

Trust in programming tools: the formal verification of compilers and static analysers

Xavier Leroy

Inria Paris

Verified trustworthy software systems, April 2016



Tool-assisted formal verification

Old, fundamental ideas...

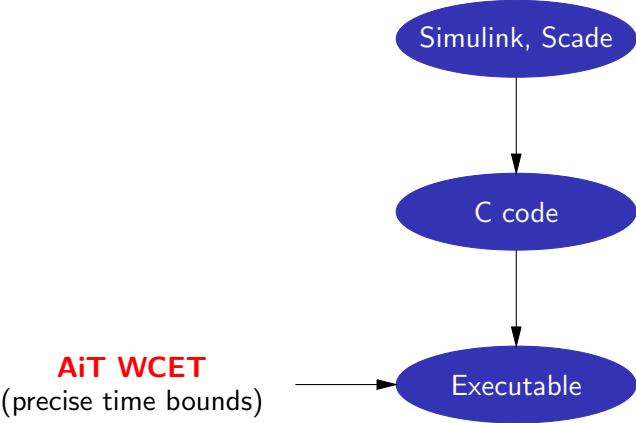
(Hoare logic, 1960's; model checking, abstract interpretation, 1970's)

that remained theoretical for a long time...

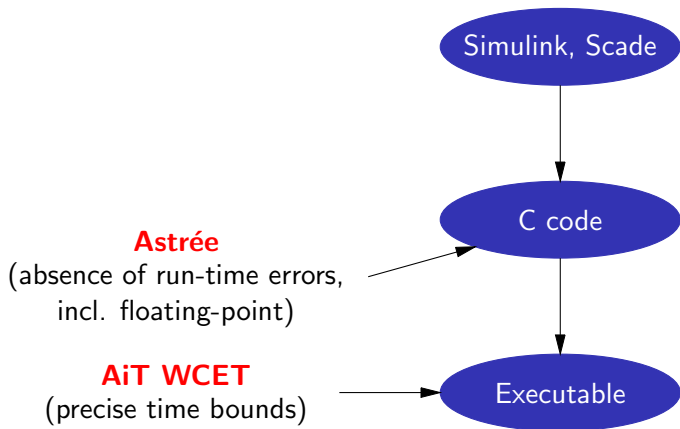
are now implemented and automated in verification tools...

usable and sometimes used in the critical software industry.

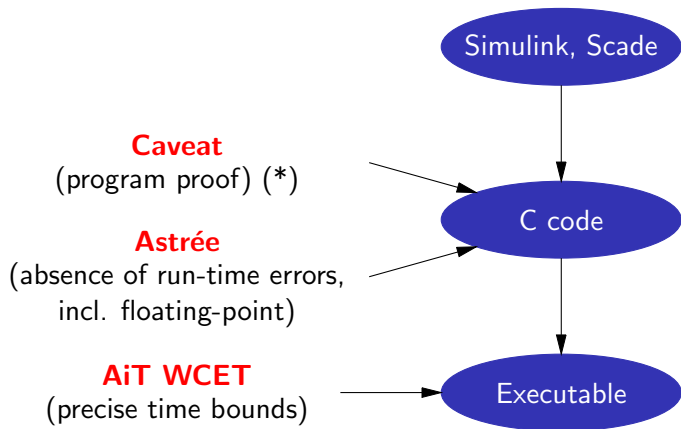
Examples of uses for avionics software



Examples of uses for avionics software



Examples of uses for avionics software



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

Examples of uses for avionics software

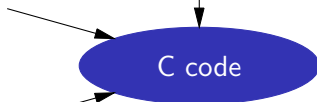
Rockwell-Collins toolchain

(model-checking + proof)



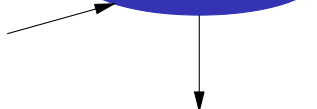
Caveat

(program proof) (*)



Astrée

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



AiT WCET

(precise time bounds)

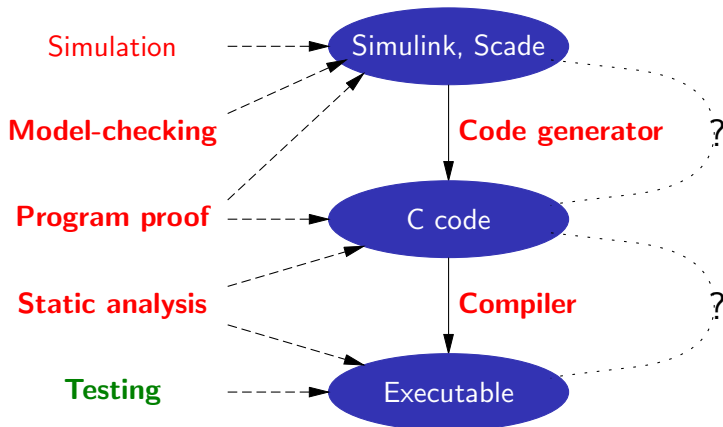


(*) Motto: "unit proofs as a replacement for unit tests"

Trust in tools

that participate in
the production and verification of critical software

Trust in formal verification



The unsoundness risk: Are verification tools semantically sound?

The miscompilation risk: Are compilers semantics-preserving?

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

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 with a good compiler, then 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;
}

```

```
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;
```

```

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;

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;
}

```

Formal verification of tools

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.

John McCarthy
James Painter¹

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS²

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

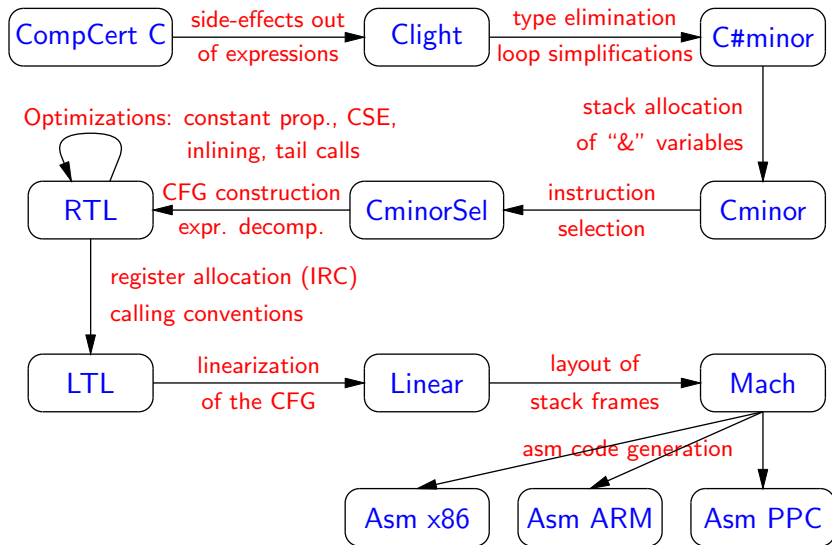
(X.Leroy, S.Blazy, et al)

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
⇒ careful code generation; some optimizations.

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

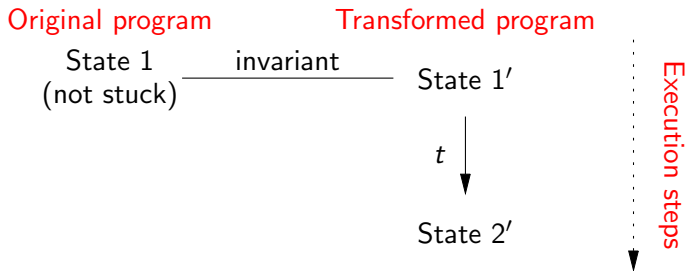
The formally verified part of the compiler



Formally verified using Coq

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

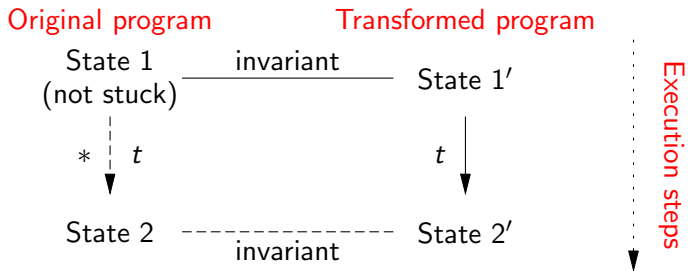
Proof pattern: simulation/refinement diagrams such as:



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:



Formally verified using Coq

As a consequence, the observable behavior of the compiled code (trace of I/O operations) is identical to one of the possible behaviors of the source code, or improves over one:

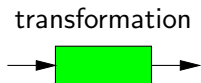
Source code:	$i_1.o_1.o_2.i_2.o_3$	$i_1.o_1.\dagger$ undefined behavior
Compiled code:	$i_1.o_1.o_2.i_2.o_3$	$i_1.o_1.o_2\dots$
	(same behavior)	(“improved” undefined behavior)

Theorem `transf_c_program_preservation`:

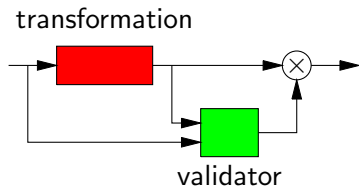
```
forall p tp beh,  
transf_c_program p = OK tp ->  
program_behaves (Asm.semantics tp) beh ->  
exists beh', program_behaves (Csem.semantics p) beh'  
  /\ behavior_improves beh' beh.
```

Compiler verification patterns (for each pass)

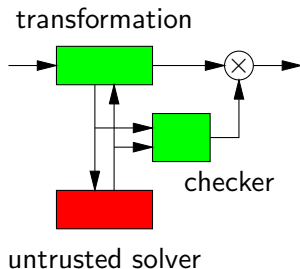
Verified transformation




Verified translation validation



External solver with verified validation



 = formally verified

 = not verified

Programmed (mostly) in Coq

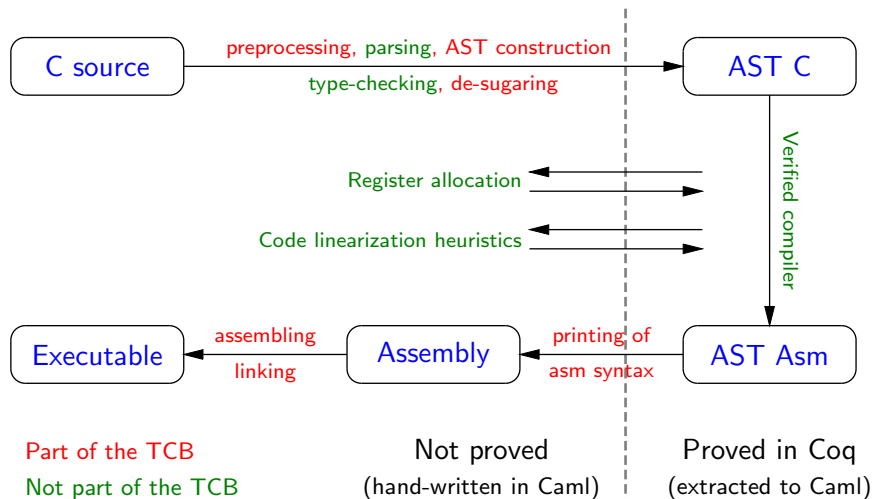
All the verified parts of the compiler are programmed directly in Coq's specification language, using pure functional style.

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

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

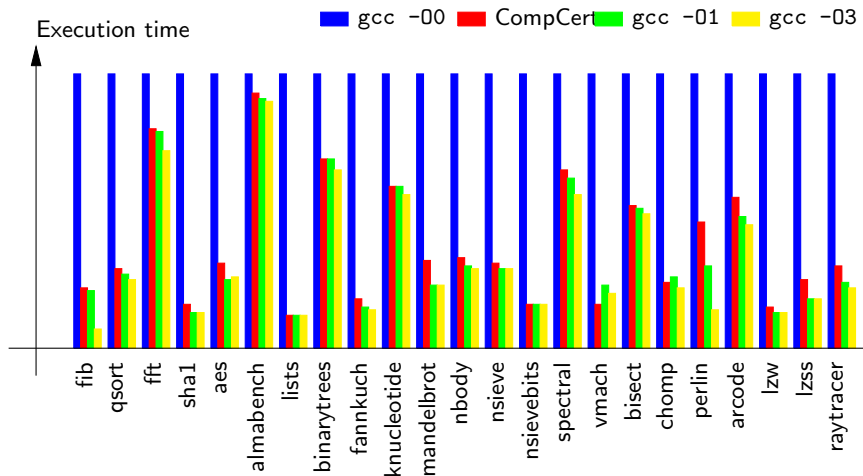
Claim: purely functional programming is the shortest path to writing and proving a program.

The whole Compcert compiler



Performance of generated code

(On a Power 7 processor)



A tangible increase in quality

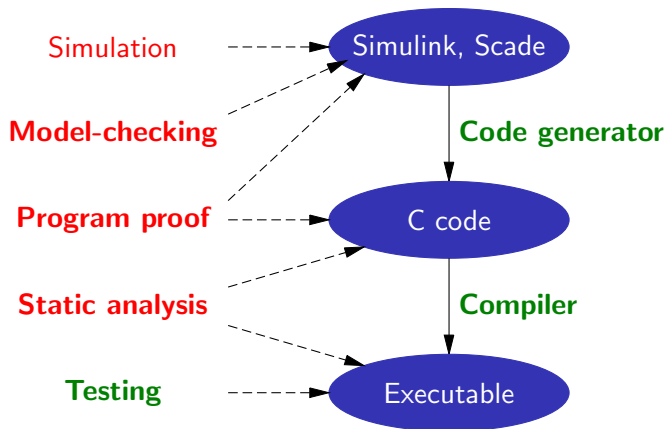
The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have 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

Verasco:

a formally-verified static analyzer
based on abstract interpretation

Trust in formal verification (again)



The unsoundness risk: Are verification tools semantically sound?

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.

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

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 the **concrete domain** of 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

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

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

Example of abstract interpretation with intervals

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

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

$x \in [0, 0]$

Example of abstract interpretation with intervals

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

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

$x \in [0, 0]$

$x \in [1000, 1000]$

Example of abstract interpretation with intervals

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

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

$$x \in [0, 0]$$

$$x \in [1000, 1000]$$

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

Example of abstract interpretation with intervals

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

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

$$x \in [0, 0]$$

$$x \in [1000, 1000]$$

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

$$x \in [0, 0] \cup [1000, 1000] \cup [0, 1000] = [0, 1000]$$

Example of abstract interpretation with intervals

```
x := 0;
```

$x \in [0, 0]$

```
WHILE x <= 1000 DO
```

```
    x := x + 1;
```

```
DONE
```

Example of abstract interpretation with intervals

`x := 0;`

$x \in [0, 0]$

`WHILE x <= 1000 DO`

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

`x := x + 1;`

$x \in [1, 1]$

`DONE`

Example of abstract interpretation with intervals

`x := 0;`

$x \in [0, 0]$

`WHILE x <= 1000 DO`

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

`x := x + 1;`

$x \in [1, 2]$

`DONE`

Example of abstract interpretation with intervals

`x := 0;`

$x \in [0, 0]$

`WHILE x <= 1000 DO`

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

`x := x + 1;`

$x \in [1, 3]$

`DONE`

Example of abstract interpretation with intervals

`x := 0;`

$x \in [0, 0]$

`WHILE x <= 1000 DO`

$x \in [0, \infty]$

`x := x + 1;`

$x \in [1, \infty]$

`DONE`

Widening heuristic to accelerate convergence

Example of abstract interpretation with intervals

`x := 0;`

$x \in [0, 0]$

`WHILE x <= 1000 DO`

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

`x := x + 1;`

$x \in [1, 1001]$

`DONE`

Narrowing iteration to improve the result

Example of abstract interpretation with intervals

`x := 0;`

$x \in [0, 0]$

`WHILE x <= 1000 DO`

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

`x := x + 1;`

$x \in [1, 1001]$

`DONE`

Fixpoint reached!

Example of abstract interpretation with intervals

`x := 0;`

$$x \in [0, 0]$$

`WHILE x <= 1000 DO`

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

`x := x + 1;`

$$x \in [1, 1001]$$

`DONE`

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

Fixpoint reached!

Properties inferred by Verasco

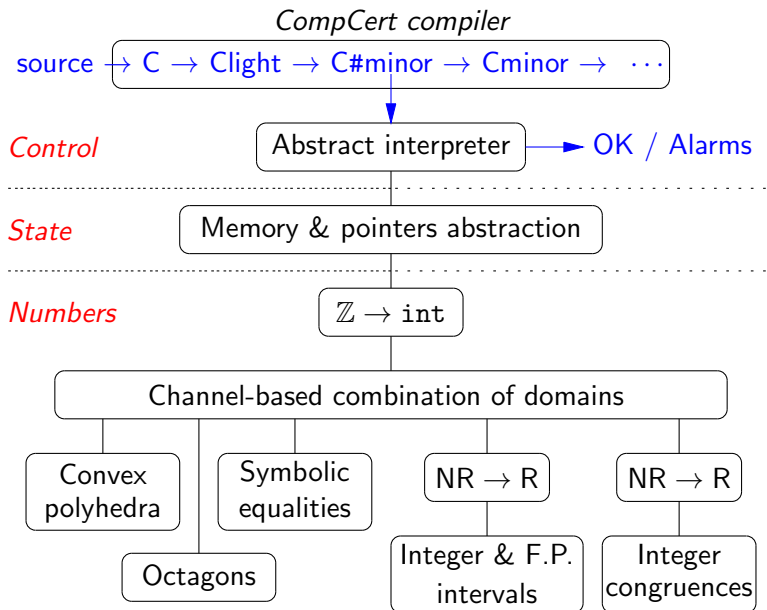
Properties of a single variable / memory cell: (value analysis)

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

Relations between variables: (relational analysis)

Polyhedra	$c_1x_1 + \dots + c_nx_n \leq c$
Octagons	$\pm x_1 \pm x_2 \leq c$
Symbolic equalities	$x = \text{expr}$

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).

Conclusions and future work

Conclusions

CompCert and especially Verasco are still ongoing projects.

However, they demonstrate that the formal verification of compilers, static analyzers, and related tools is feasible.

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

Future work

For verified compilers:

- Other source languages: functional, reactive.
- More optimizations, esp. loop optimizations.
- Increase confidence even further.

For static analyzers based on abstract interpretation:

- Algorithmic efficiency.
- More advanced domains (e.g. shape analysis).

For both:

- Separate compilation / modular static analysis.
- Support for shared-memory concurrency.

In closing...

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

Let's make this a reality!