Tools you can trust: formal verification of compilers and static analyzers

Xavier Leroy

Inria Paris

Inria/EPFL workshop, 13 January 2016

Plan

1. The distance between verified model and actual executable
2. CompCert: formal verification of a C compiler
3. Verasco: formal verification of a C static analyzer
4. Future directions

Tool-assisted formal verification

Old, fundamental ideas...

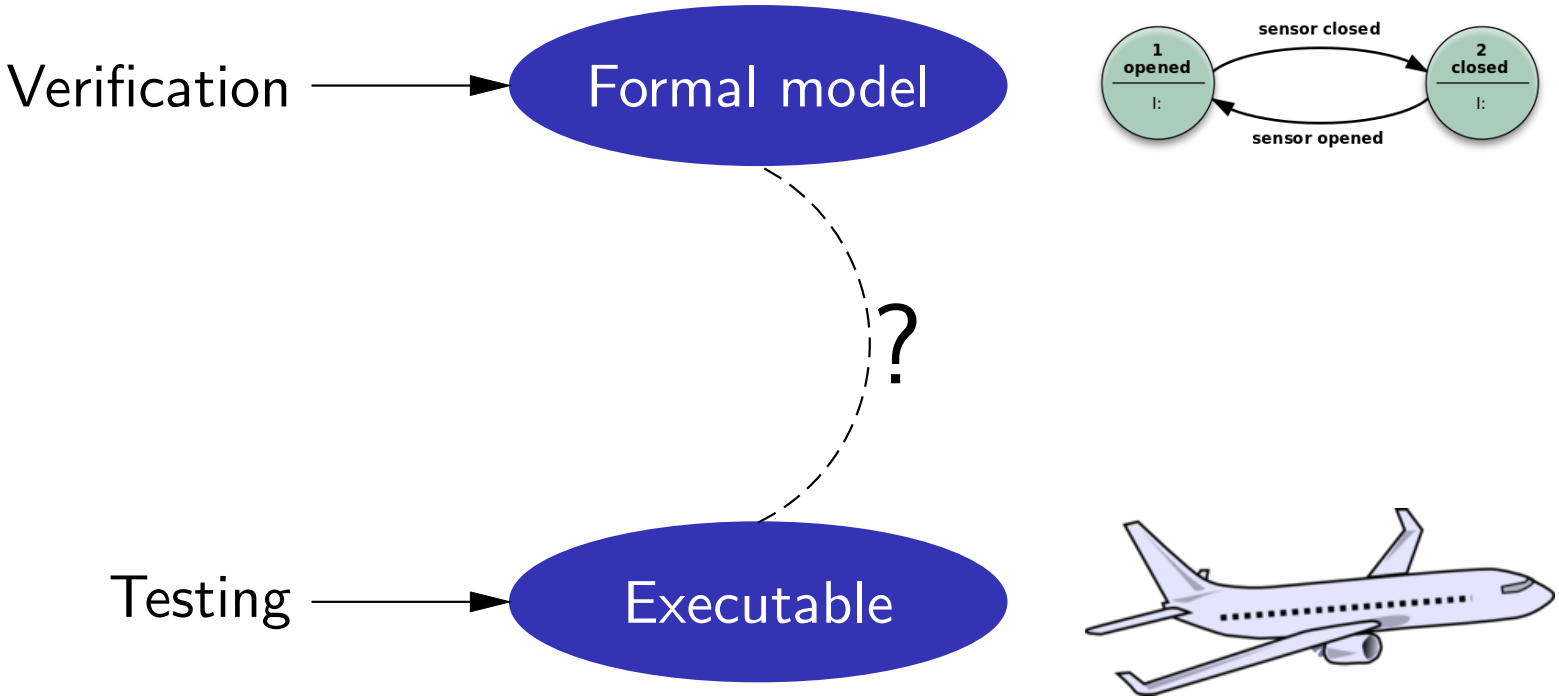
(Hoare logic, 1960's; model checking, abstract interpretation, 1970's)

that remained purely theoretical for a long time...

are now implemented and automated in verification tools...

usable and sometimes used in the critical software industry.

The verification gap



What is formally verified is not what actually executes.
(Unlike with testing.)

What can go wrong?

Programming errors when manually implementing the model as code.

- E.g. TLS security holes: 50% are protocol bugs, 50% implementation mistakes.

Modeling / abstraction errors

- The model over-simplifies the code.
- Not all behaviors are accounted for.

Reducing the gap

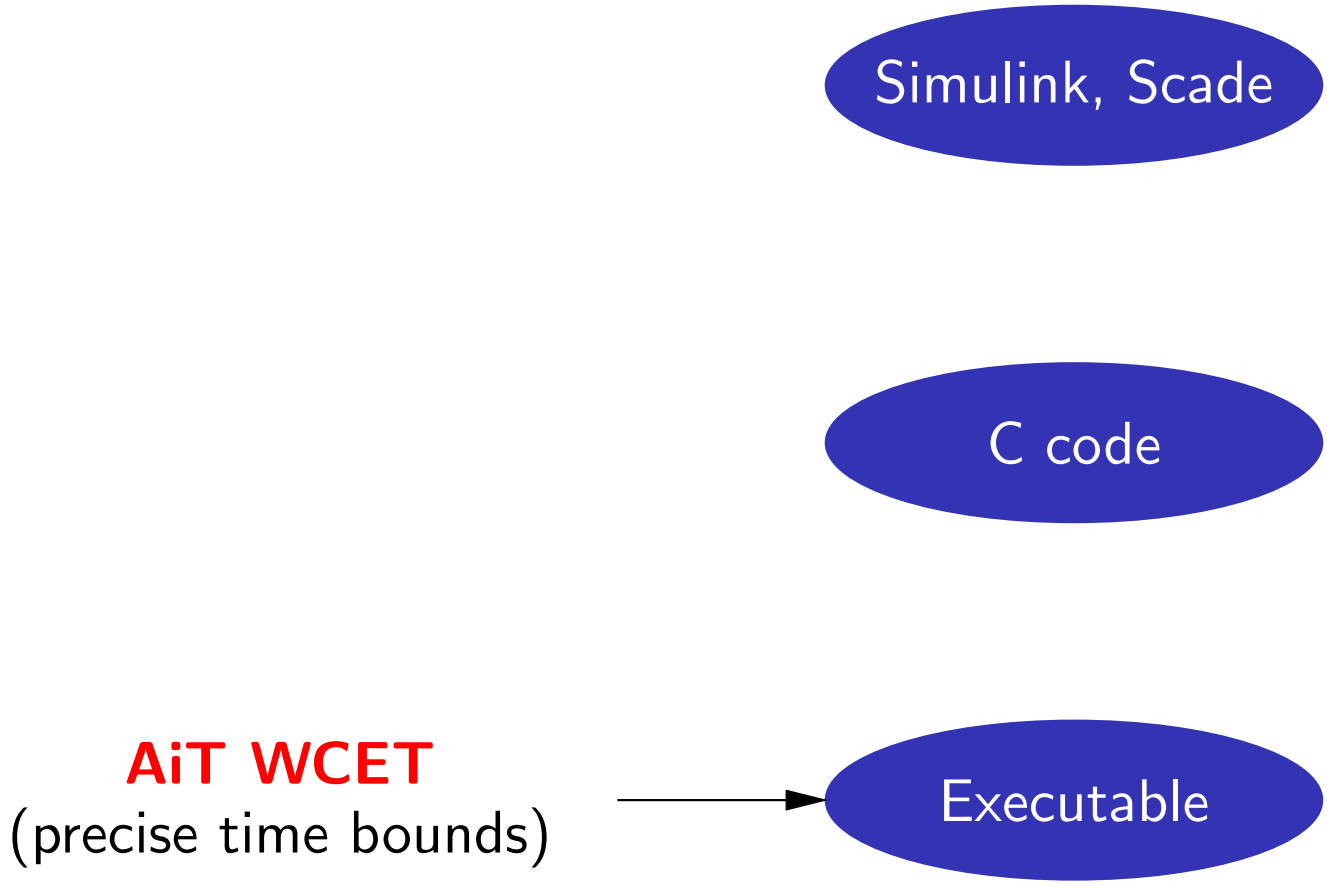
Higher-level programming languages ("the program is the model"):

- Domain-specific languages, e.g. Simulink, Scade for control-command.
- Declarative programming: functional, constraint-based.

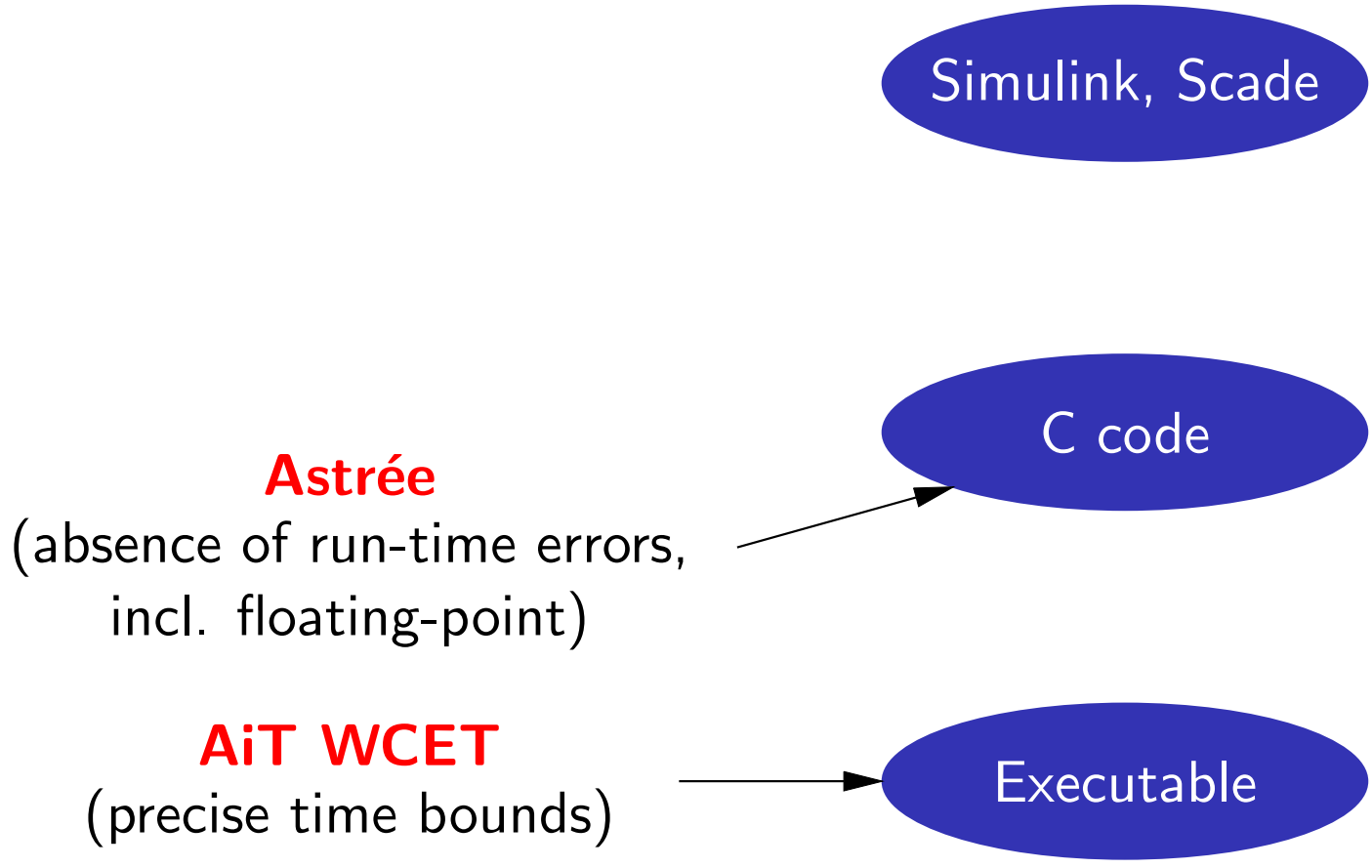
Verification at a lower level:

- Source code level: Java, C, ...
- Or even on the machine code of the actual executable!

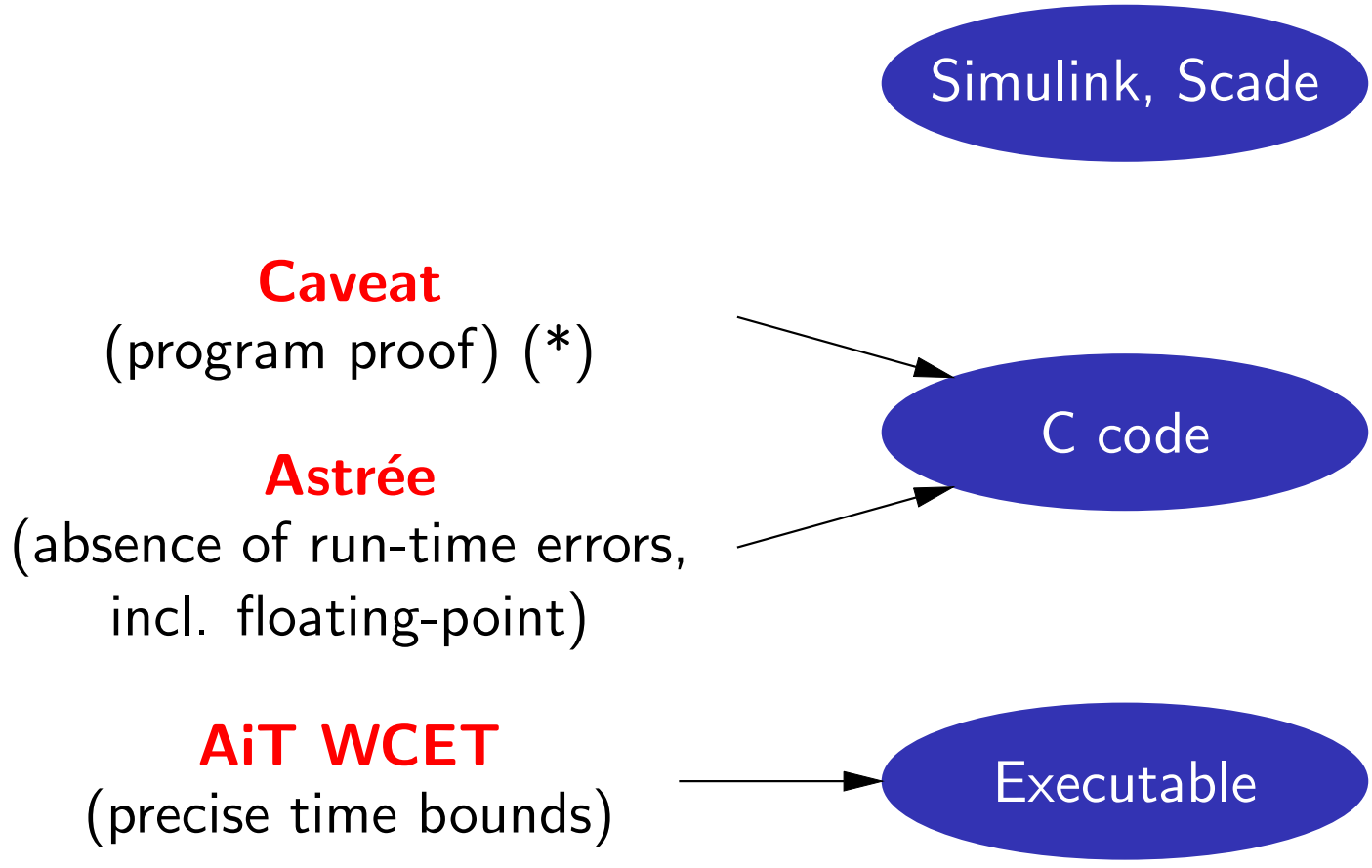
Examples for the aircraft industry



Examples for the aircraft industry



Examples for the aircraft industry



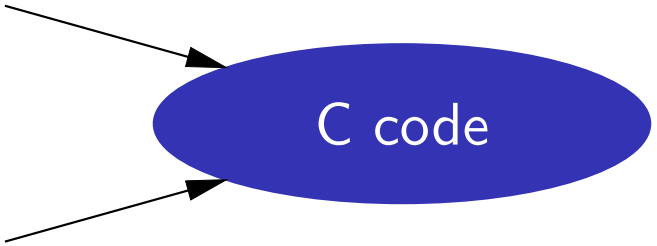
(*) Motto: "unit proofs as a replacement for unit tests"

Examples for the aircraft industry

Rockwell-Collins toolchain
(model-checking + proof)

Caveat
(program proof) (*)

Astrée
(absence of run-time errors, incl. floating-point)



AiT WCET
(precise time bounds)

(*) Motto: "unit proofs as a replacement for unit tests"

Remaining risks

Unsoundness of verification tools:

- The tool does not account for all behaviors of the program.
- Can lead to wrong programs that pass verification.
- C verification tools are particularly vulnerable, owing to the many dark corners of the C language.

Miscompilation and wrong code generation:

- Wrong executables are generated from correct source programs.
- Typical of overly-aggressive optimizations.
- Or just mistakes while writing the compiler.

Miscompilation happens!

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.

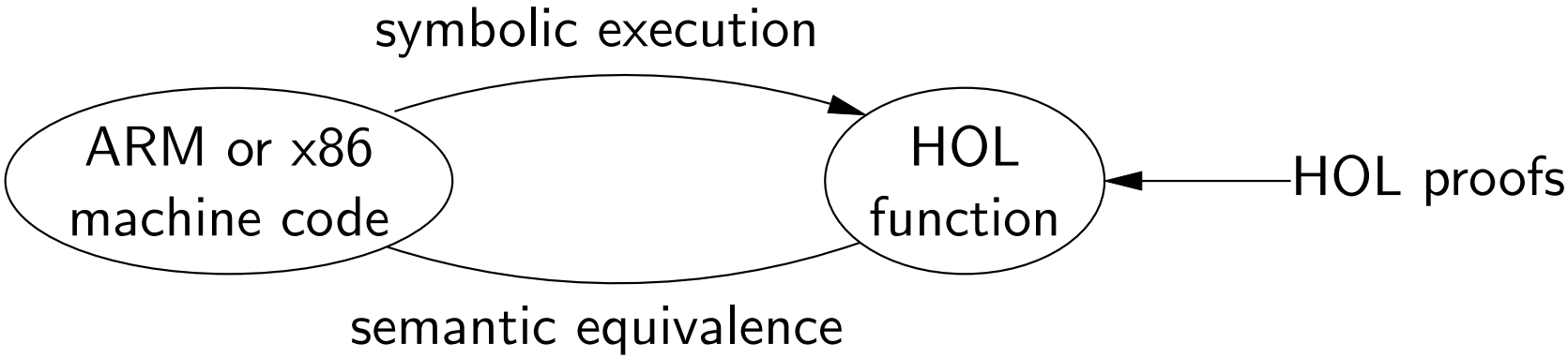
X. Yang, Y. Chen, E. Eide & J. Regehr, PLDI 2011

Foundational approaches

E.g. Foundational Proof-Carrying Code (Appel, Shao, et al); decompilation into logic (Myreen, Gordon, et al).

- Reasoning principles for machine code.
- Built from first principles: operational semantics for machine code.
- Embedded in proof assistants with simple logics (LF, HOL).

Example: Myreen & Gordon's approach.



Limitations: not portable; little automation; hard to scale.

Formal verification of tools

Why not verify (by program proof) the tools themselves?

After all, compilers and verifiers have simple specifications:

> **Semantic soundness:** *If the verification tool raises no alarms, all executions of the program satisfy the properties that were verified.*

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

Additional motivations

Nice algorithms are used in compilers and verification tools
$\rightarrow$ interesting proof challenges.

Widely-used tools
$\rightarrow$ amortize the cost of a formal verification.

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.

Plan

1. The distance between verified model and actual executable
2. CompCert: formal verification of a C compiler
3. Verasco: formal verification of a C static analyzer
4. Future directions

The CompCert project

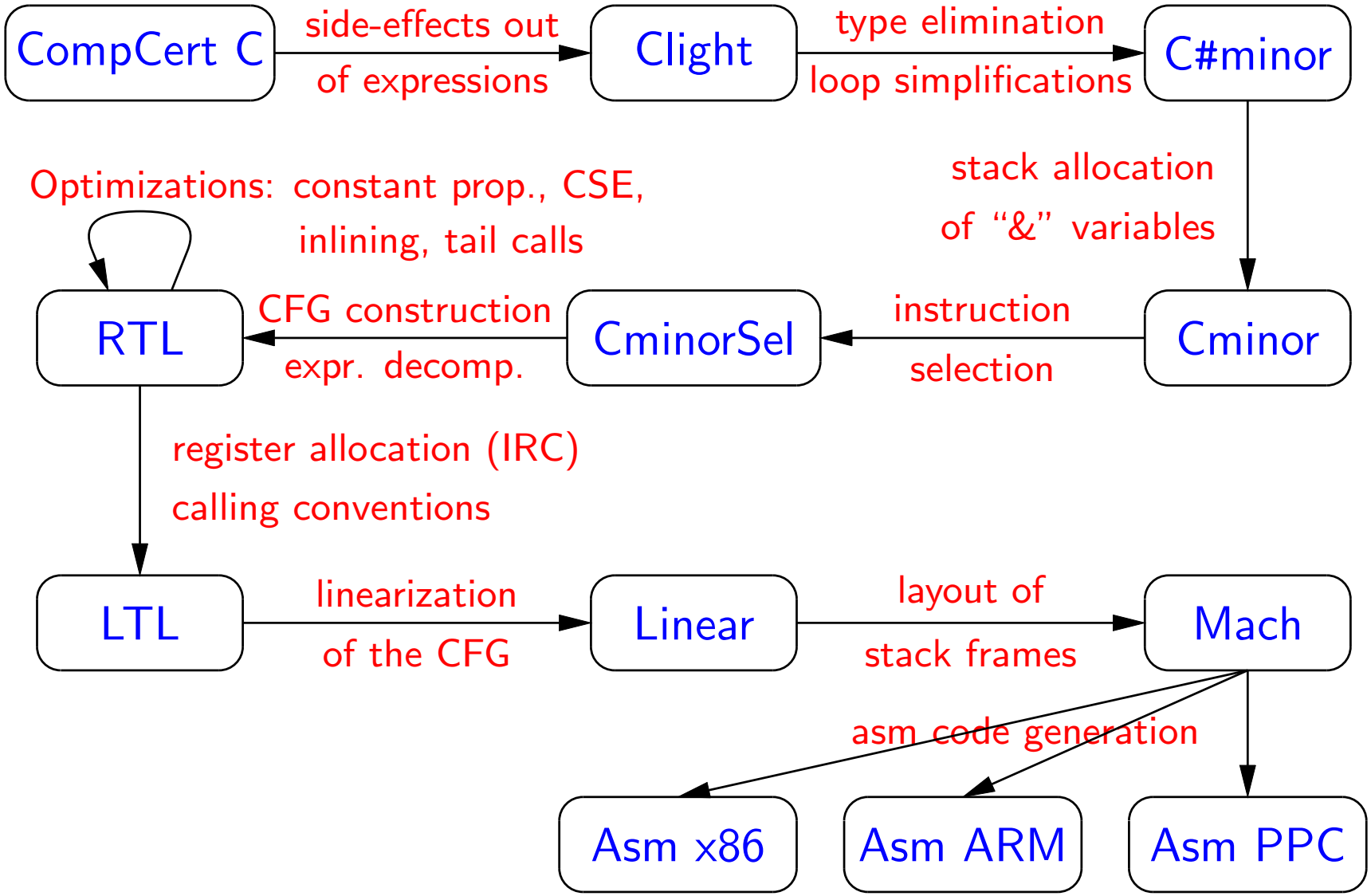
(X.Leroy, S.Blazy, et al)

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

- Source language: a very large subset of C99.
- 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.

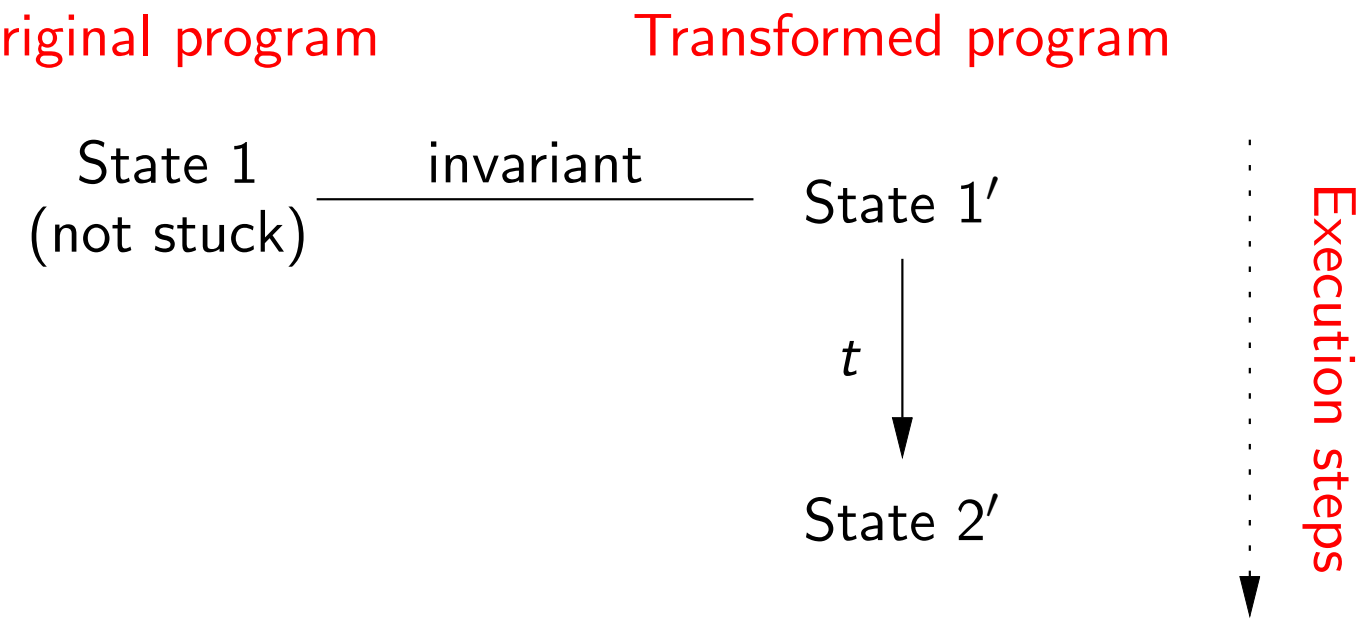
The formally verified part of the compiler



Formally verified using Coq

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

Proof pattern: simulation/refinement diagrams such as:

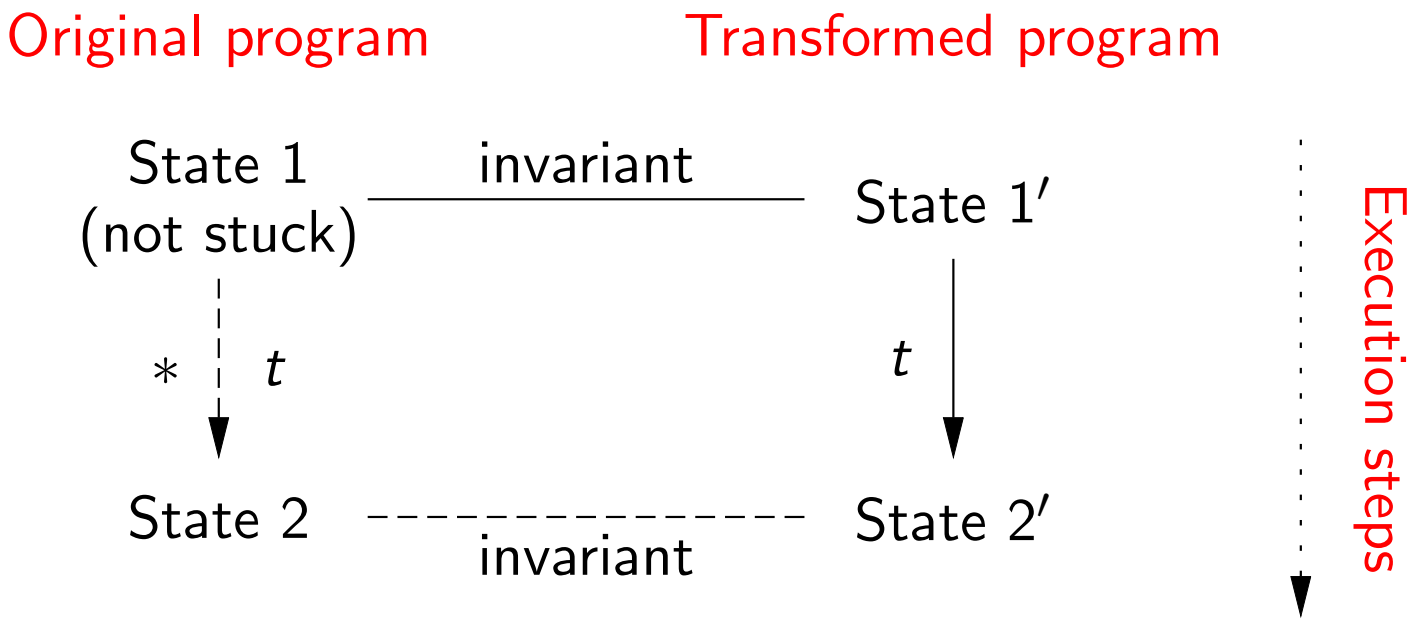


Note: error behaviors ("stuck states") are not preserved, they can be optimized away.

Formally verified using Coq

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

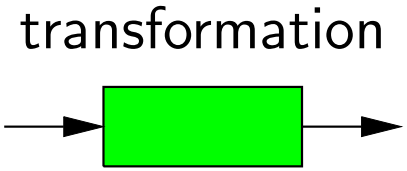
Proof pattern: simulation/refinement diagrams such as:



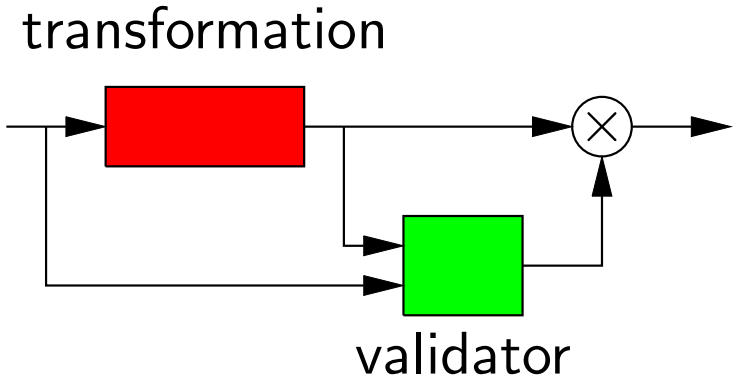
Note: error behaviors ("stuck states") are not preserved, they can be optimized away.

Compiler verification patterns (for each pass)

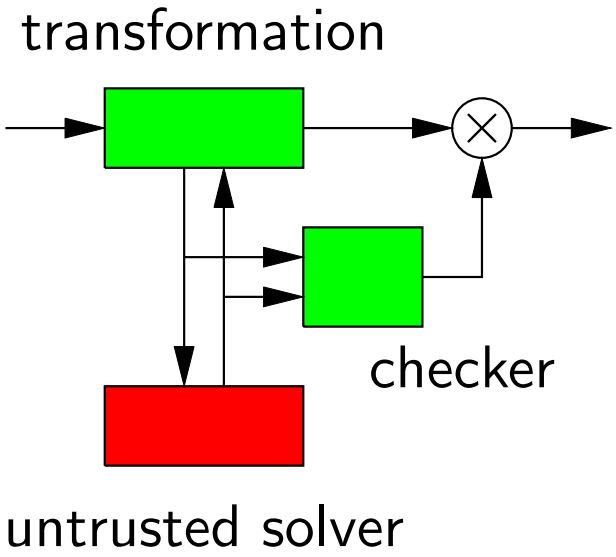
Verified transformation



Verified translation validation



External solver with verified validation



= formally verified

= not verified

Programmed (mostly) in Coq

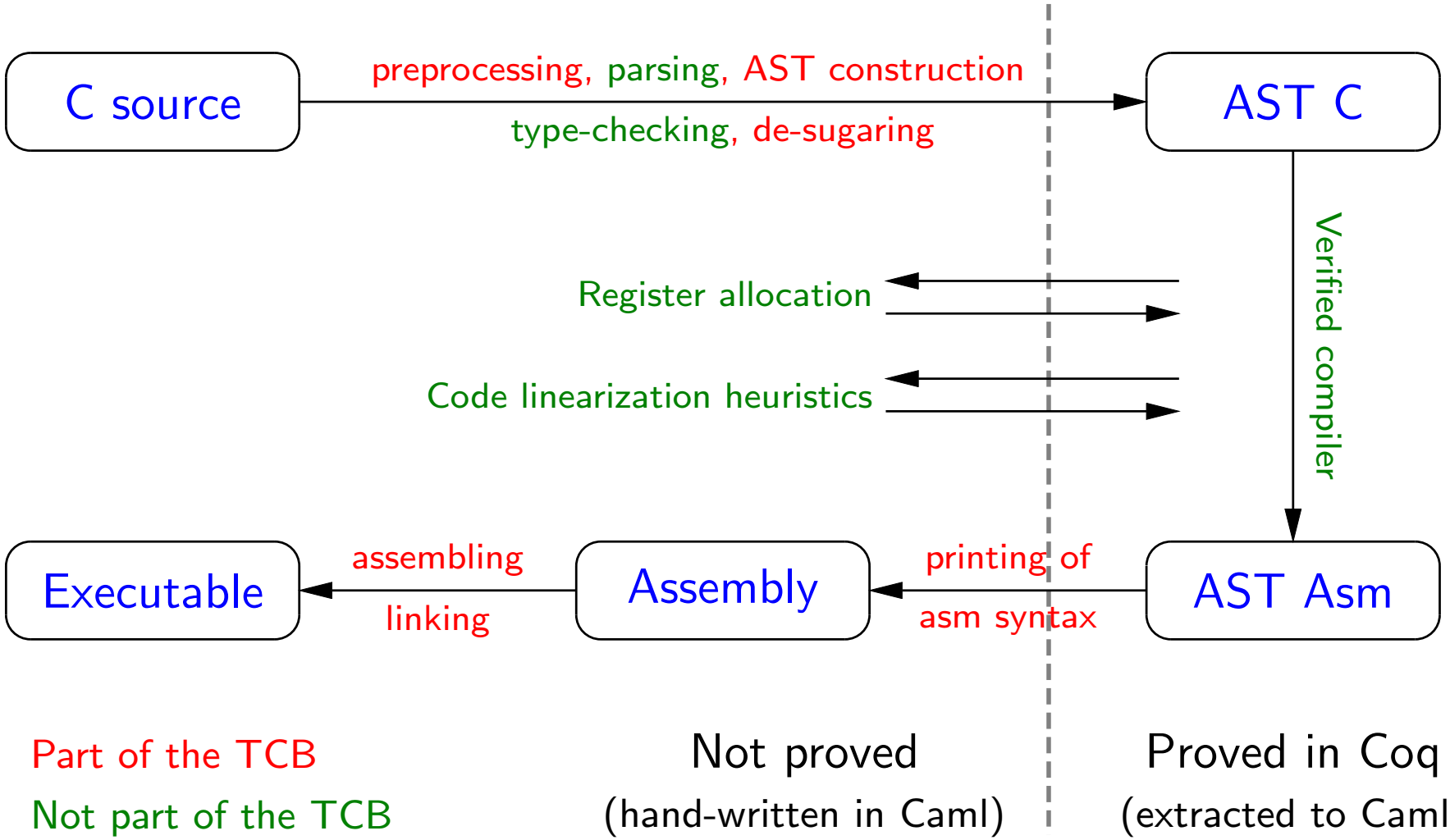
All the verified parts of the compiler are programmed directly in Coq's specification language, using pure functional style.

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

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

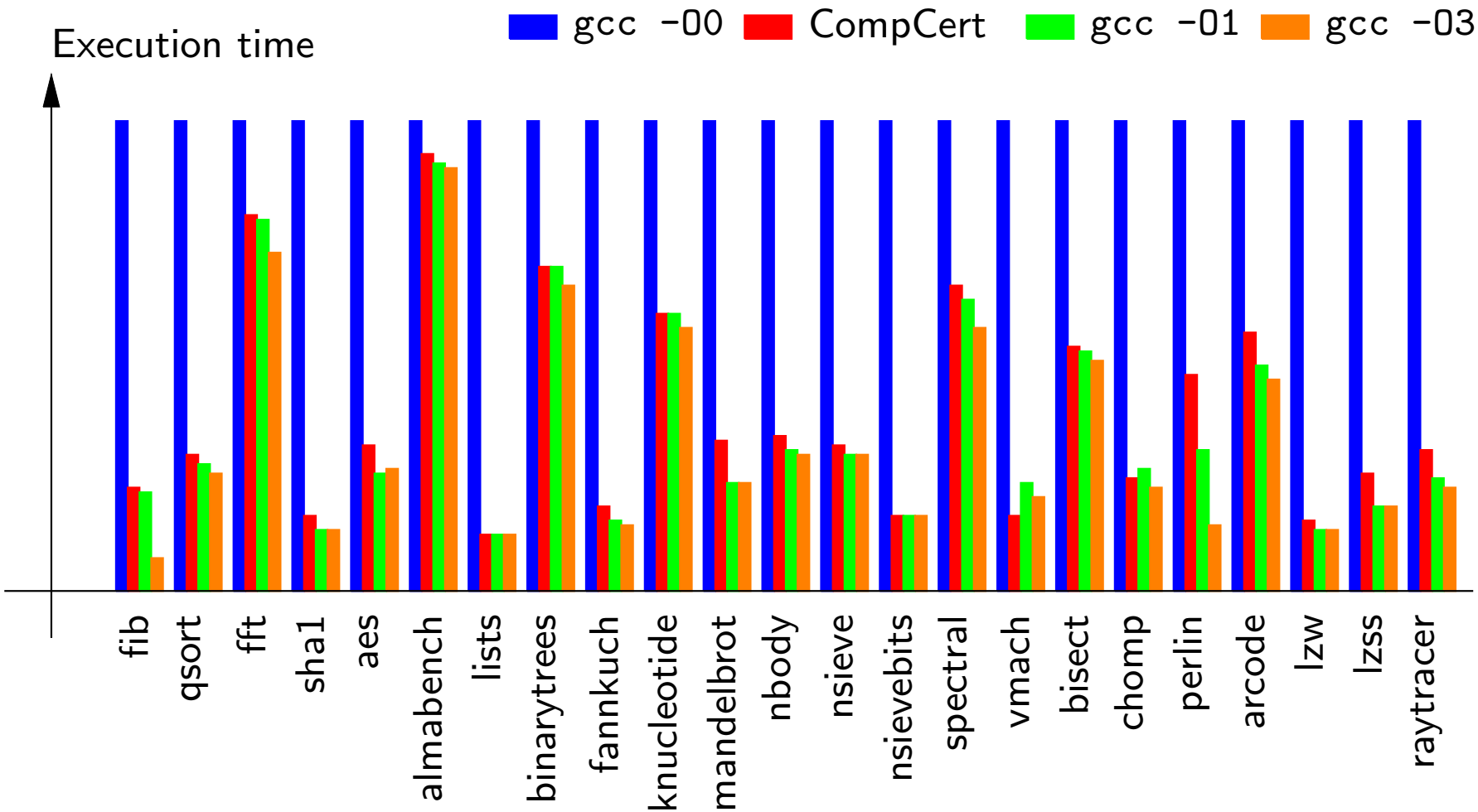
Claim: purely functional programming is the shortest path to writing and proving a program.

The whole Compcert compiler



Performance of generated code

(On a Power 7 processor)



A tangible increase in quality

The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. 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

Plan

1. The distance between verified model and actual executable
2. CompCert: formal verification of a C compiler
3. Verasco: formal verification of a C static analyzer
4. Future directions

The Verasco project

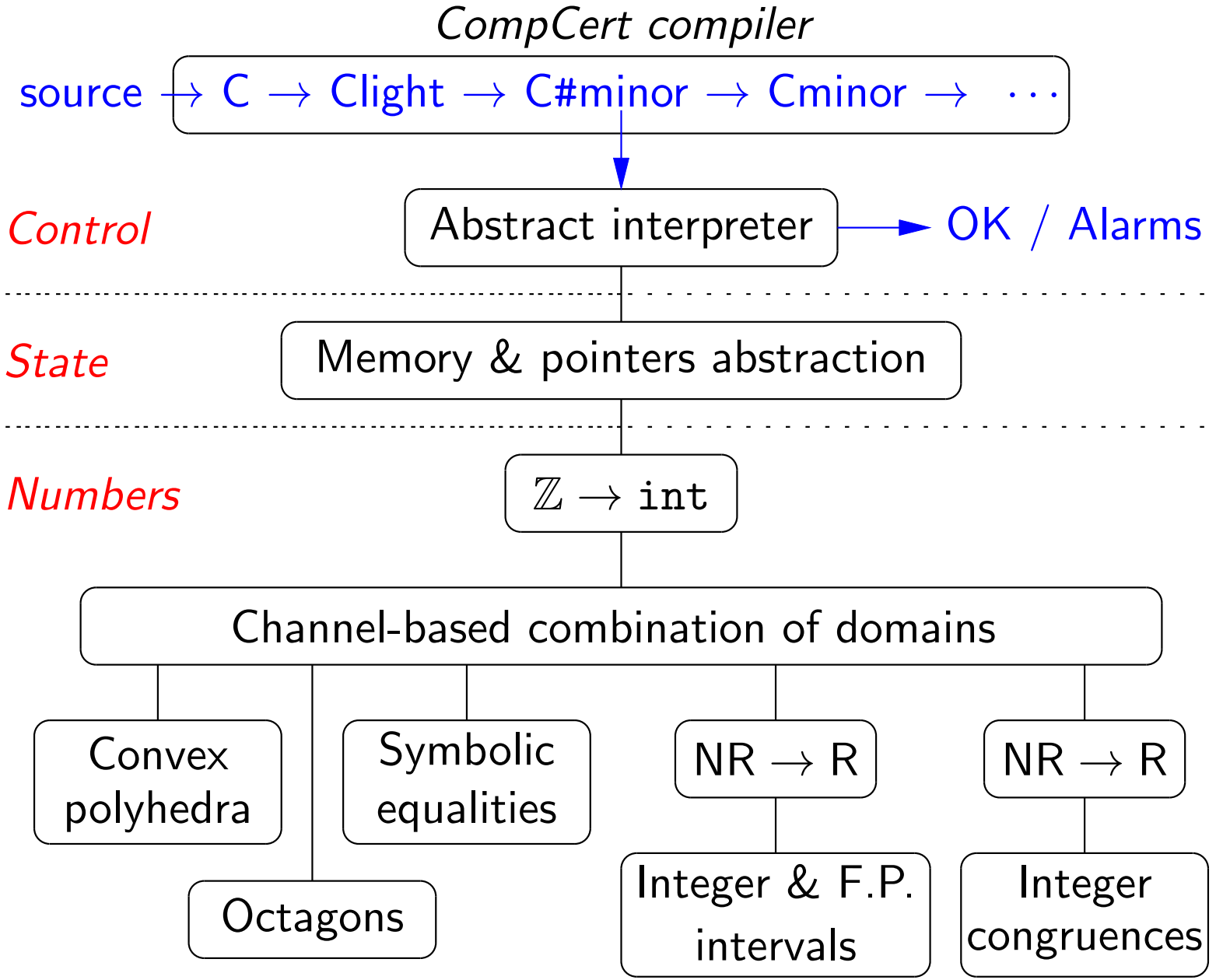
Inria Celtique, Gallium, Abstraction, Toccata + Verimag + Airbus

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.

Slogan: if "CompCert = 1/10th of GCC but formally verified",
likewise "Verasco = 1/10th of Astrée but formally verified".

Architecture



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 `pointsTo` $\{x_1, \ldots, x_n\}$

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	$x = \text{expr}$

Proof methodology

The abstract interpretation framework, with some simplifications:

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

$$\gamma : \text{abstract object} \to \text{set of concrete things}$$

- 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 (30 000 lines of Coq)
- Executable analyzer obtained by extraction.
- Able to show absence of run-time errors in small but nontrivial C programs.

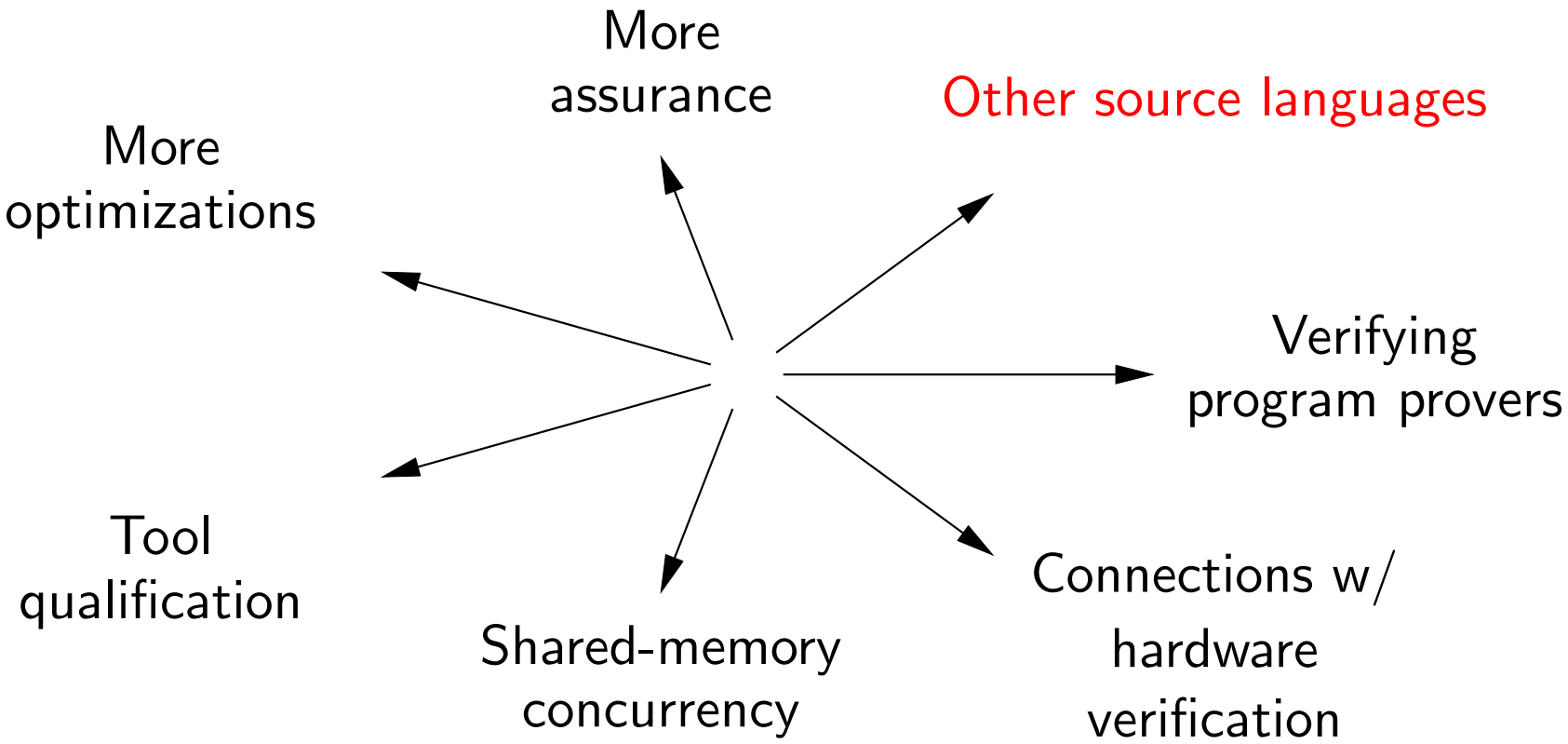
It needs improving!

- Some loops need full unrolling
  (to show that an array is fully initialized at the end of a loop).
- Analysis is slow (e.g. 1 minutes for 100 LOC).

Plan

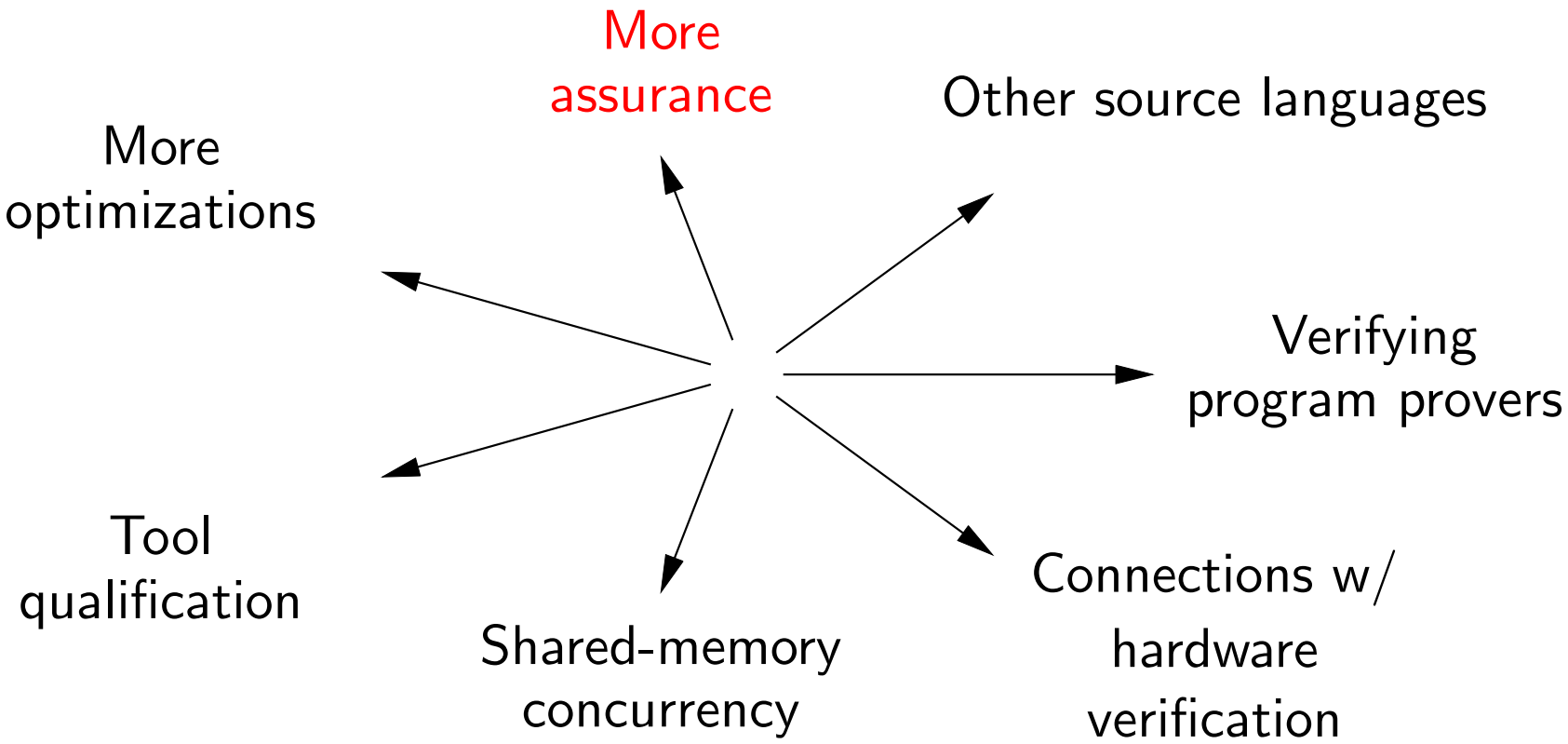
1. The distance between verified model and actual executable
2. CompCert: formal verification of a C compiler
3. Verasco: formal verification of a C static analyzer
4. Future directions

Ongoing and future work



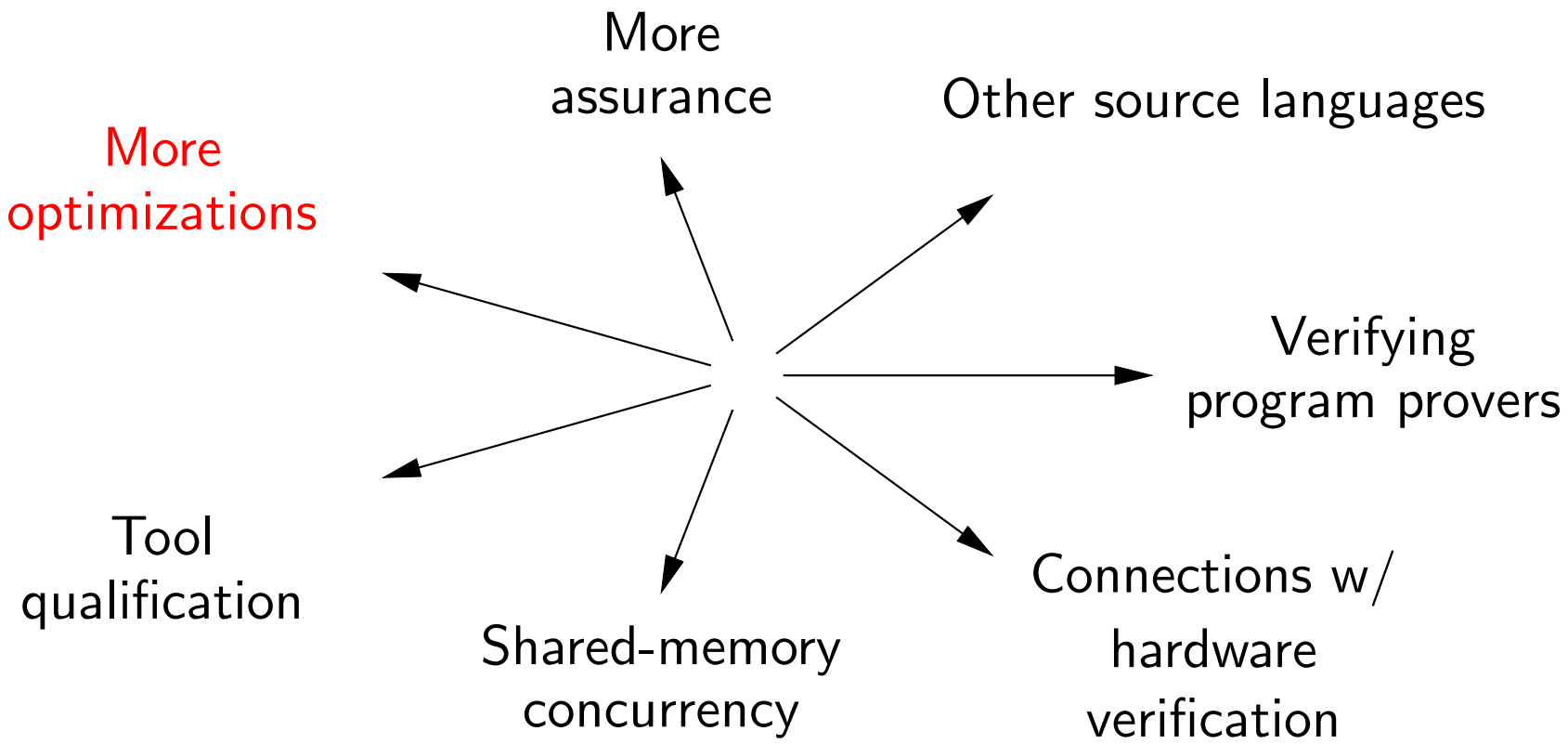
Other source languages besides C, see next slide.

Ongoing and future work



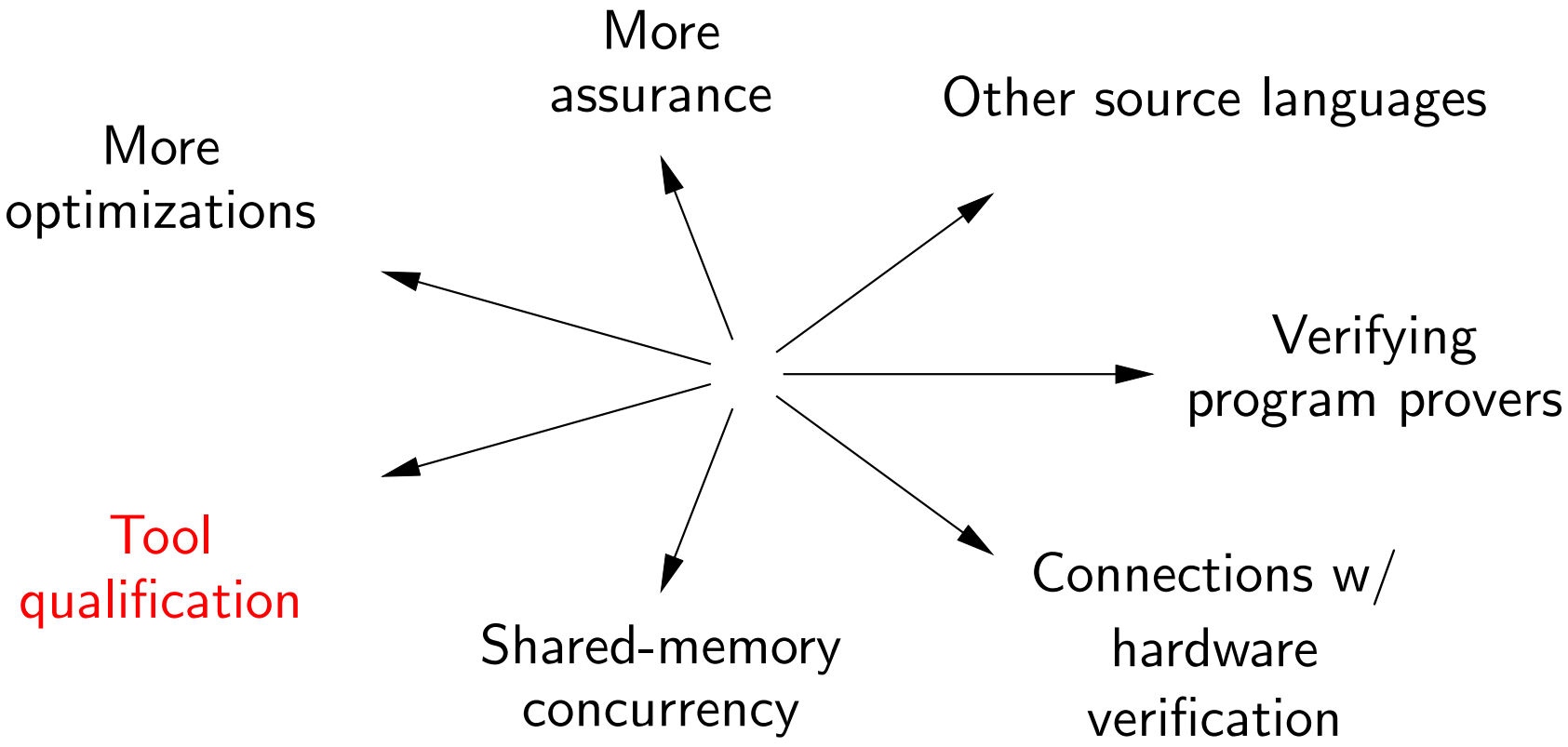
Prove or validate more of the trusted base:
preprocessing, lexing, elaboration, assembling, linking, ...

Ongoing and future work



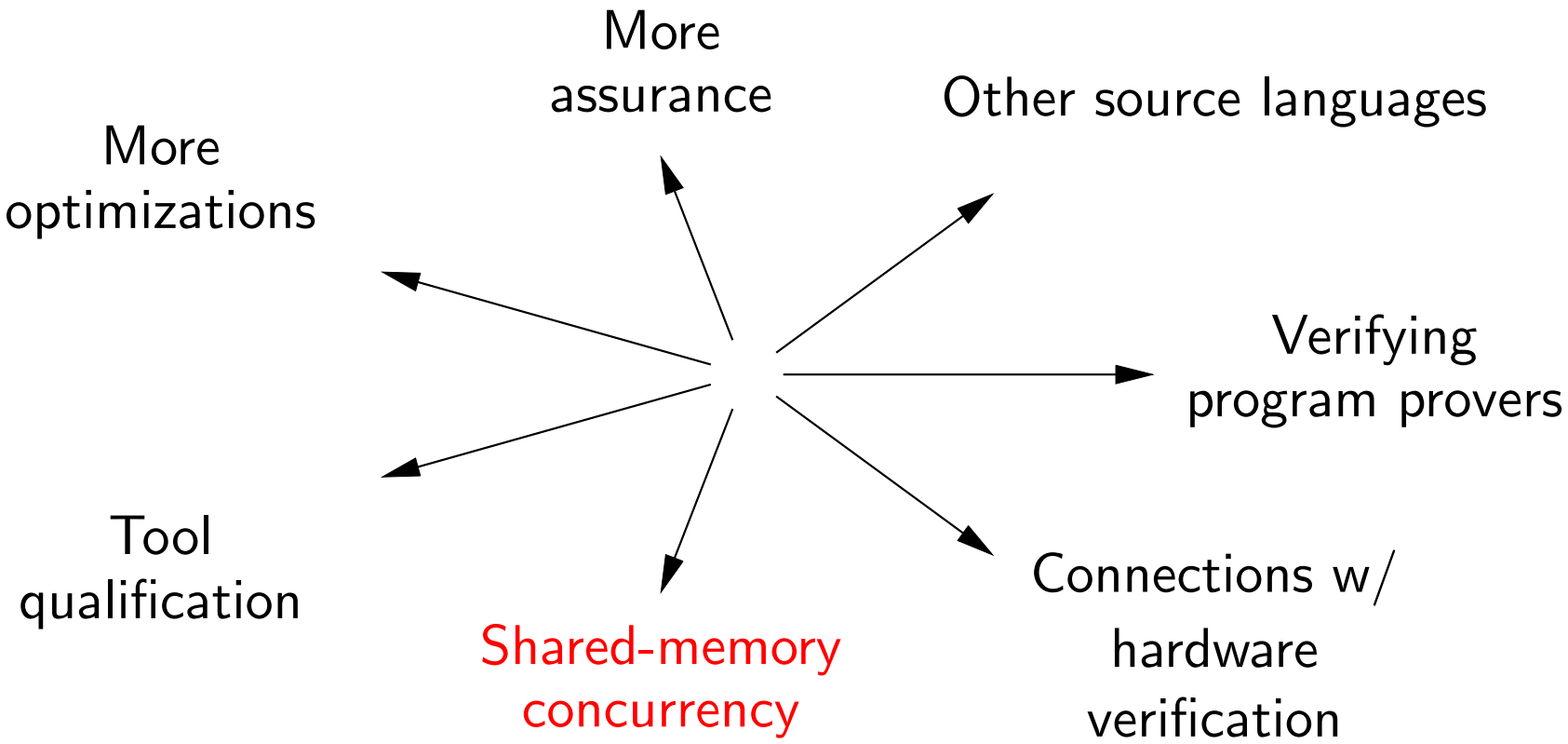
Add advanced optimizations to CompCert, e.g. loop optims.
Add more advanced abstract domains to Verasco.

Ongoing and future work



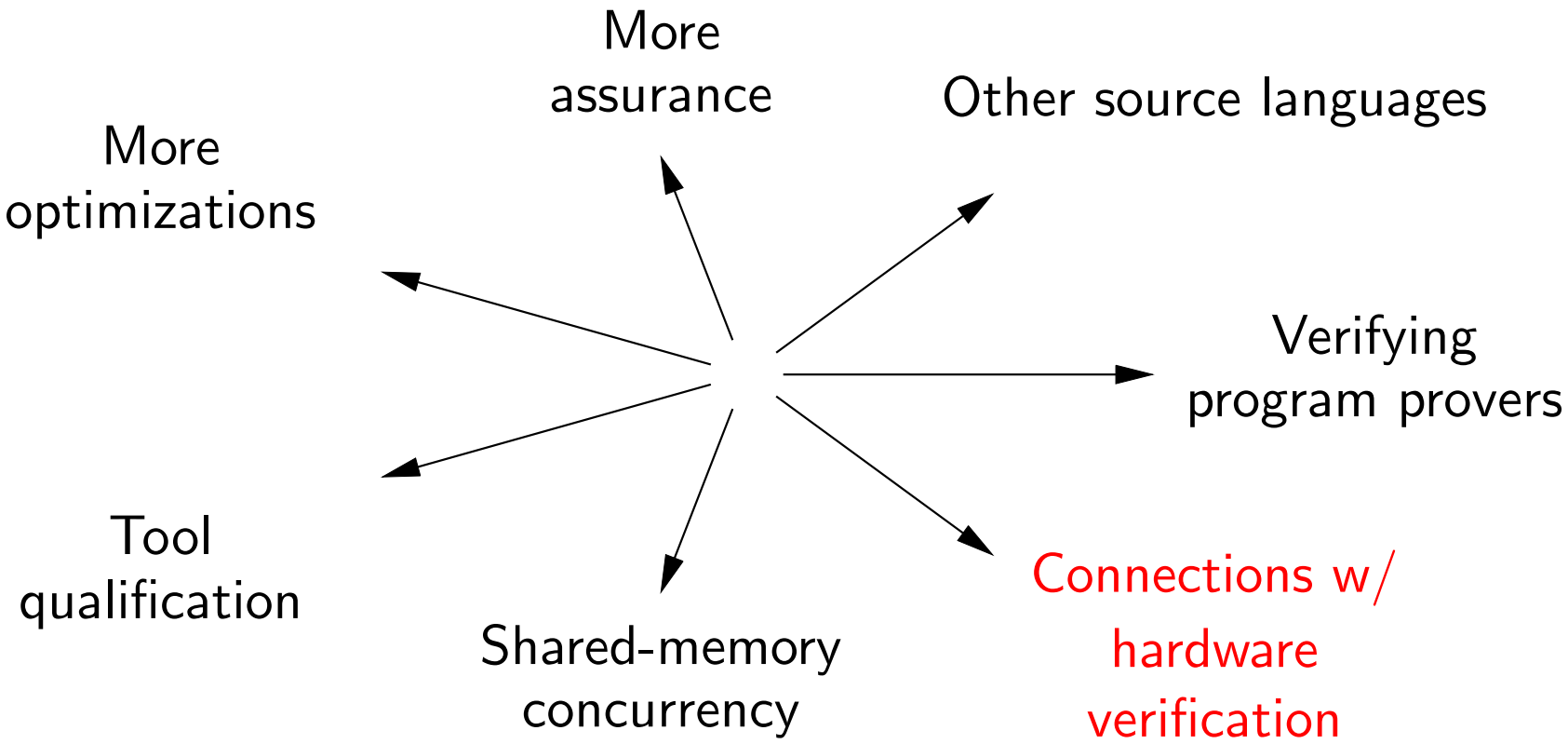
Taking advantage of CompCert's and Verasco's proofs for a DO-330/DO-178 tool qualification.

Ongoing and future work



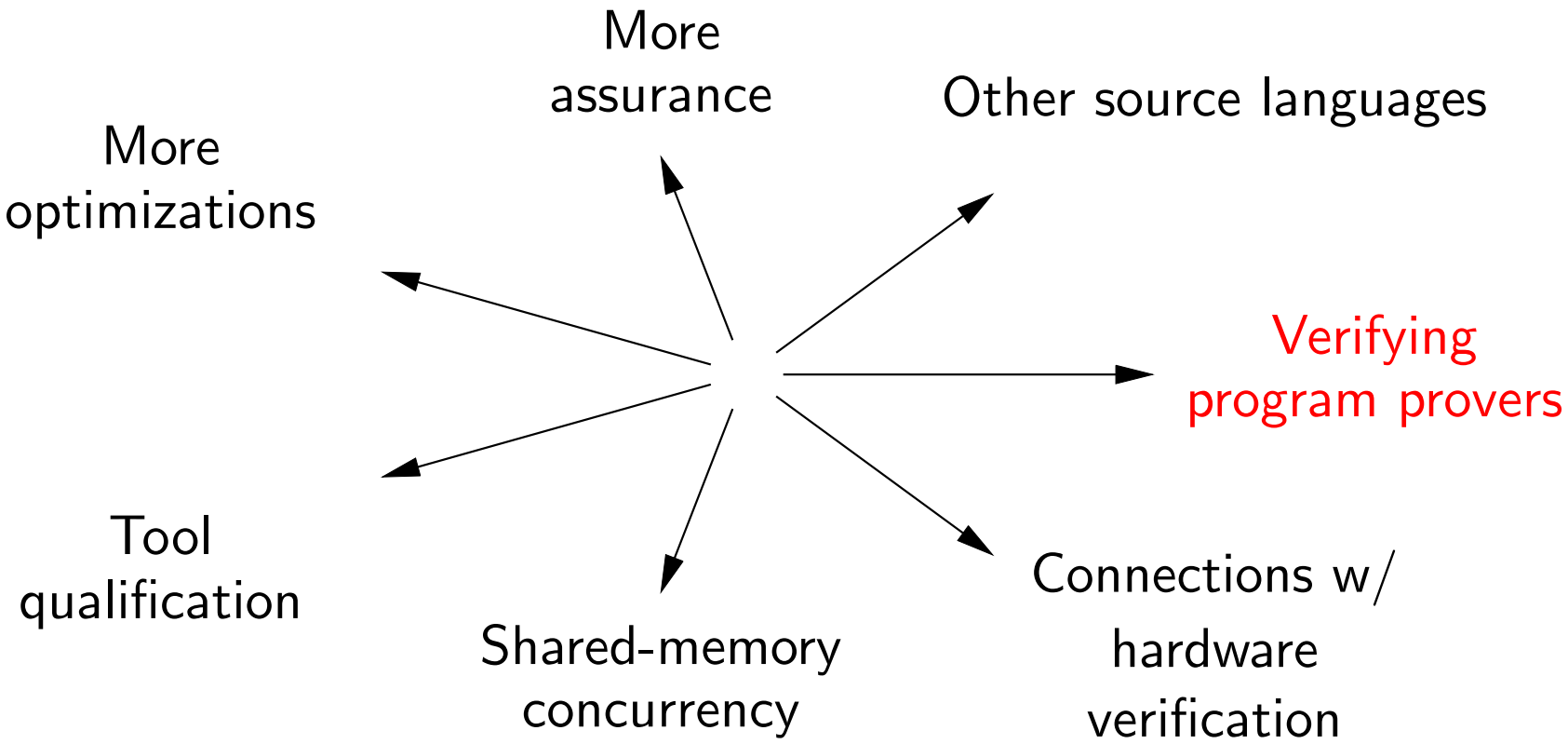
"The elephant in the room." See next slide.

Ongoing and future work



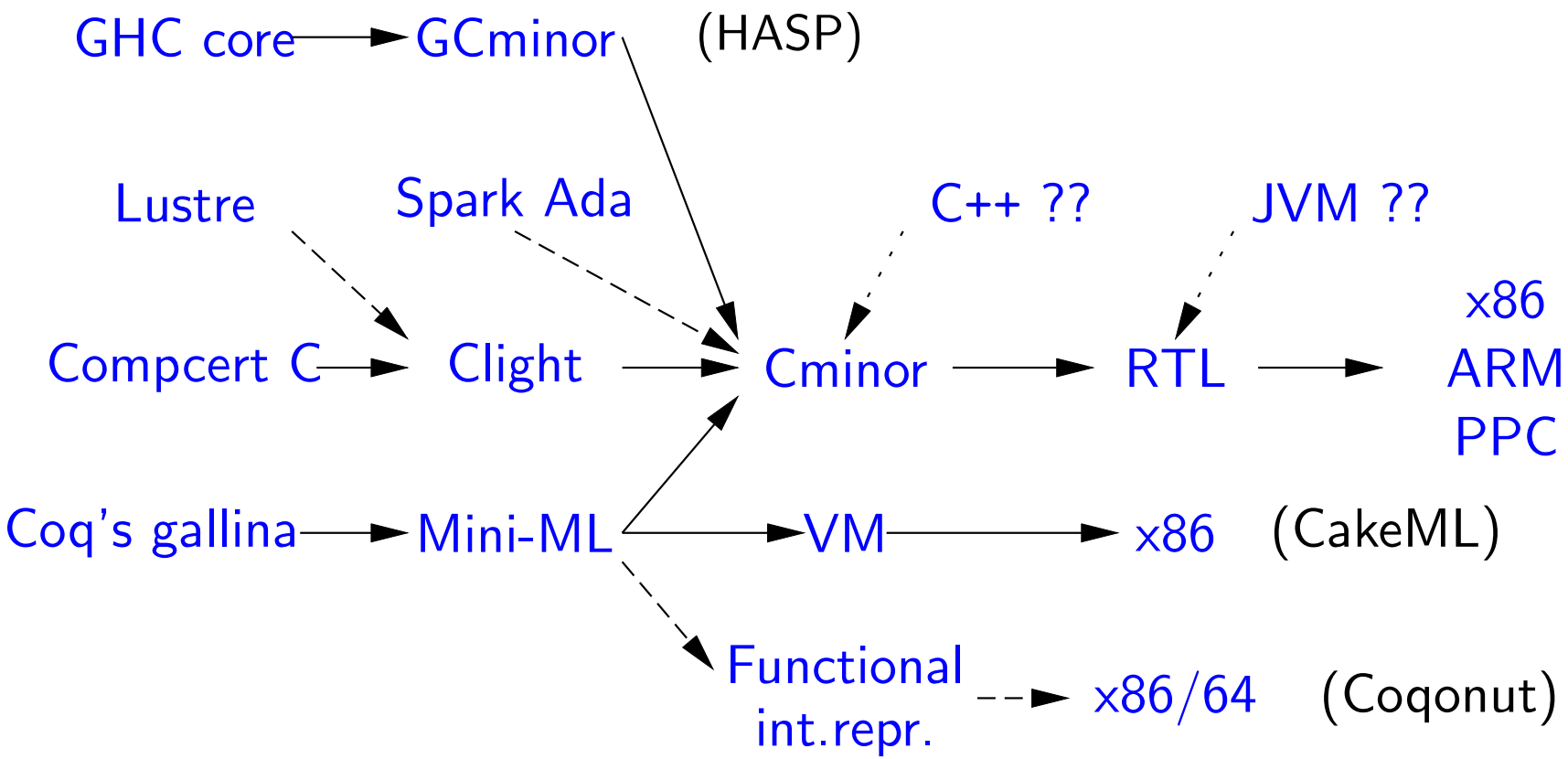
Formal specs for architectures & instruction sets, as the missing link between compiler verification and hardware verification.

Ongoing and future work



E.g. the VST program logic for Clight of Appel et al, or the verified v.c.gen. of Herms.

Verified compilers for other languages



Reusing intermediate languages of CompCert, or not.

An old problem: need appropriate formal semantics for the language...

A new problem: run-time system verification (exceptions, allocator, GC).

Towards shared-memory concurrency

Programs containing data races are generally compiled in a non-semantic-preserving manner.

- Apparently atomic operations are decomposed into sequences of instructions, exhibiting more behaviors than present in the source.
- Weakly-consistent memory models, as implemented in hardware, introduce more behaviors than just interleavings of loads and stores.
- Some classic optimizations are unsound for certain memory models (e.g. CSE of loads versus TSO).

Towards shared-memory concurrency

For data race-free source programs: (A. Appel et al, Princeton)
E.g. programs provable in concurrent separation logic.

- Most of the compiler can assume pseudo-sequential behaviors.
- Almost all sequential optimizations & proofs remain valid.
- Weakly-consistent memory appears very late (machine level).

For C2011-style low-level atomics:
Expose concurrency & relaxed memory models in Compcert C and throughout the compiler.

- Precise semantics of low-level atomics are unclear!
- Generally formalized in axiomatic style (event structures).
- New proof principles needed for semantic preservation and abstract interpretation.

To conclude...

Critical software deserves the best verification and code generation tools computer science can provide.

Let's make this a reality!