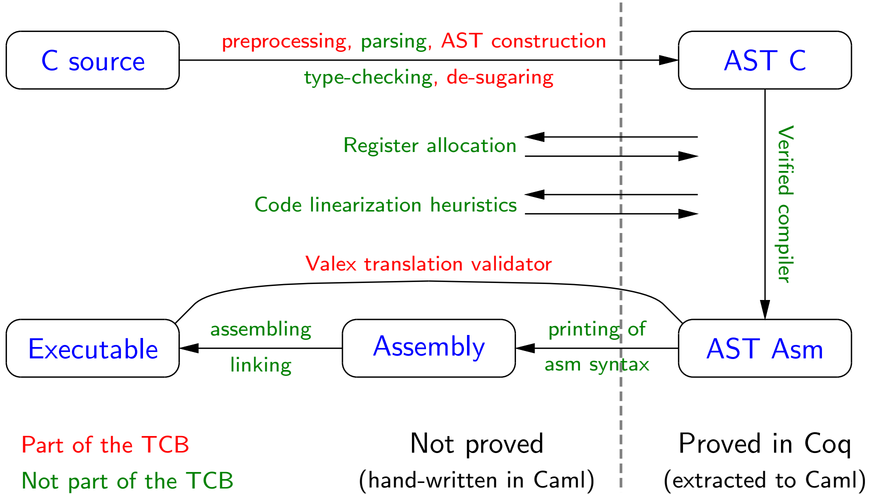
All the verified parts of the compiler are programmed directly in Coq's specification language, using pure functional style.

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

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

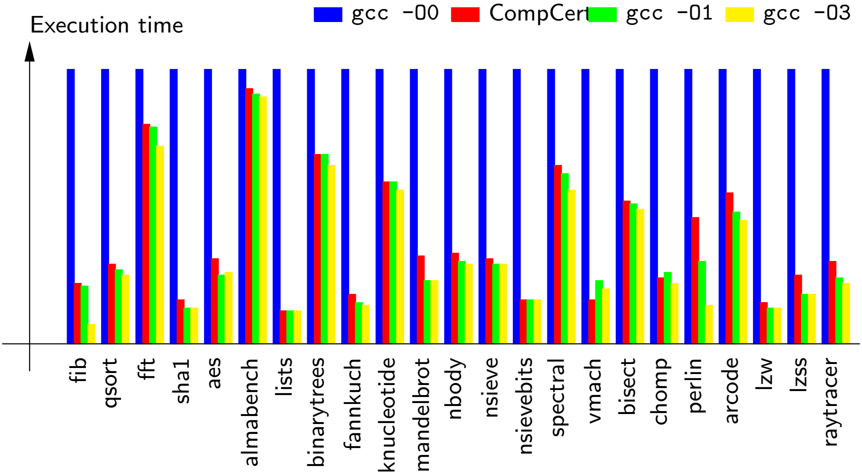
Claim: purely functional programming is the shortest path to writing and proving a program.

The whole Compcert compiler

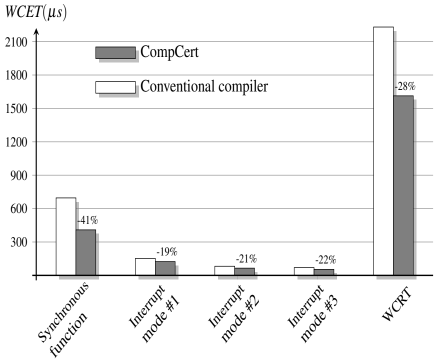


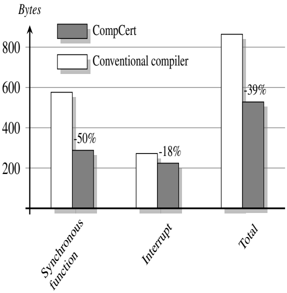
Performance of generated code

(On a Power 7 processor)



WCET and stack use improvements on a real-time application





Daniel Kästner et al, *CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler*, ERTS 2018, session Fr1B.

Verasco:
a formally-verified C static analyzer

The Verasco project

J.H. Jourdan, V. Laporte, et al

Goal: develop and verify in Coq a realistic static analyzer by abstract interpretation:

- Language analyzed: the CompCert subset of C.
- Property established: absence of run-time errors
  (out-of-bound array accesses, null pointer dereferences, division by zero, etc).
- Nontrivial abstract domains, including relational domains.
- Modular architecture inspired from Astrée's.
- Decent (but not great) alarm reporting.

Properties inferred by Verasco

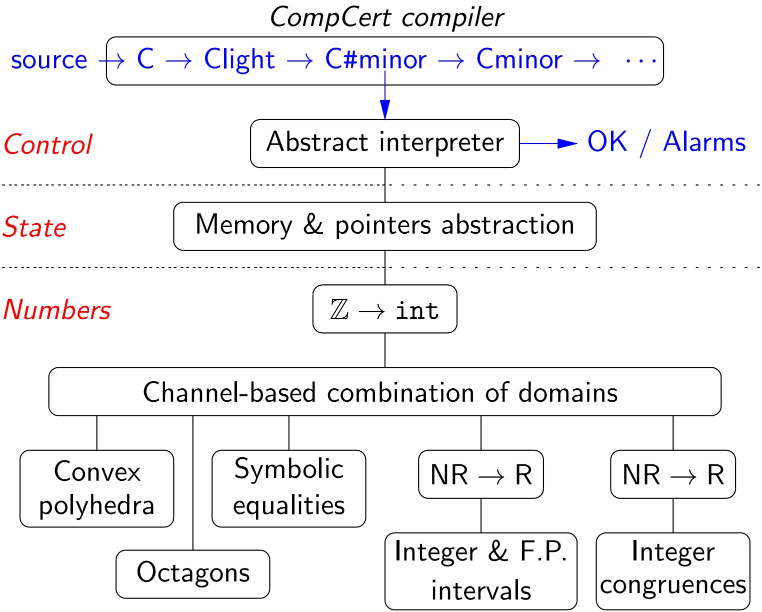
Properties of a single variable / memory cell: (value analysis)

Variation intervals	$x \in [c_1; c_2]$
Integer congruences	$x \bmod c_1 = c_2$
Points-to and nonaliasing	$p\ \texttt{poinsTo}\ \{x_1, \ldots, x_n\}$

Relations between variables: (relational analysis)

Polyhedra	$c_1 x_1 + \cdots + c_n x_n \leq c$
Octagons	$\pm x_1 \pm x_2 \leq c$
Symbolic equalities	$x = \mathrm{expr}$

Architecture



Proof methodology

The abstract interpretation framework, with some simplifications:

- Only prove the soundness of the analyzer, using the γ half of Galois connections:

$$\gamma : \text{abstract object} \rightarrow \wp(\text{concrete things})$$

- Don't prove relative optimality of abstractions (the α half of Galois connections).
- Don't prove termination of the analyzer.

Status of Verasco

It works!

- Fully proved (46 000 lines of Coq)
- Executable analyzer obtained by extraction.
- Able to show absence of run-time errors in small but nontrivial C programs.

It needs improving!

- Some loops need full unrolling
  (to show that an array is fully initialized at the end of a loop).
- Analysis is slow (e.g. 10 sec for 100 LOC).

Perspectives

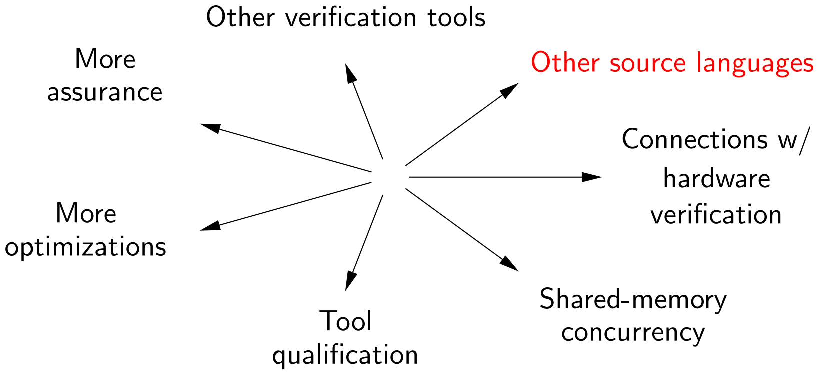
Current status

Formal verification of development tools is just starting:

- CompCert is entering industrial use, with commercial support from AbsInt;
- Verasco is still at the advanced prototype stage.

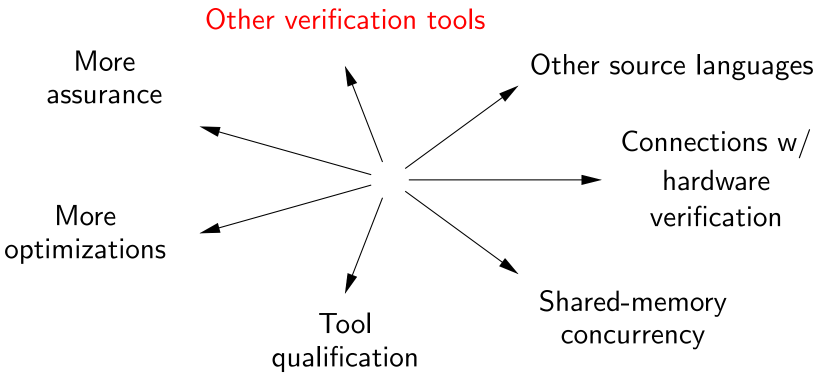
However, these projects demonstrate that the formal verification of compilers, static analyzers, and related tools is feasible.
(Within the limitations of today's proof assistants.)

Future directions



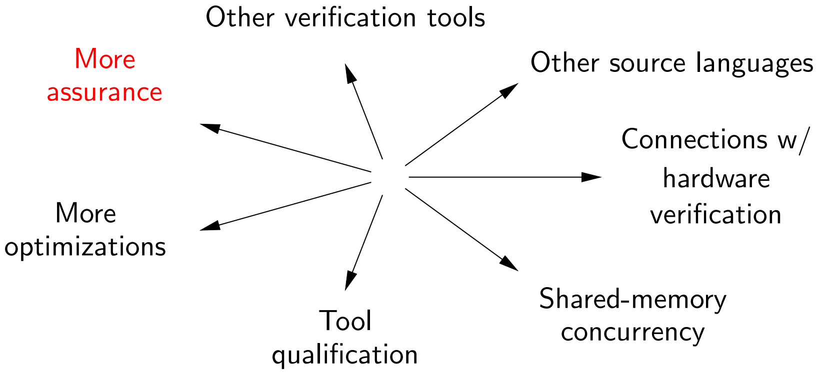
Other source languages besides C:
reactive languages (Velus project), functional languages, Rust, ...

Future directions



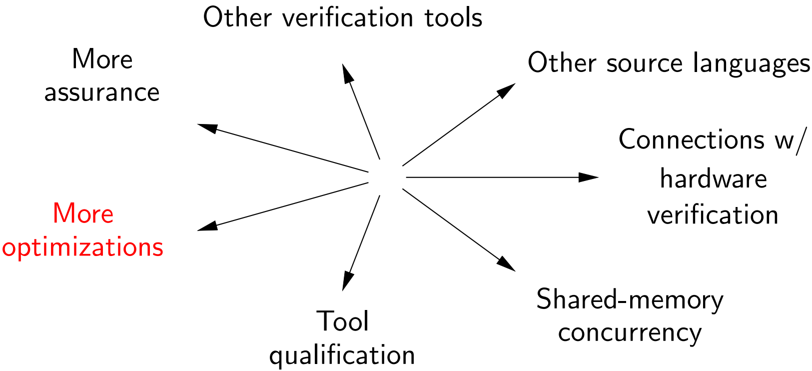
Other verification tools besides static analyzers:
program provers, model checkers, SAT and SMT solvers

Future directions



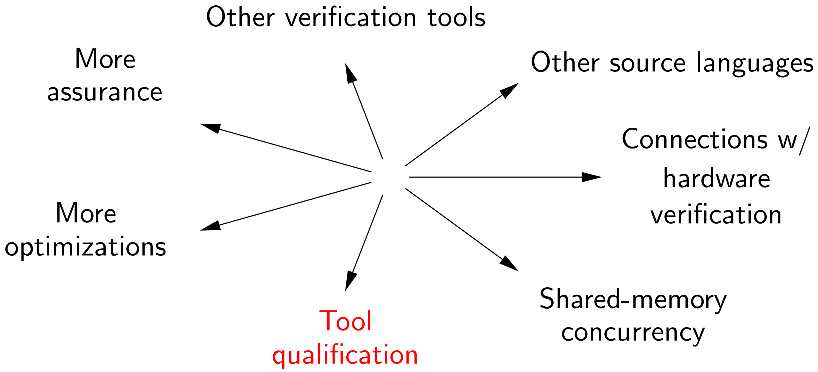
Prove or validate more of the TCB:
preprocessing, elaboration, . . .

Future directions



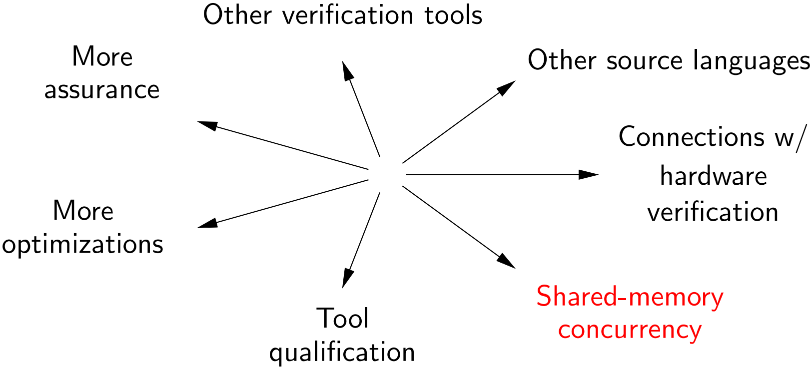
Add advanced optimizations, esp. loop optimizations.

Future directions



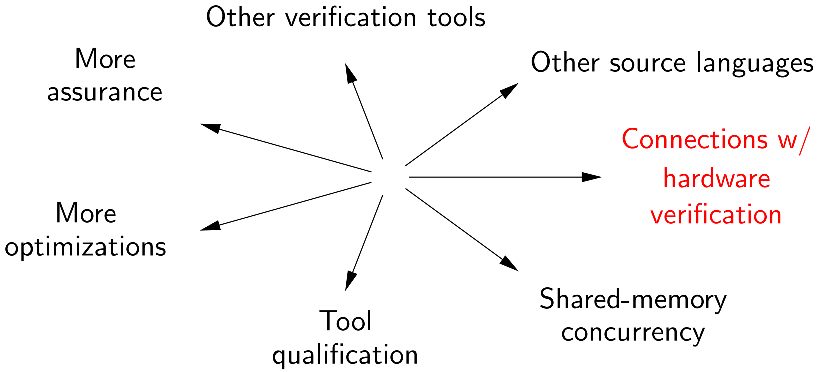
How to leverage Coq proofs in a tool qualification?
See Kästner et al in session Fr1B for a IEC 60880 certification involving CompCert.

Future directions



Race-free programs + concurrent separation logic
or: racy programs + hardware memory models a la C++11

Future directions



Formal specs for architectures & instruction sets, as the missing link between compiler verification and hardware verification.

In closing...

Critical software deserves the most trustworthy tools
that computer science can produce.

Let's make this a reality!