Compiler verification and beyond: verified tools for high-assurance software

Xavier Leroy

INRIA Paris-Rocquencourt

HCSS, 2011-05-03

The coming of age of formal verification tools

Formal, tool-assisted verification of critical software:

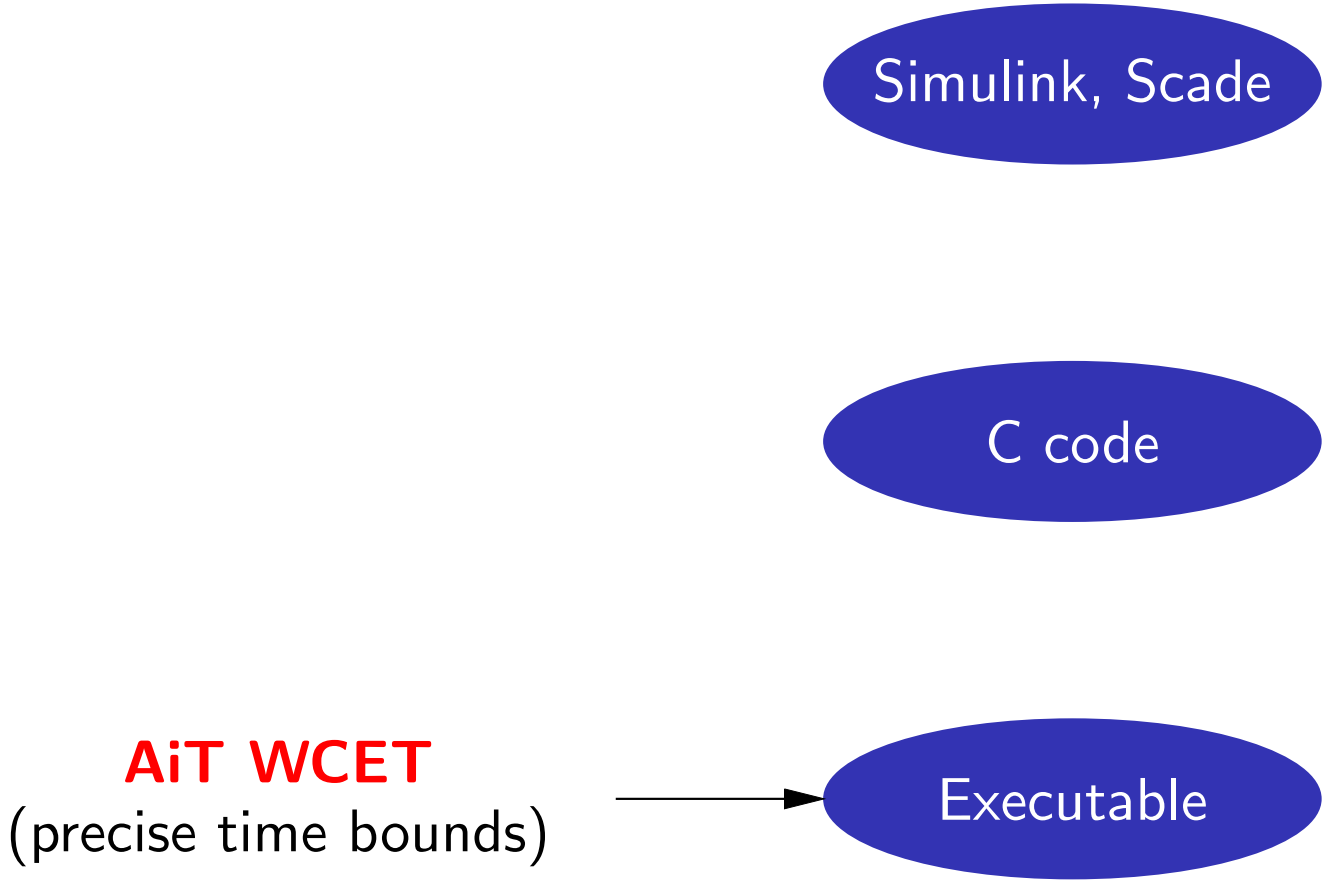
- an old academic dream
- . . . that is slowly entering industrial practice.

Motivations:

- Stronger assurance.
- Saves on testing.
- Regulations (Common Criteria EAL7; DO-178C).

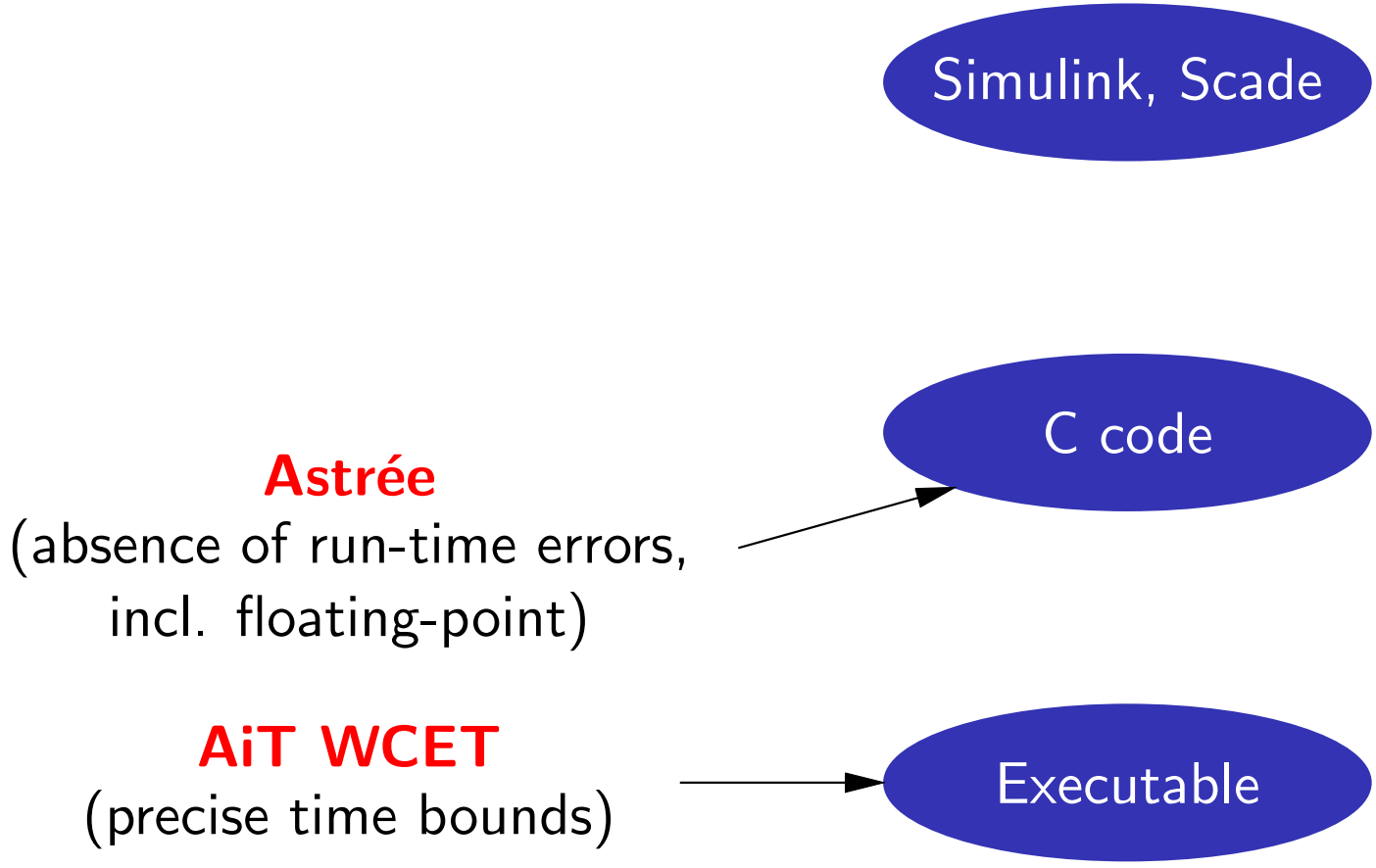
Some success stories in verification of avionics code

(e.g. fly-by-wire systems)



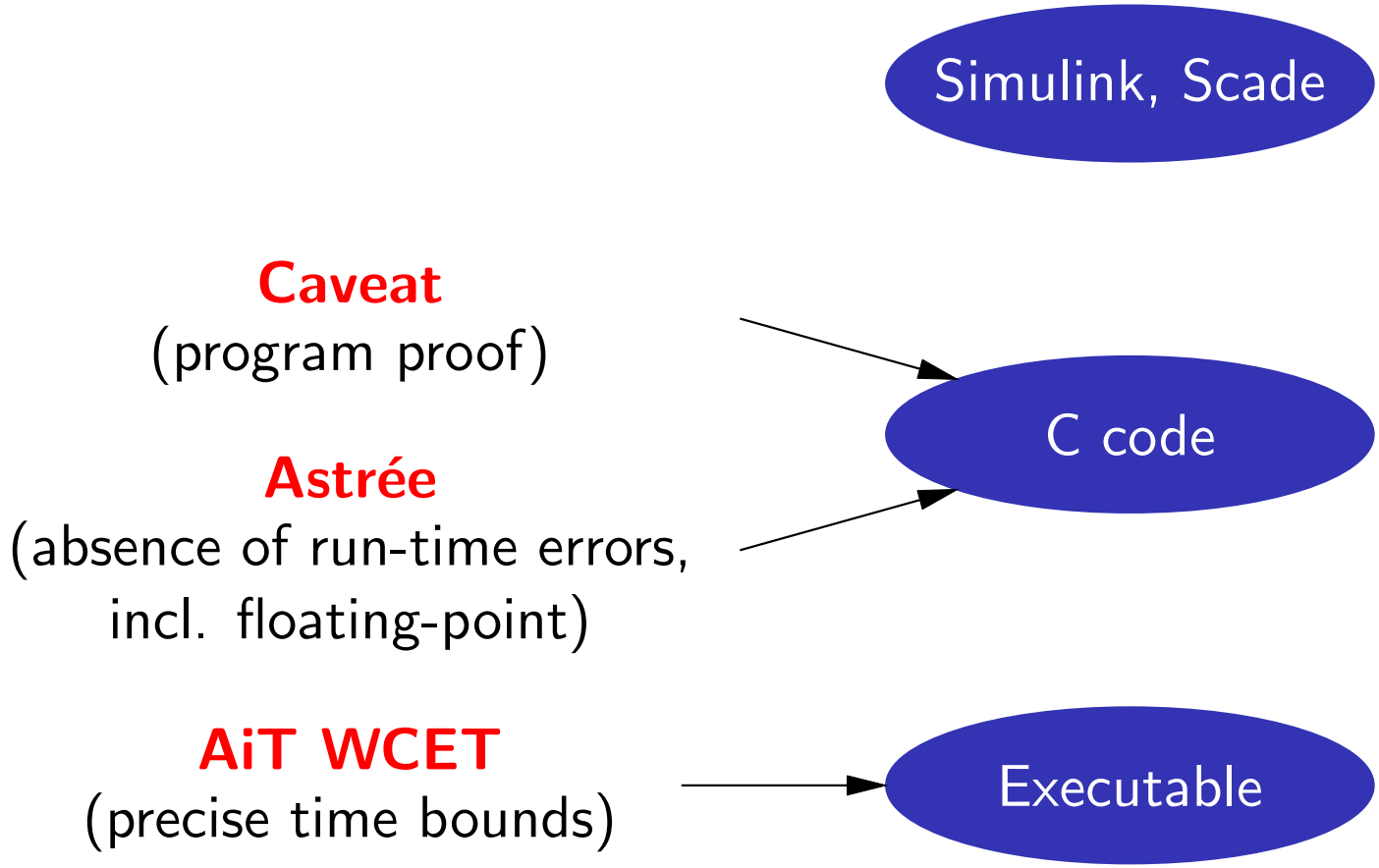
Some success stories in verification of avionics code

(e.g. fly-by-wire systems)



Some success stories in verification of avionics code

(e.g. fly-by-wire systems)



Some success stories in verification of avionics code

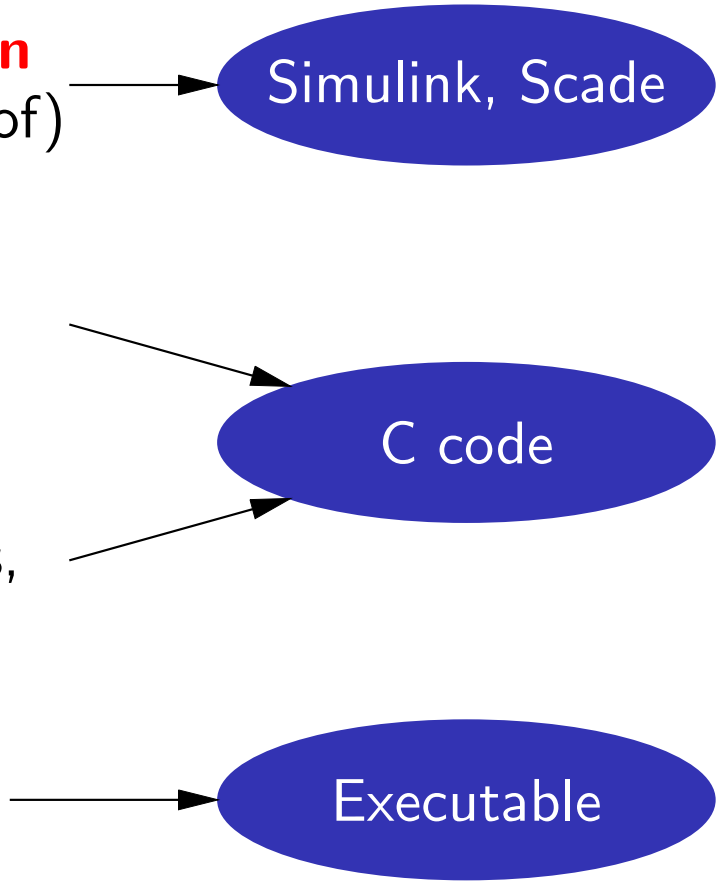
(e.g. fly-by-wire systems)

Rockwell-Collins toolchain
(model-checking + PVS proof)

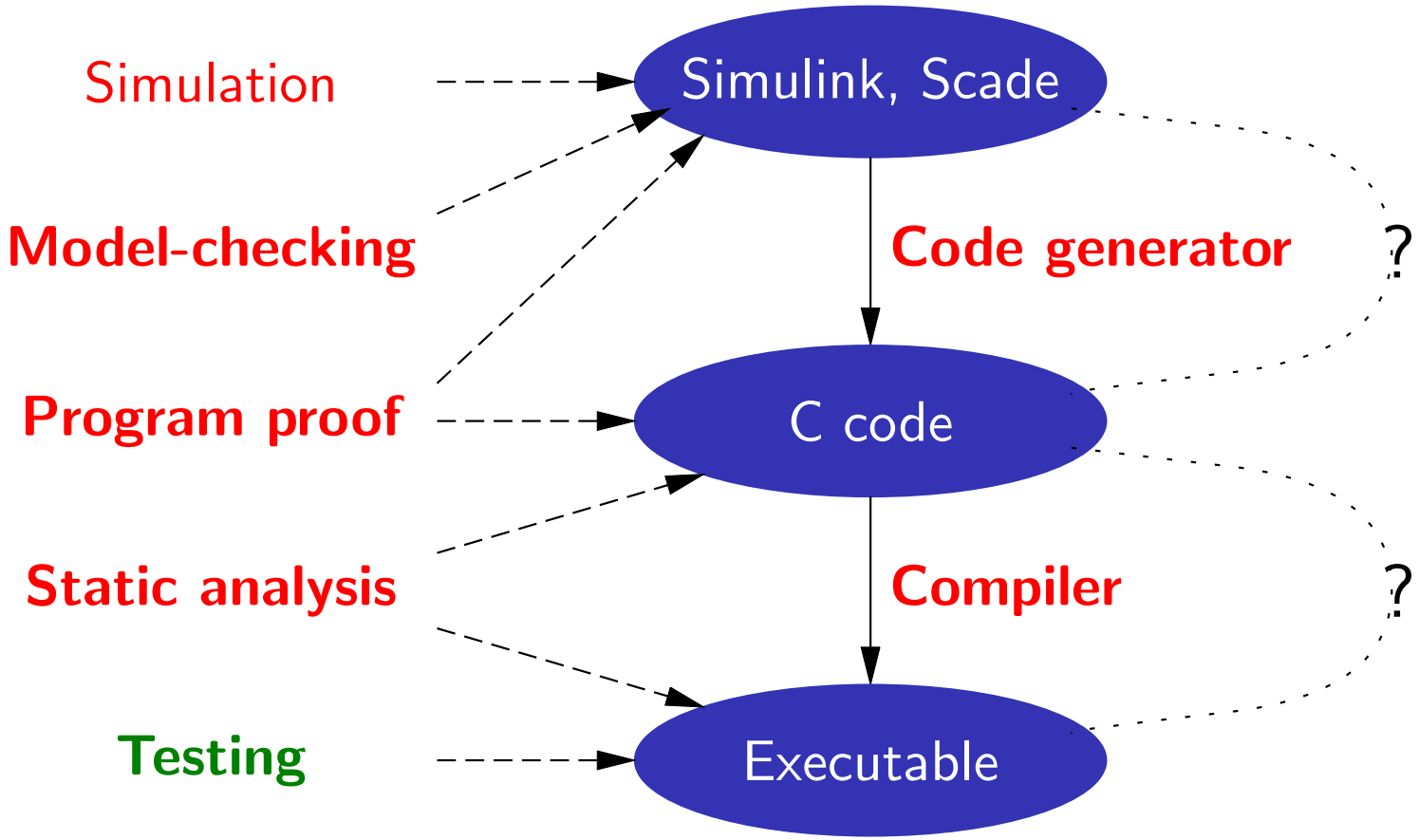
Caveat
(program proof)

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

AiT WCET
(precise time bounds)



Trust in formal verification



Are verification tools semantically sound?
Couldn't compilers and code generators miscompile correct source codes?

Can you trust your compiler?

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

An example of compiler optimization

Consider:

```
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;
}
```

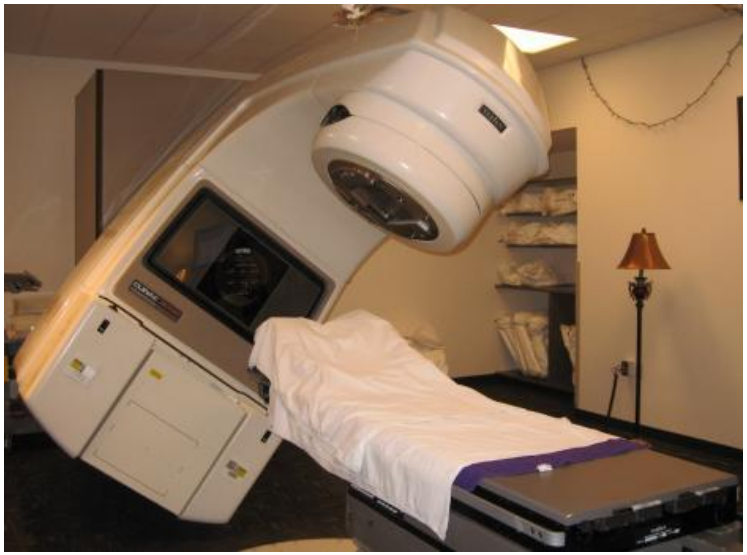
Compiled with the Tru64/Alpha compiler and manually decompiled back to C...

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.

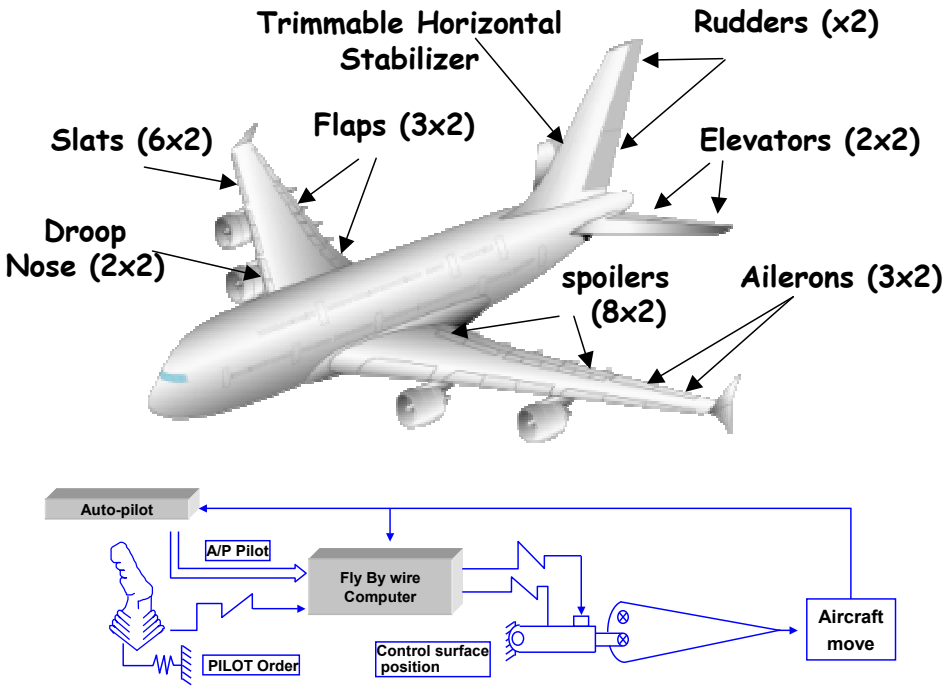
Are miscompilation bugs a problem?



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.

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.*

As a corollary, we obtain:

> *Any safety property of the observable behavior of the source program carries over to the generated executable code.*

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.

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

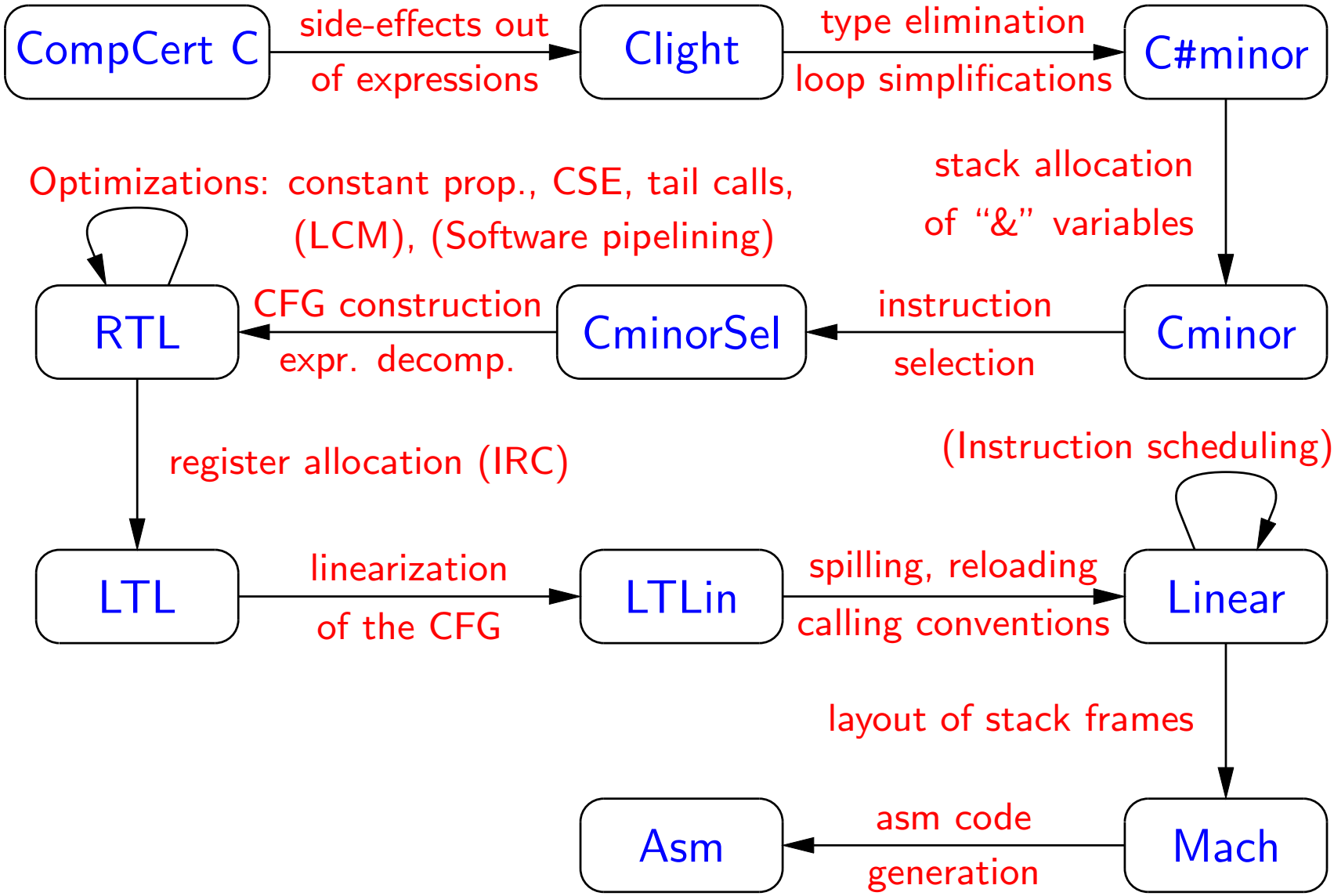
Not supported at all:

- 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

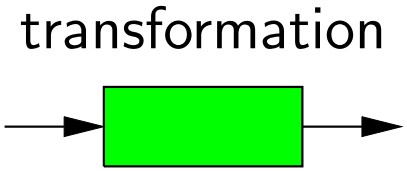
After 50 000 lines of Coq and 4 person.years of effort:

```
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).
```

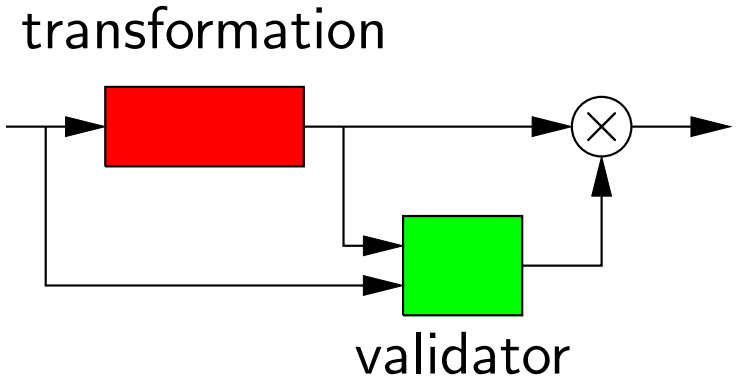
Behaviors `beh` = termination / divergence / going wrong
+ trace of I/O operations (syscalls, volatile accesses).

Compiler verification patterns (for each pass)

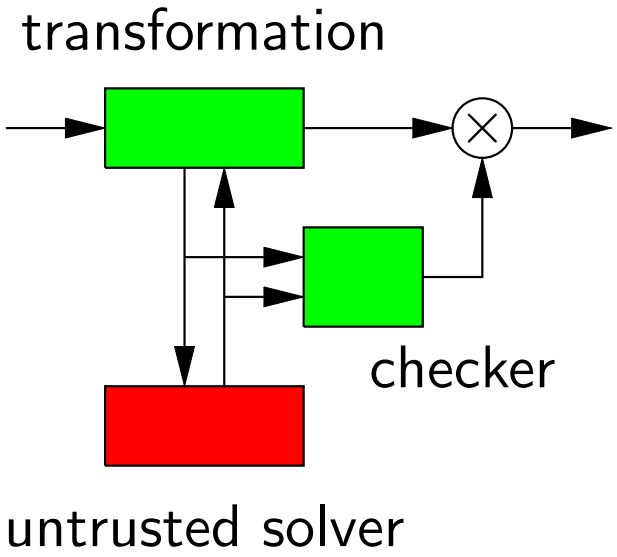
Verified transformation



Verified translation validation



External solver with verified validation



= formally verified

= not verified

Programmed (mostly) in Coq

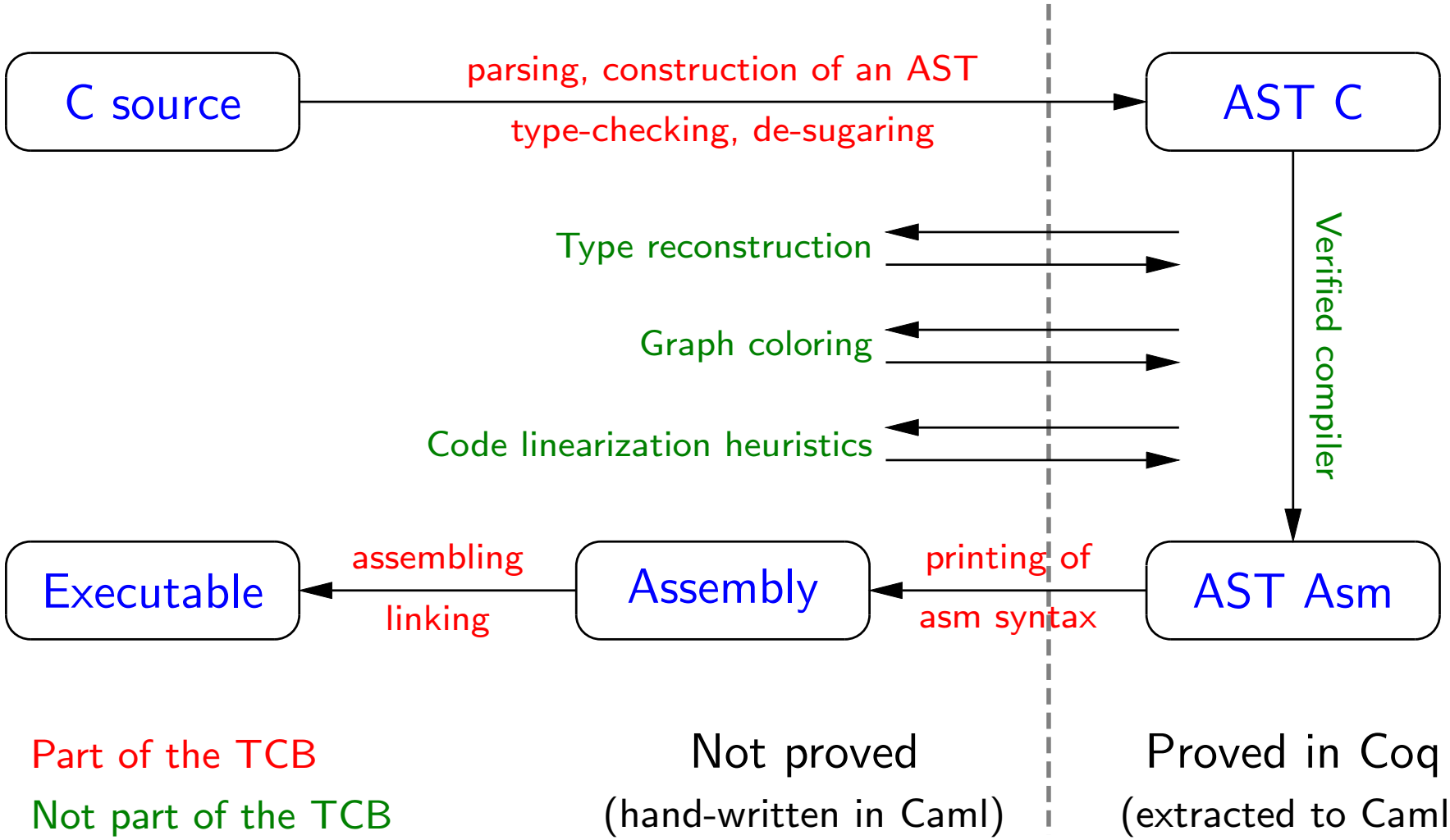
All the verified parts of the compiler are programmed directly in Coq's specification language, using pure functional style.

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

Advantage: no need for a program logic!
(Implementations = functional specifications.)

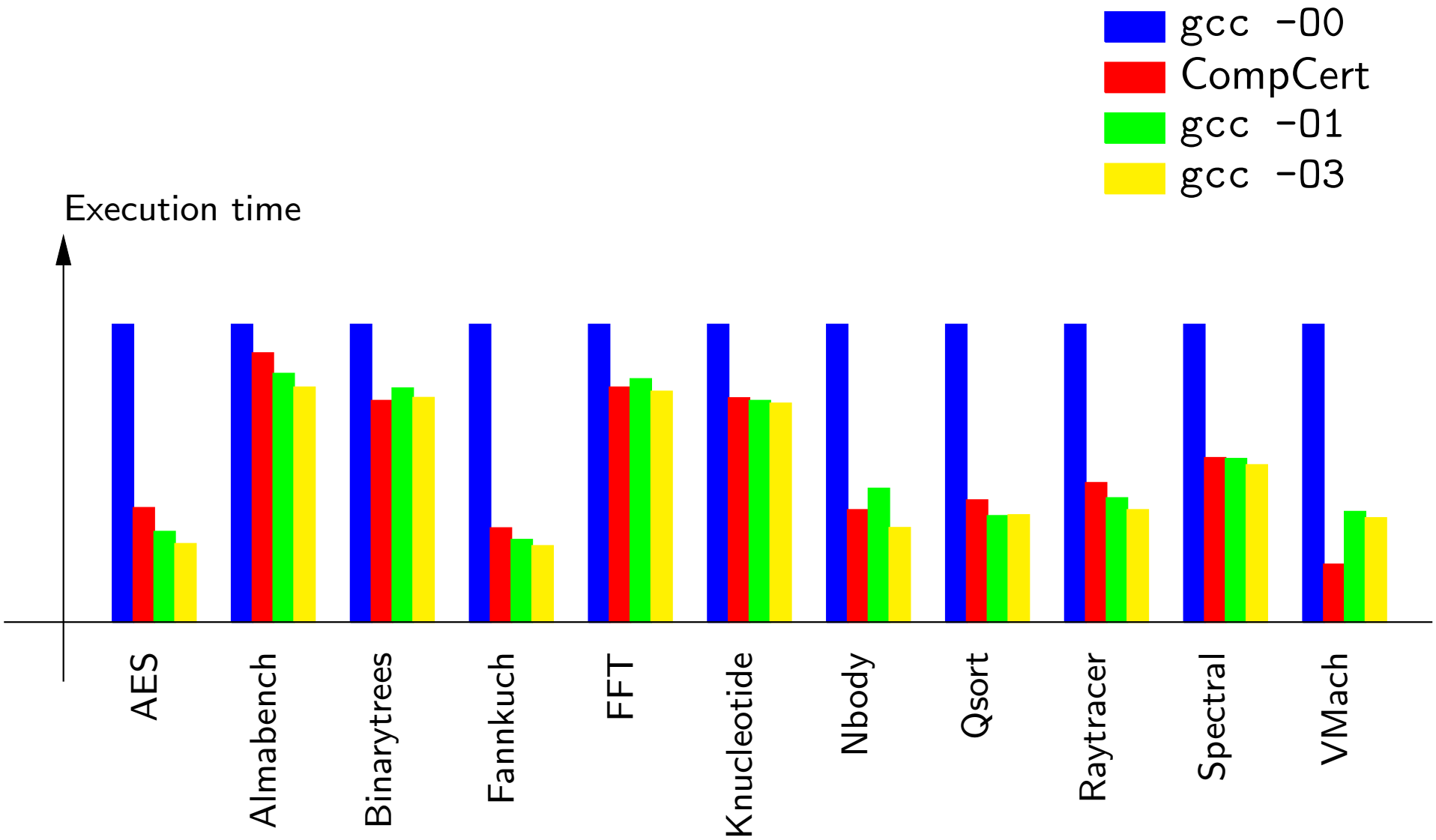
Executable via automatic extraction Coq $\rightarrow$ Caml
and Caml compilation.

The whole CompCert compiler



Performance of generated code

(On a PowerPC G5 processor)

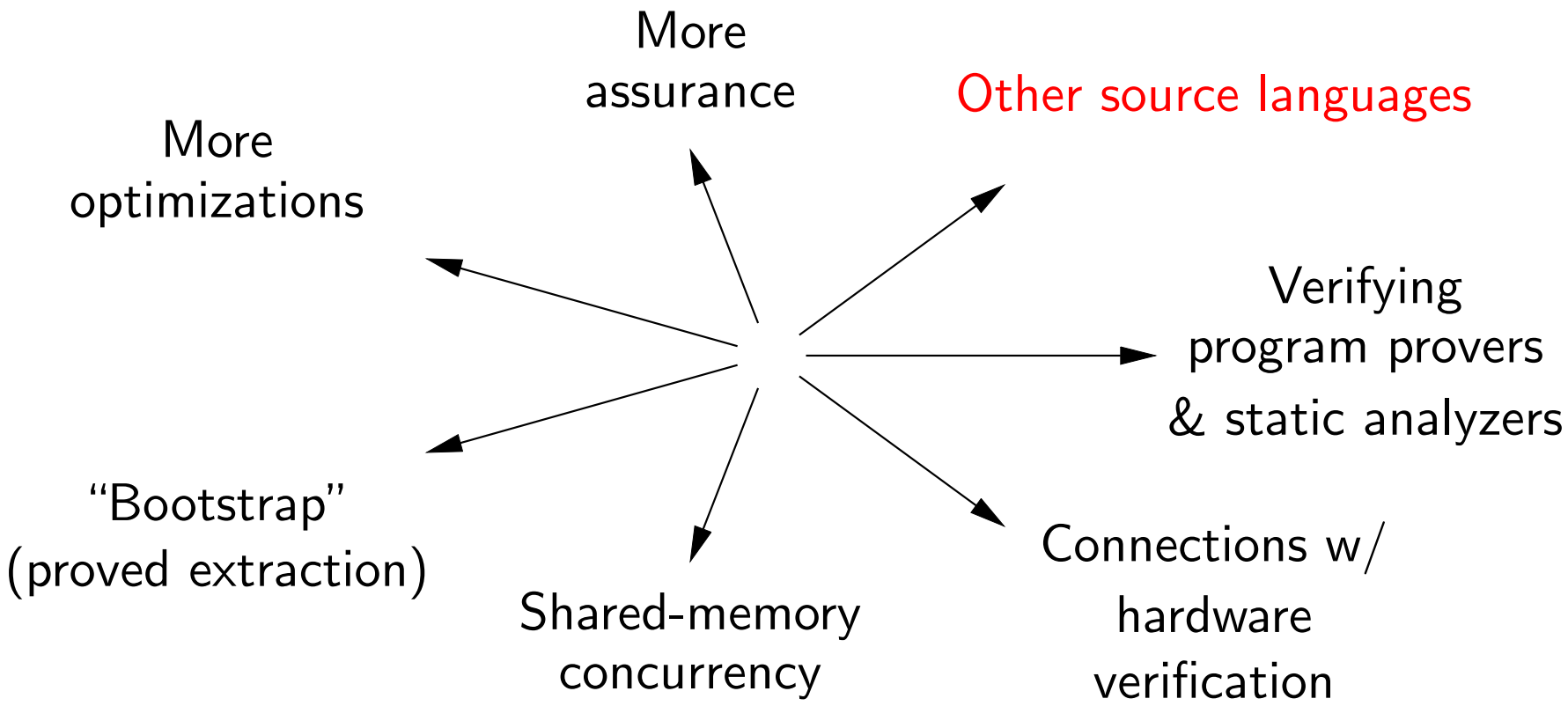


Current status

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

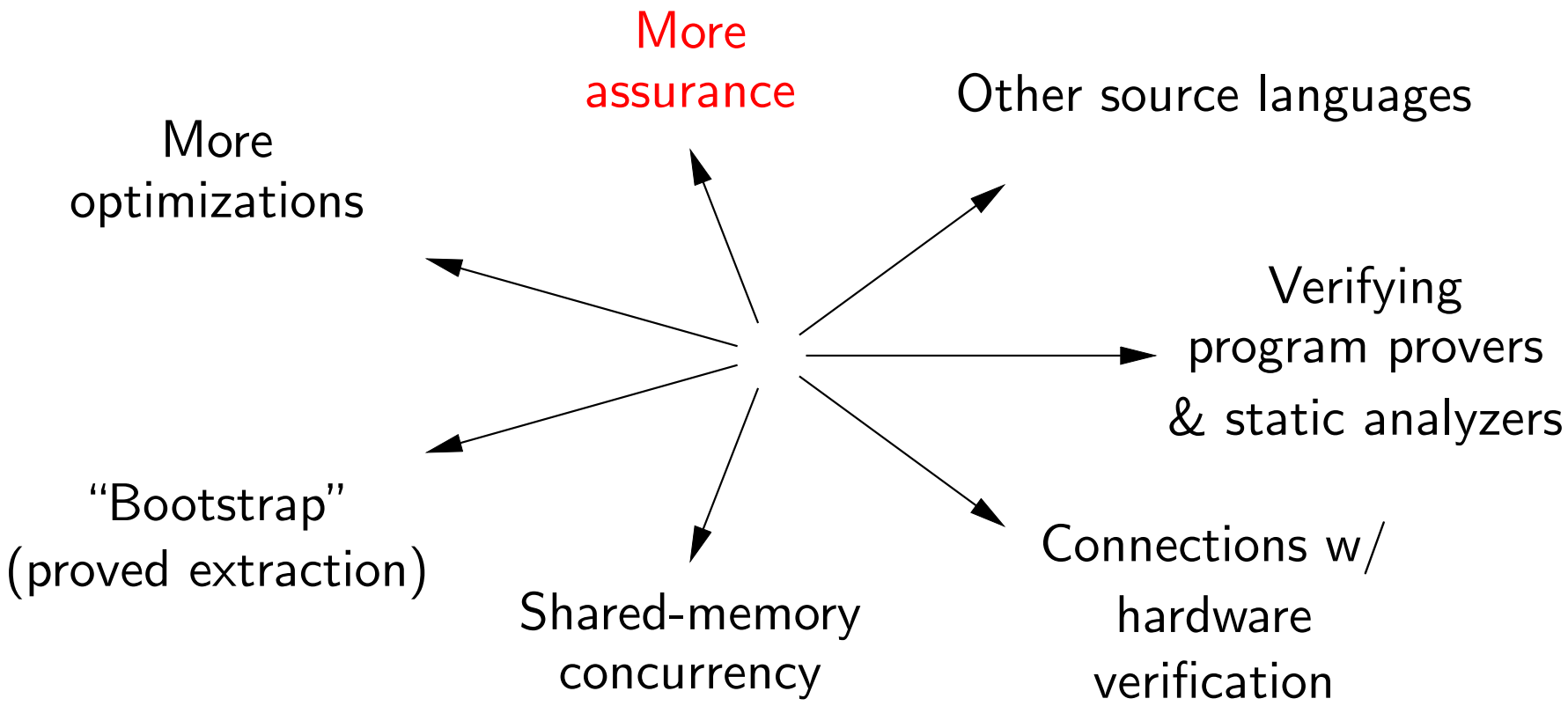
(Within the limitations of today's proof assistants such as Coq.)

Some directions for future work



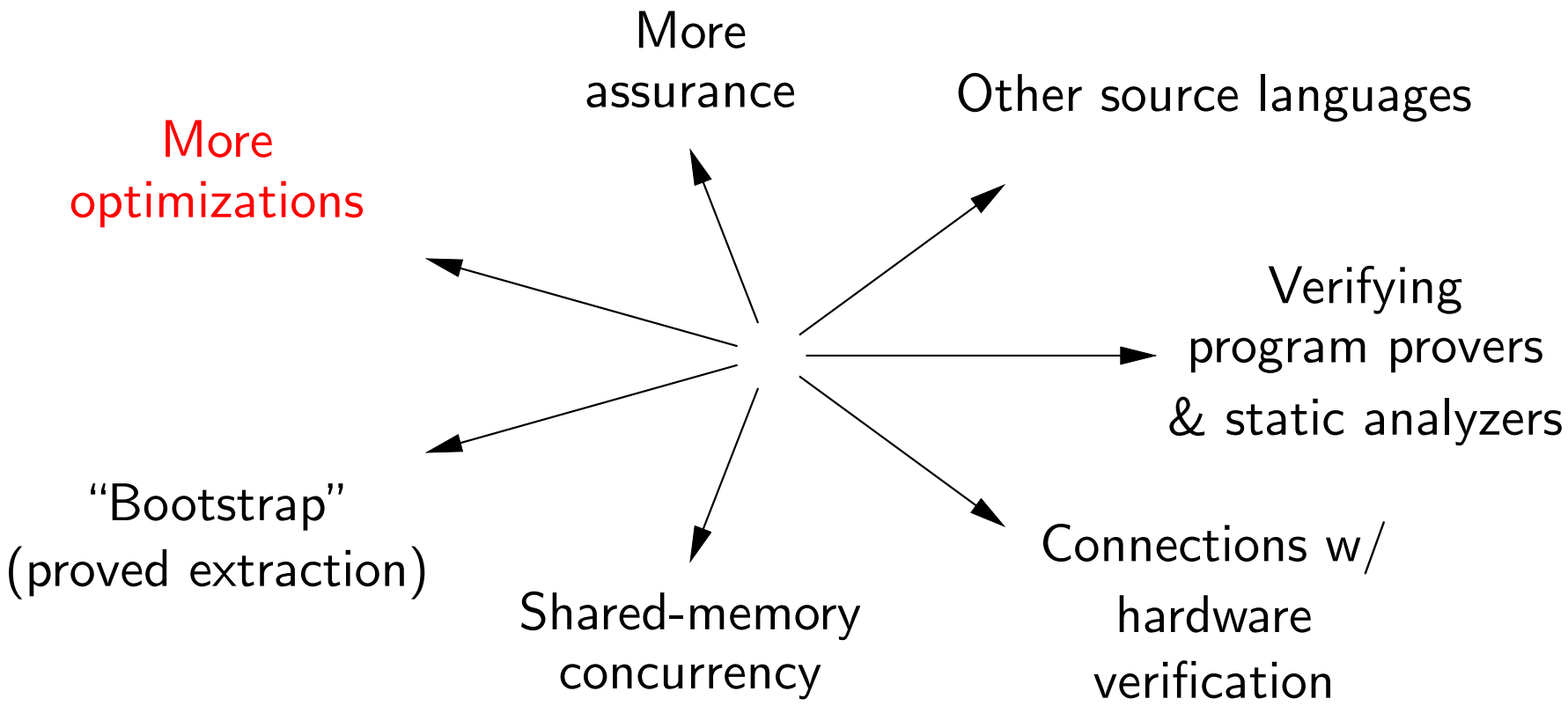
Other source languages besides C:
functional (Mini-ML); OO (fragments of C++); Spark Ada?; reactive?
Problem: verifying run-time systems ($\rightarrow$ A. Tolmach's talk).

Some directions for future work



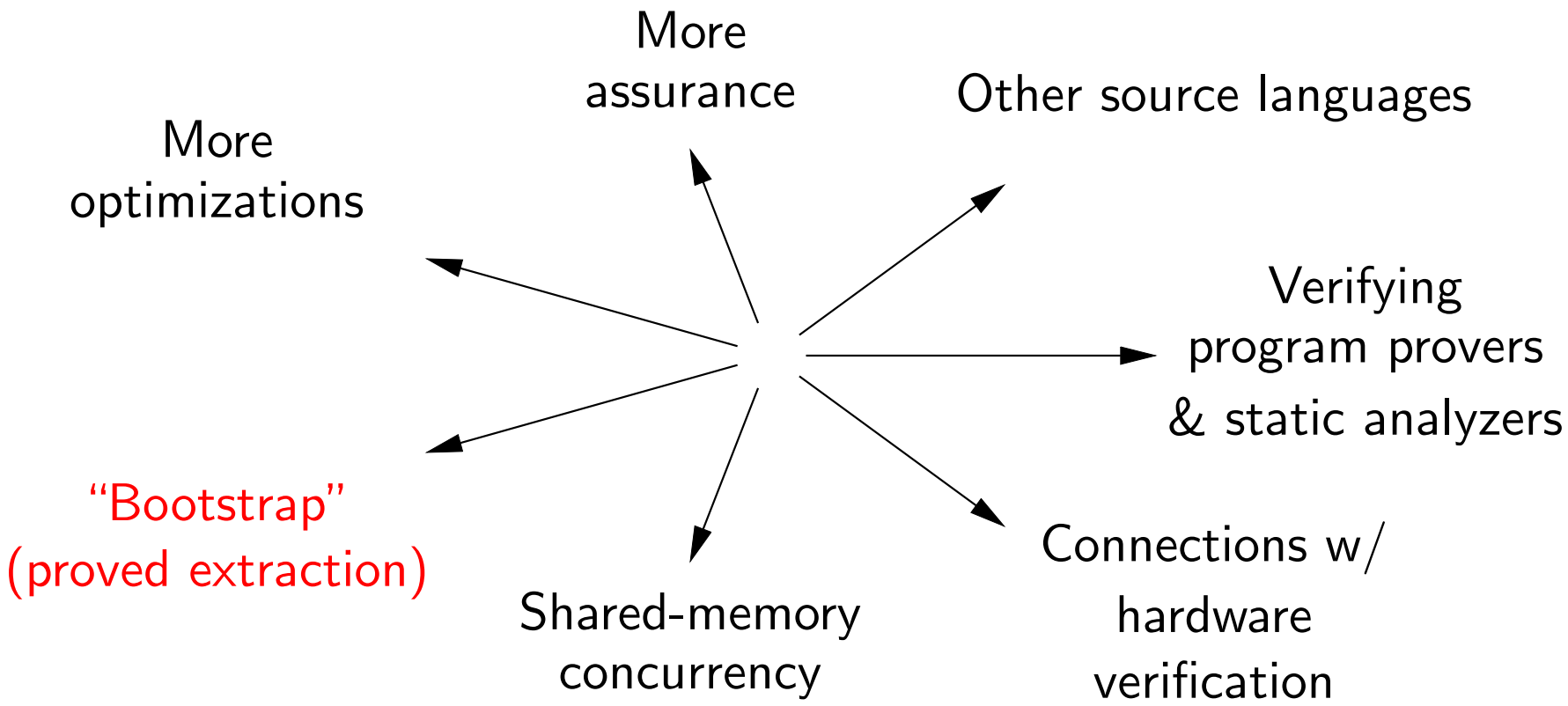
Prove or validate more of the TCB:
parsing, typing, elaboration, assembling, linking, ...

Some directions for future work



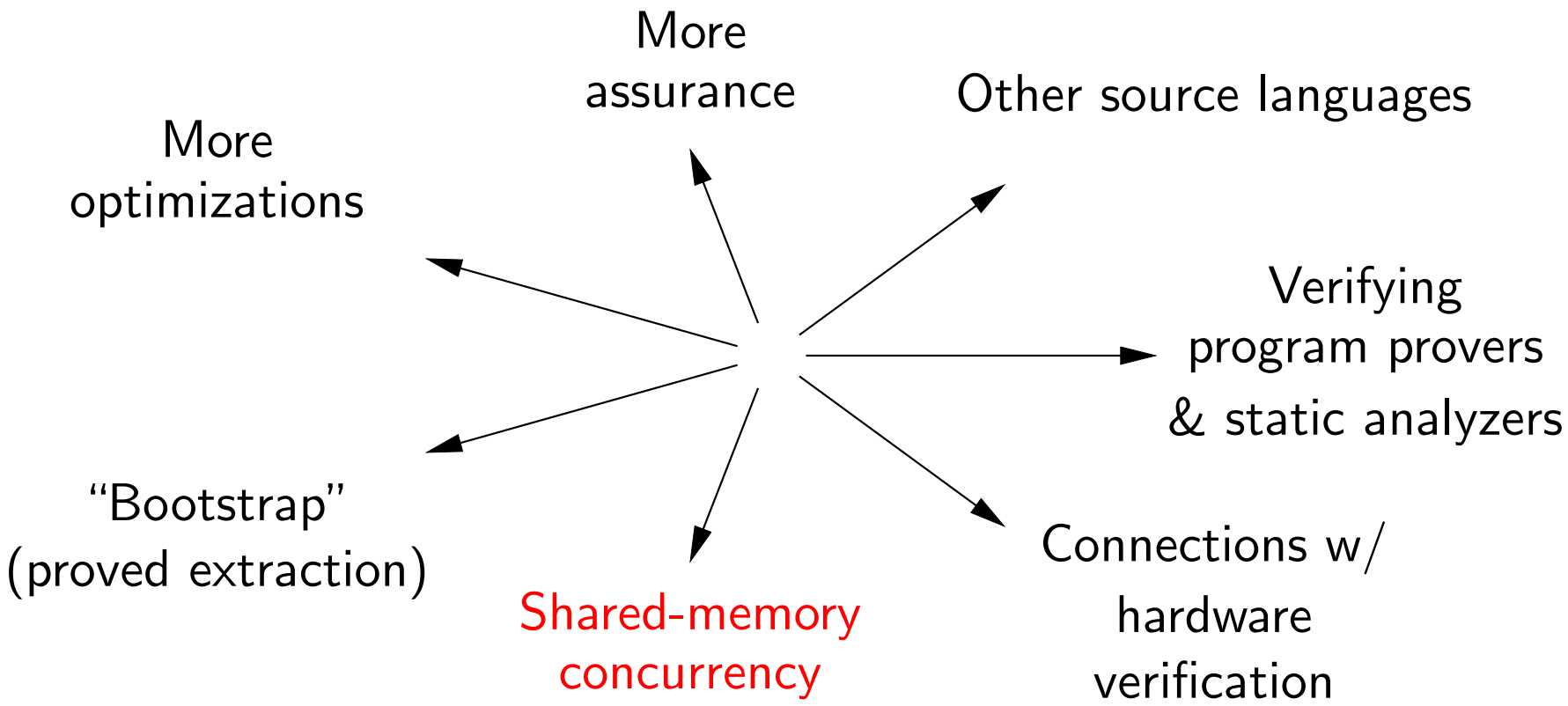
Add advanced optimizations, esp. loop optimizations.
Verified validation as the approach of least resistance.

Some directions for future work



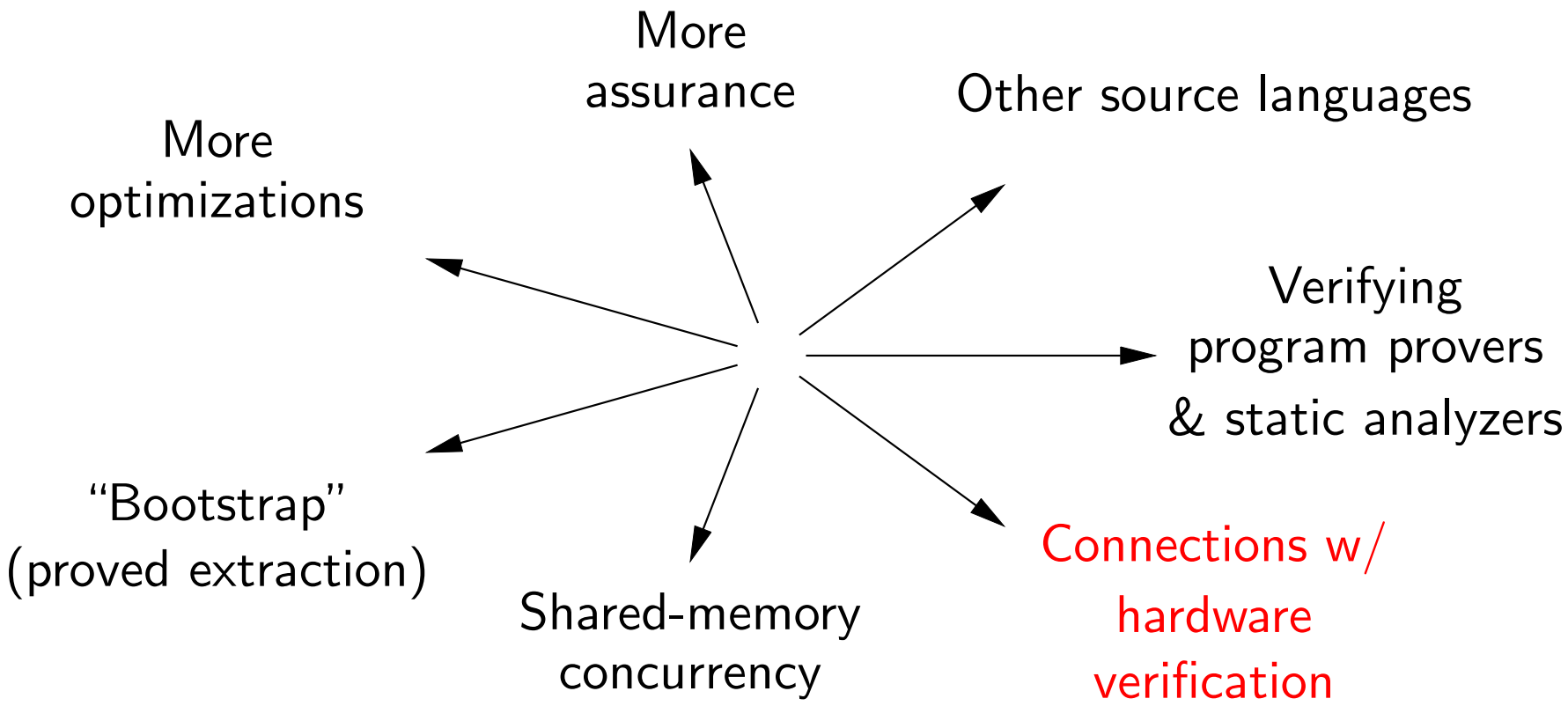
Increase confidence in the tools used to build CompCert:
Coq's extraction facility + the Caml compiler.

Some directions for future work



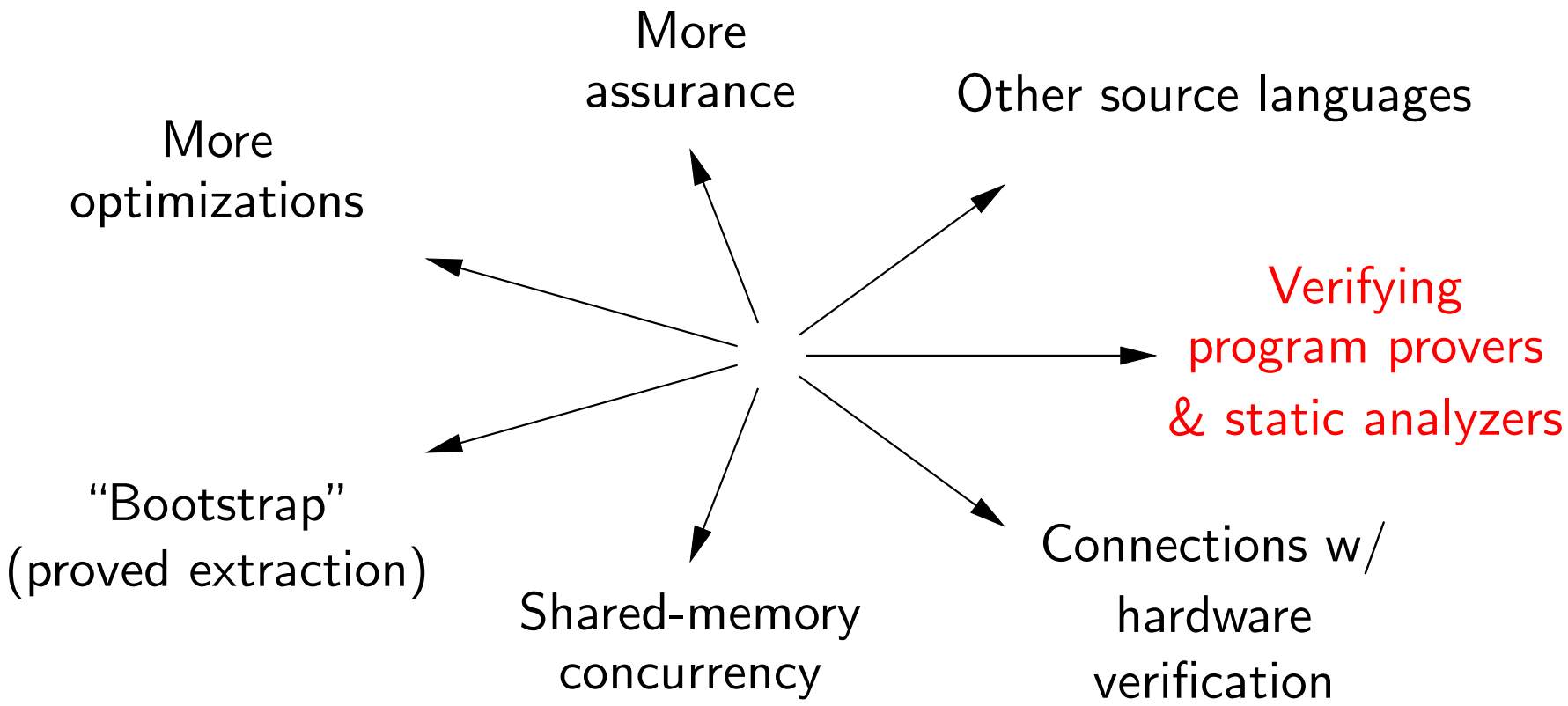
Race-free programs + concurrent separation logic ($\rightarrow$ A. Appel's talk)
or: racy programs + hardware memory models (P. Sewell et al).

Some directions for future work



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

Some directions for future work



Discussed next.

For more information

http://compcert.inria.fr/

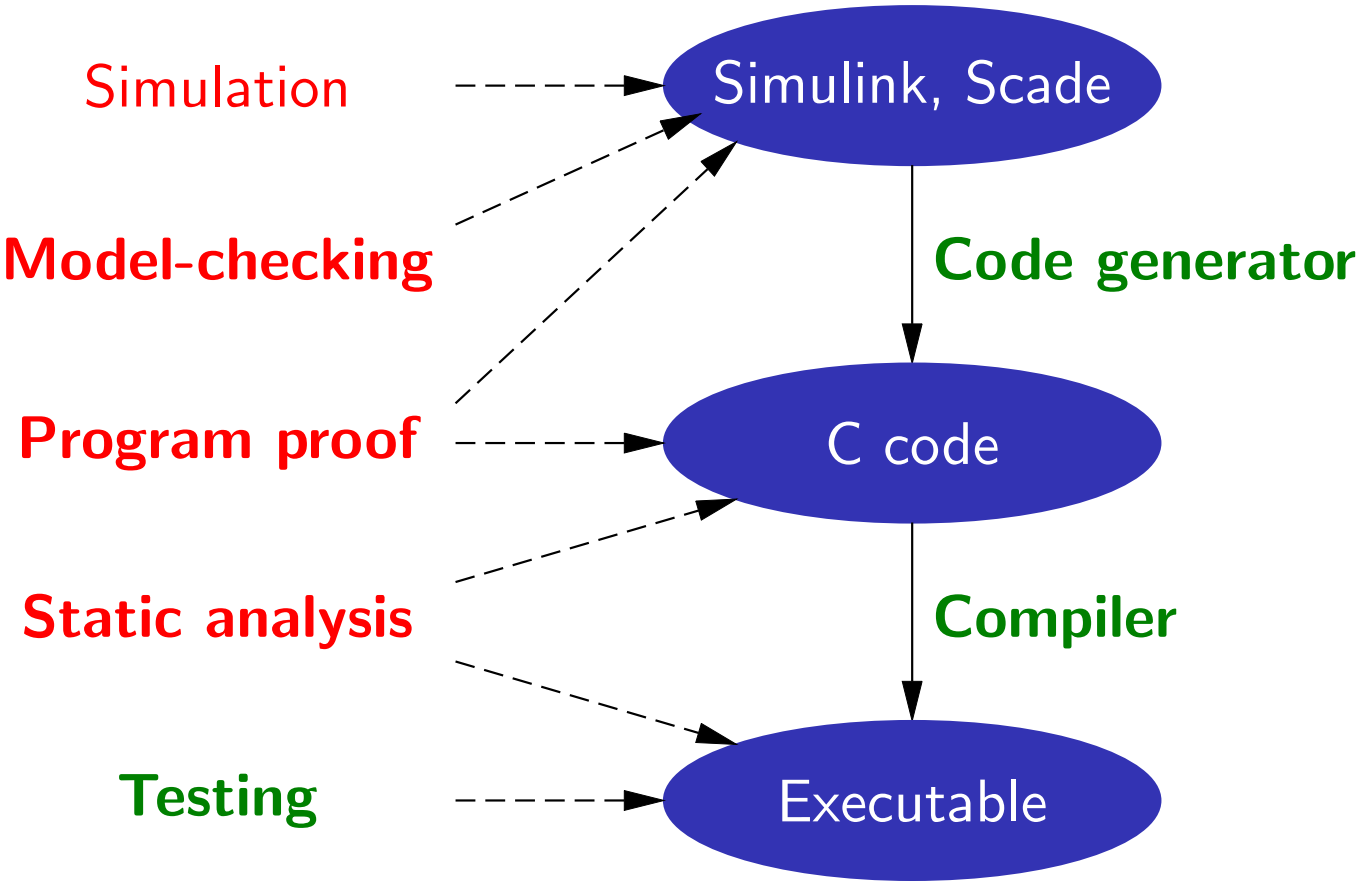
Research papers.

Complete source & proofs available for evaluation and research purposes.

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

Can you trust your verification tools?

Trust in formal verification (again)



Requirements on verification tools

(Static analyzers, program provers, model-checkers)

When used as sophisticated bug-finders: everything goes!

> *[Coverity] did not verify the absence of errors but rather tried to find as many of them as possible. Unsoundness let us focus on handling the easiest cases first, scaling up as it proved useful. [...] Circa 2000, unsoundness was controversial in the research community, though it has since become almost a de facto tool bias for commercial products and many research projects.*

When used to establish properties of programs and remove the corresponding tests: evidence of soundness is required.

Verified verification tools: ongoing work

Deductive verification: (program proof)

- Embeddings of Hoare logics and separation logics in HO logic. ($\rightarrow$ A. Appel's talk, J. Andronick's talk)
- Correctness proofs for verification condition generators.
- Automated theorem provers that are proved correct or generate independently-checkable certificates.

Static analysis:

- Verification of specialized analyzers, e.g. the JVM bytecode verifier.
- Verification of generic abstract interpreters.

Abstract interpretation for dummies

Execute ("interpret") the program using a non-standard semantics that:

- Computes over an abstract domain of the desired properties
  (e.g. "$x \in [n_1, n_2]$" for interval analysis)
  instead of concrete "things" like values and states.
- Handles boolean conditions, even if they cannot be resolved statically.
  (`THEN` and `ELSE` branches of `IF` are considered both taken.)
  (Loops execute arbitrarily many times.)
- Always terminates.

Example of abstract interpretation with intervals

$x \in [-\infty, \infty]$

```
IF x < 0 THEN
    x := 0;
ELSE IF x > 1000 THEN
    x := 1000;
ELSE
    SKIP;
ENDIF
```

Example of abstract interpretation with intervals

$x \in [-\infty, \infty]$

```
IF x < 0 THEN
    x := 0;
```

$x \in [0, 0]$

```
ELSE IF x > 1000 THEN
    x := 1000;
ELSE
    SKIP;
ENDIF
```

Example of abstract interpretation with intervals

```
                                  x ∈ [−∞, ∞]
IF x < 0 THEN
    x := 0;                       x ∈ [0, 0]
ELSE IF x > 1000 THEN
    x := 1000;                    x ∈ [1000, 1000]
ELSE
    SKIP;
ENDIF
```

Example of abstract interpretation with intervals

```
                                  x ∈ [−∞, ∞]
IF x < 0 THEN
    x := 0;                       x ∈ [0, 0]
ELSE IF x > 1000 THEN
    x := 1000;                    x ∈ [1000, 1000]
ELSE
    SKIP;                         x ∈ [0, ∞] ∩ [−∞, 1000] = [0, 1000]
ENDIF
```

Example of abstract interpretation with intervals

```
                                  x ∈ [−∞, ∞]
IF x < 0 THEN
    x := 0;                       x ∈ [0, 0]
ELSE IF x > 1000 THEN
    x := 1000;                    x ∈ [1000, 1000]
ELSE
    SKIP;                         x ∈ [0, ∞] ∩ [−∞, 1000] = [0, 1000]
ENDIF
                                  x ∈ [0, 0] ∪ [1000, 1000] ∪ [0, 1000] = [0, 1000]
```

Example of abstract interpretation with intervals

```
x := 0;                          x ∈ [0, 0]

WHILE x <= 1000 DO

    y := y + 1;

DONE
```

Example of abstract interpretation with intervals

```
x := 0;

WHILE x <= 1000 DO

    y := y + 1;

DONE
```

$x \in [0, 0]$

$x \in [0, 0] \cap [-\infty, 1000] = [0, 0]$

$x \in [1, 1]$

Example of abstract interpretation with intervals

```
x := 0;

WHILE x <= 1000 DO

    y := y + 1;

DONE
```

$x \in [0, 0]$

$x \in ([0, 0] \cup [1, 1]) \cap [-\infty, 1000] = [0, 1]$

$x \in [1, 2]$

Example of abstract interpretation with intervals

```
x := 0;

WHILE x <= 1000 DO

    y := y + 1;

DONE
```

$x \in [0, 0]$

$x \in ([0, 0] \cup [1, 2]) \cap [-\infty, 1000] = [0, 2]$

$x \in [1, 3]$

Example of abstract interpretation with intervals

```
x := 0;                          x ∈ [0, 0]

WHILE x <= 1000 DO
                                 x ∈ [0, ∞]
    y := y + 1;
                                 x ∈ [1, ∞]
DONE
```

Widening heuristic to accelerate convergence

Example of abstract interpretation with intervals

```
x := 0;

WHILE x <= 1000 DO

    y := y + 1;

DONE
```

$x \in [0, 0]$

$x \in ([0, 0] \cup [1, \infty]) \cap [-\infty, 1000] = [0, 1000]$

$x \in [1, 1001]$

Example of abstract interpretation with intervals

```
x := 0;

WHILE x <= 1000 DO

    y := y + 1;

DONE
```

$x \in [0, 0]$

$x \in ([0, 0] \cup [1, 1001]) \cap [-\infty, 1000] = [0, 1000]$

$x \in [1, 1001]$

Fixpoint reached!

Example of abstract interpretation with intervals

```
x := 0;

WHILE x <= 1000 DO

    y := y + 1;

DONE
```

$x \in [0, 0]$

$x \in ([0, 0] \cup [1, 1001]) \cap [-\infty, 1000] = [0, 1000]$

$x \in [1, 1001]$

$x \in [1001, \infty] \cap [1, 1001] = [1001, 1001]$

Fixpoint reached!

Anatomy of a static analyzer based on abstract interp.

(e.g. Astrée)

A generic, domain-independent abstract interpreter,
parameterized by a hierarchy of language-independent abstract domains:

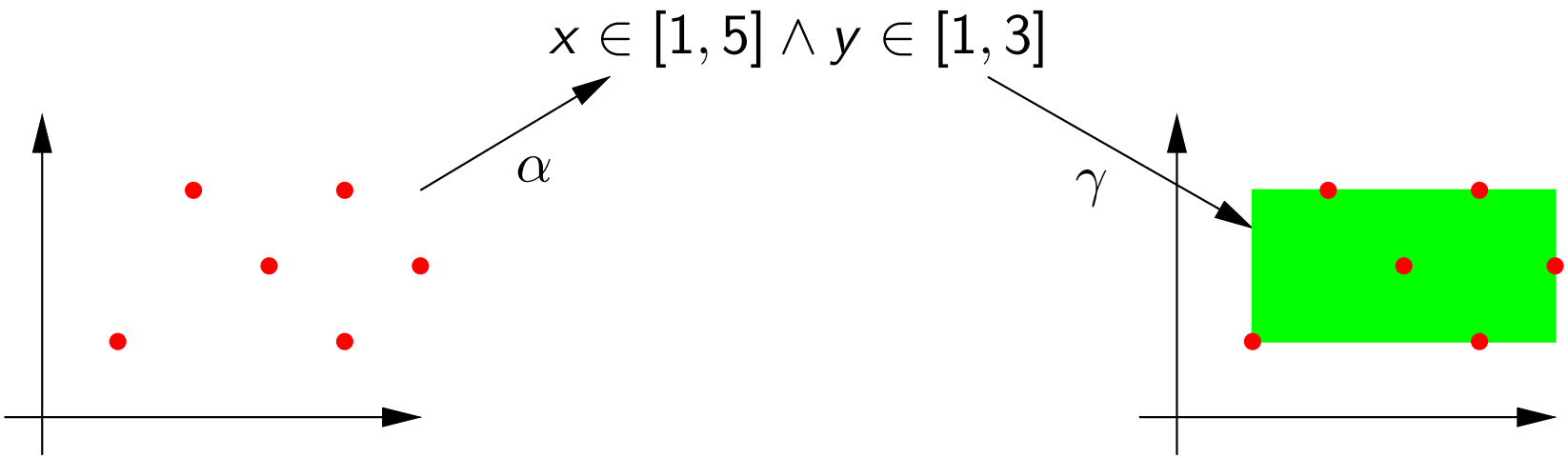
- Non-relational domains (properties of a single variable):
  intervals $x \in [c_1, c_2]$
  congruences $x = c_1 \pmod{c_2}$
- Relational domains (properties of several variables):
  polyhedra $\sum c_i x_i \leq c$
  octagons $\sum c_i x_i \leq c$ with $c_i \in \{-1, 0, 1\}$
  memory and pointer domains (points-to and non-aliasing properties)
- Combinations of domains:
  trace partitioning
  reduced product.

To be proved: the generic interpreter (easy) + each domain (difficult).

Orthodox presentation: Galois connections

Define a lattice $\mathcal{A}, \leq$ of abstract states and two functions:

- Abstraction function α : sets of concrete states $\rightarrow$ abstract state
- Concretization function γ : abstract state $\rightarrow$ sets of concrete states



To be proved: α and γ monotonic; $X \subseteq \gamma(\alpha(X))$; and $x \leq \alpha(\gamma(x))$.

Orthodox presentation: calculating abstract operators

For each operation of the language, calculate its abstract counterpart (operating on elements of $\mathcal{A}$ instead of concrete values and states).

Example: for the $+$ operator in expressions,

$$a_1 +^{\#} a_2 = \alpha\{n_1 + n_2 \mid n_1 \in \gamma(a_1), n_2 \in \gamma(a_2)\}$$

(. . . long calculations omitted . . .)

$$[l_1, u_1] +^{\#} [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$$

$+^{\#}$ is sound and optimally precise by construction.

Towards a mechanized proof of correctness

Problems with the orthodox approach:

- α functions do not always exist
- α functions are not computable
- Poor theorem proving support for calculational reasoning.

Towards a mechanized proof of correctness

(D. Pichardie, 2005, 2008; D. Cachera and D. Pichardie, 2010)

Solution 1: forget about α; use relations:

$$\vdash \textit{concrete-thing} \in \textit{abstract-thing}$$

Solution 2: use calculations on paper and re-prove resulting definitions:

$$\vdash v_1 \in A_1 \ \wedge \ \vdash v_2 \in A_2 \implies \vdash v_1 + v_2 \in A_1 +^{\#} A_2$$

Much heroic proof work remains:

- Many, many abstract operations to prove.
- Modular construction of abstract domains.
- Fixpoints, with widening and narrowing.
- Their termination.

Worked out for simple non-relational domains, up to intervals.

Going further by cutting more corners

Forget about mechanically proving termination.

(Coq + classical logic can reason over partial functions.)

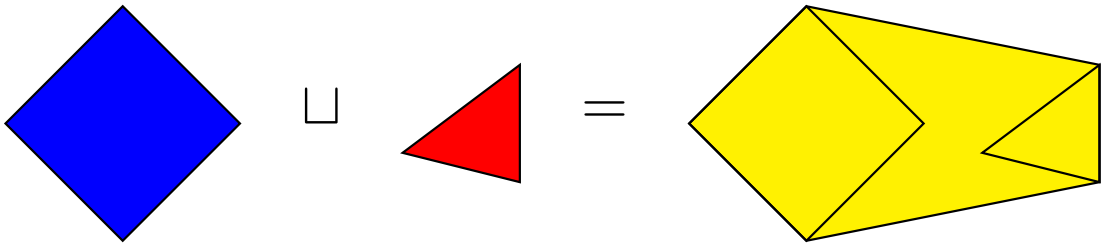
Validate *a posteriori* untrusted implementations of relational domains.

(with the help of validated decision procedures.)

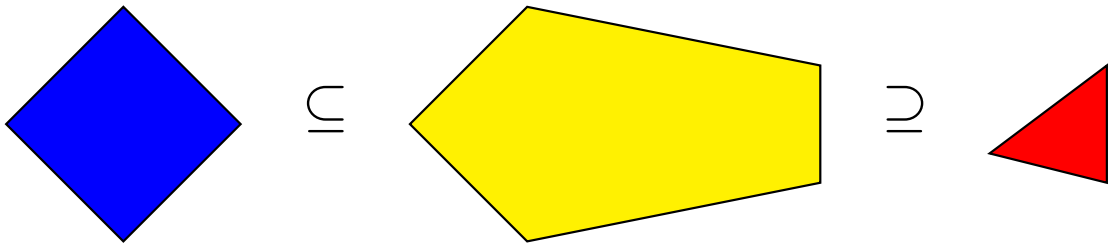
Verified validation of the polyhedra abstract domain

(D. Pichardie et al, 2010)

Computing joins $\approx$ convex hull algorithm.



Checking a posteriori the result of a join operation = inclusion checking.



Verified validation of the polyhedra abstract domain

(D. Pichardie et al, 2010)

Checking inclusion between two polyhedra:

$$\underbrace{A_1 \wedge \cdots \wedge A_n}_{\text{linear inequations}} \Longrightarrow \underbrace{B_1 \wedge \cdots \wedge B_m}_{\text{linear inequations}}$$

This amounts to checking that the formulas

$$\underbrace{A_1 \wedge \cdots \wedge A_n \wedge \neg B_i}_{\text{linear inequations}} \quad \text{for } i = 1, \ldots, m$$

are not satisfiable, using a decision procedure for Presburger arithmetic.

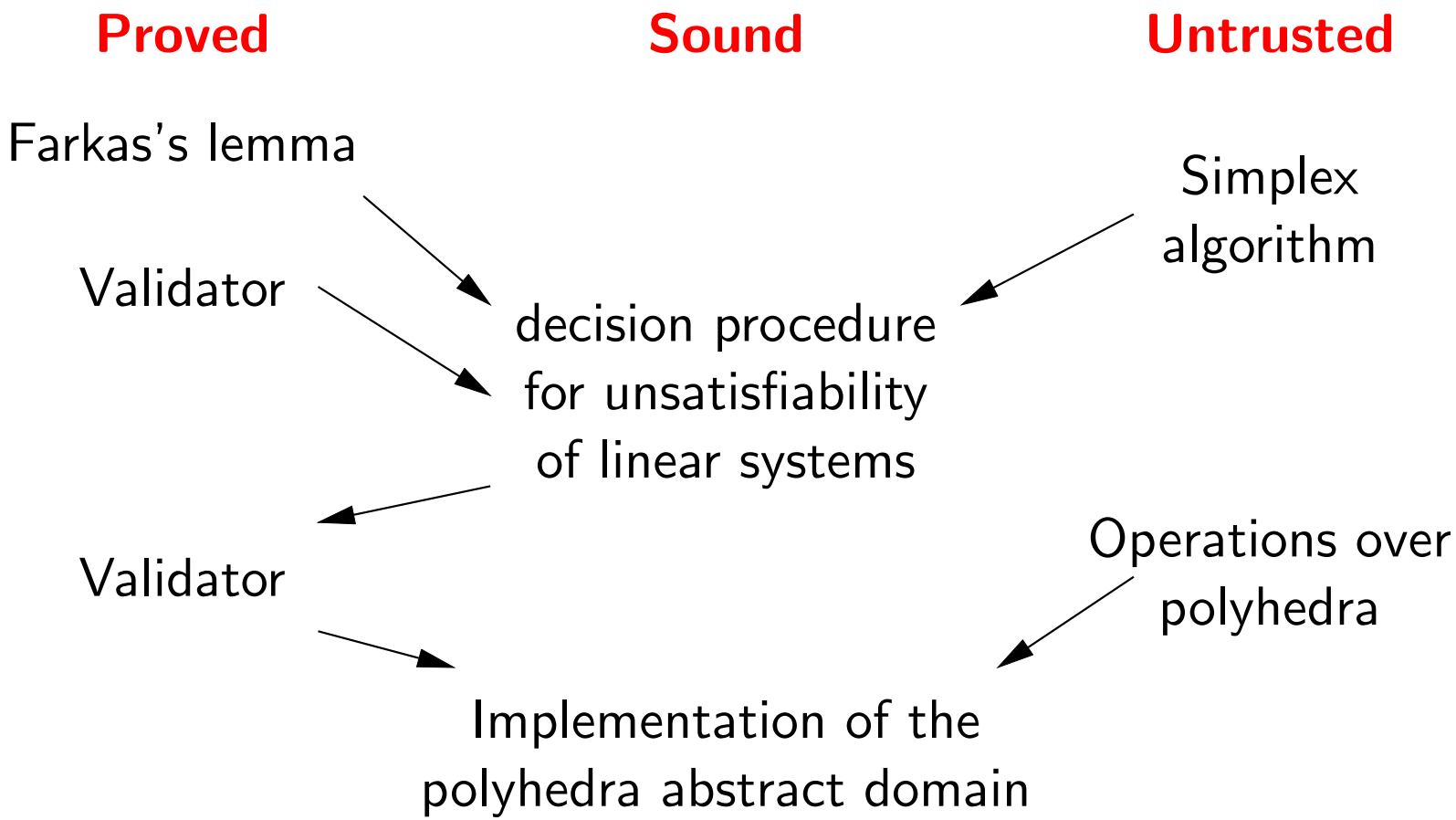
Verified validation of the decision procedure

Lemma (Farkas)

A conjunction of linear inequations $\sum_j a_{ij} x_j \leq c_i$ *for* $i = 1, \ldots, n$ *is unsatisfiable if and only if there exists coefficients* $f_1, \ldots, f_n$ *such that* $\sum_i f_i a_{ij} = 0$ *for all* j *and* $\sum_i f_i c_i < 0$.

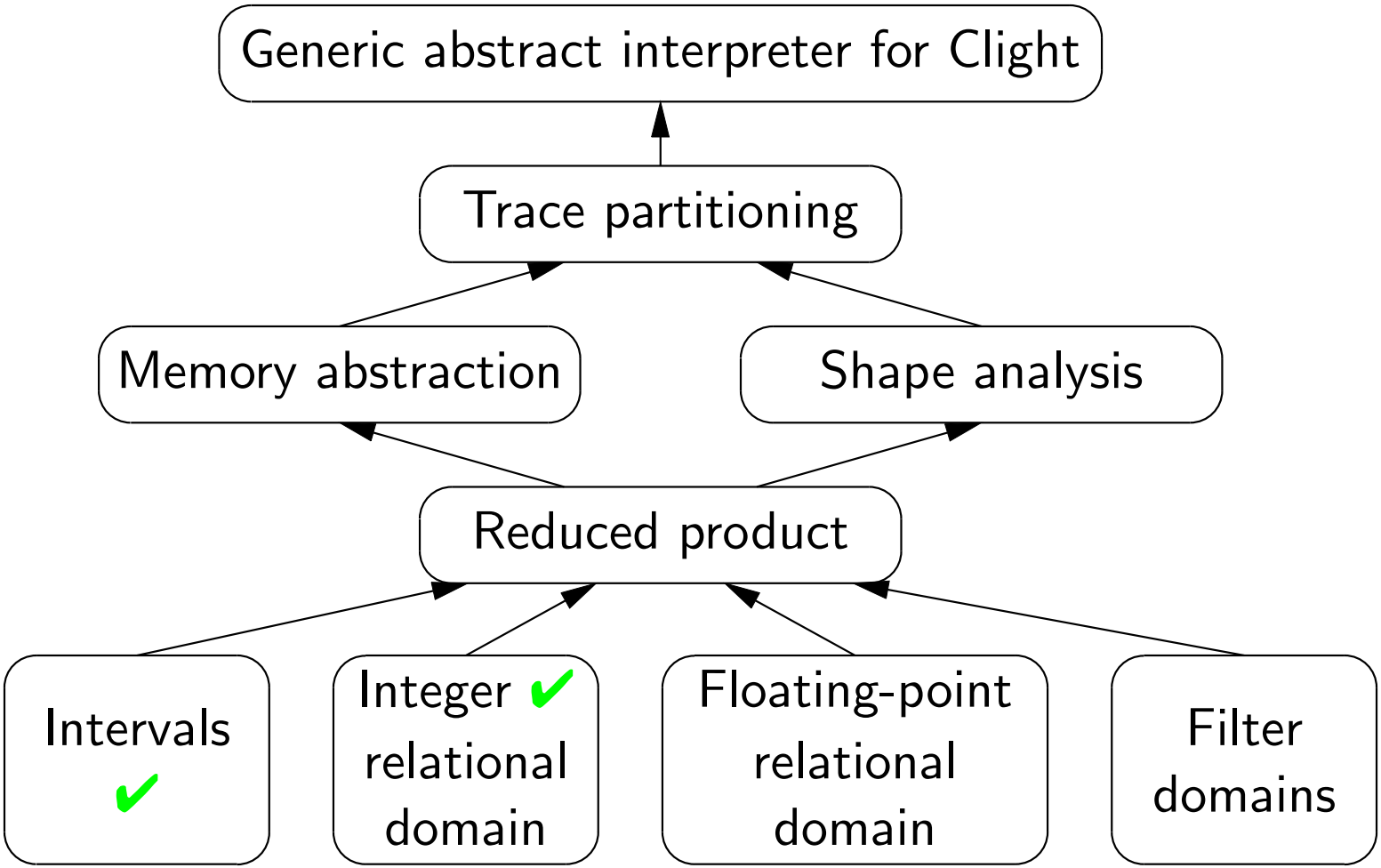
The coefficients f_i act as a certificate and can be computed by (an untrusted implementation of) the simplex algorithm.

Two-level verified validation



Future work

Scale these approaches (verified implementations or verified validation) all the way to an Astrée-like static analyzer.



In closing...

Perspectives

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

The formal verification of development and verification tools for critical software

- appears within reach,
- raises fascinating verification issues,
- and could have practical impact.

Main challenges:

- scaling up all the way to real-world usage;
- taking advantage of tool verification for DO-178-like qualification.