Formally verifying a compiler: what does it mean, exactly?

Xavier Leroy

INRIA Paris

ICALP, 2016-07-13

Tool-assisted formal verification

Old, fundamental ideas. . .

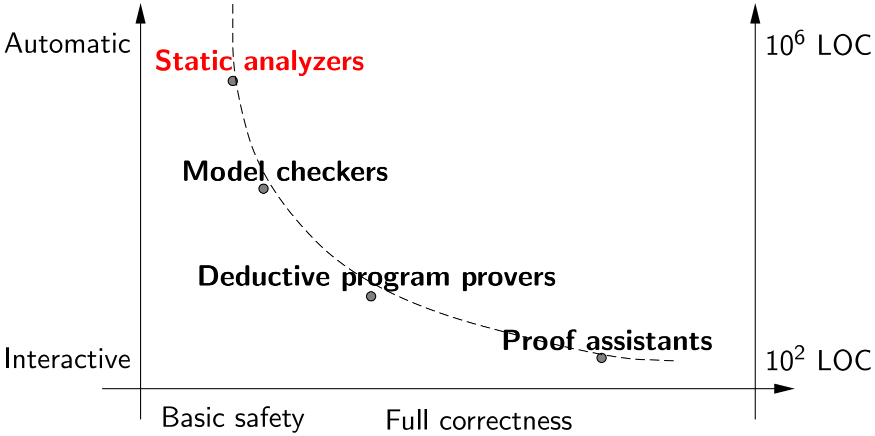
(Hoare logic, 1960's; model checking, abstract interpretation, 1970's)

that remained theoretical for a long time. . .

are now implemented and automated in verification tools. . .

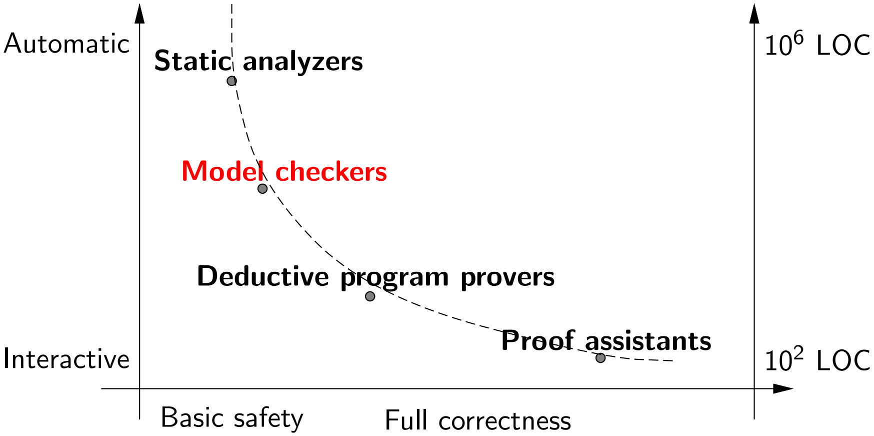
usable and sometimes used in the critical software industry.

A panorama of verification tools



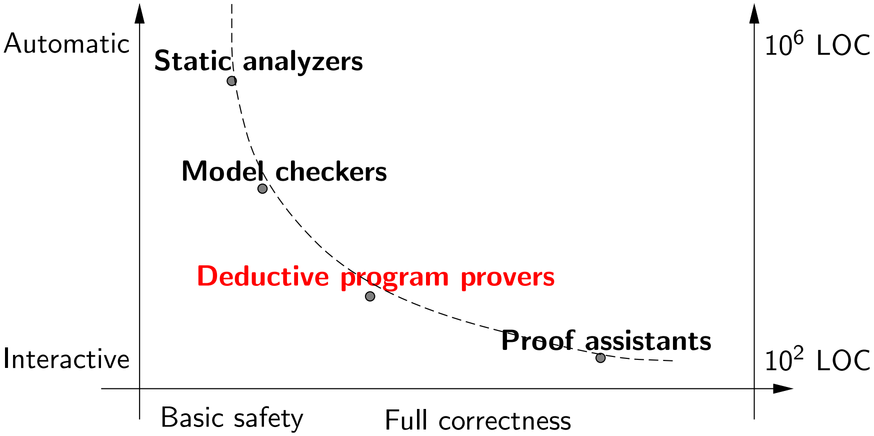
Static analysis: automatically infer simple properties of one variable ($x \in [N_1, N_2]$, $x \bmod N = 0$, etc) or several ($x + y \leq z$).

A panorama of verification tools



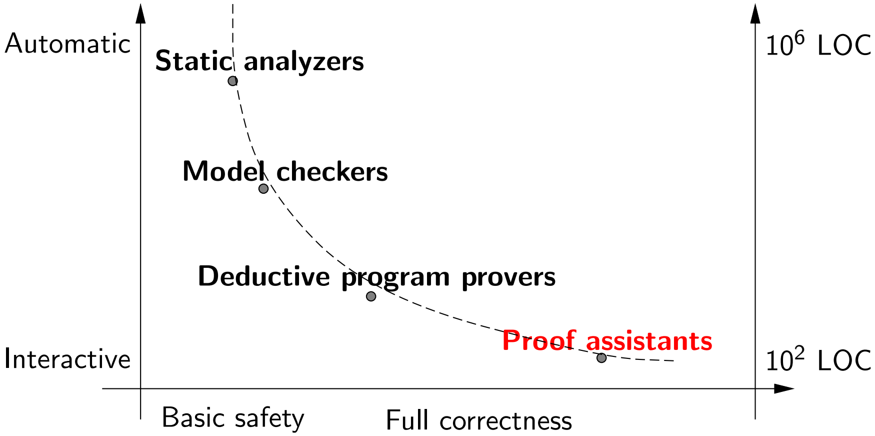
Model checking: automatically check that some “bad” program points are not reachable.

A panorama of verification tools



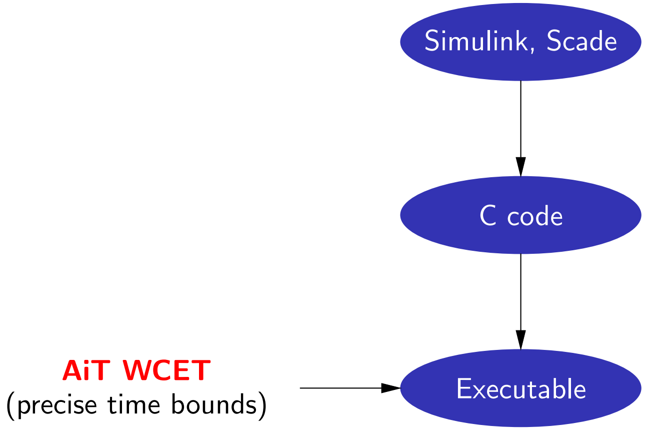
Program proof (Hoare logic, separation logic): show that
preconditions $\Rightarrow$ *invariants* $\Rightarrow$ *postconditions*
using automated theorem provers.

A panorama of verification tools

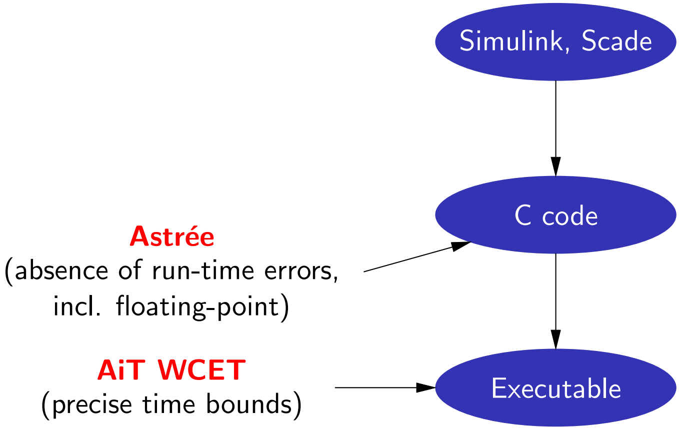


Proof assistants: conduct mathematical proofs in interaction with the user; re-check the proofs for correctness.

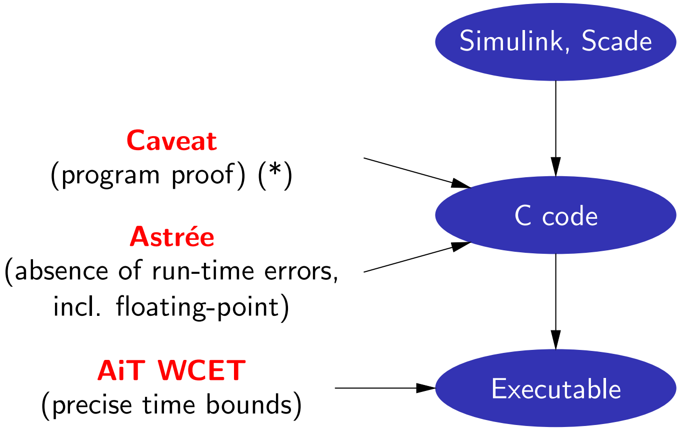
Examples of uses for avionics software



Examples of uses for avionics software

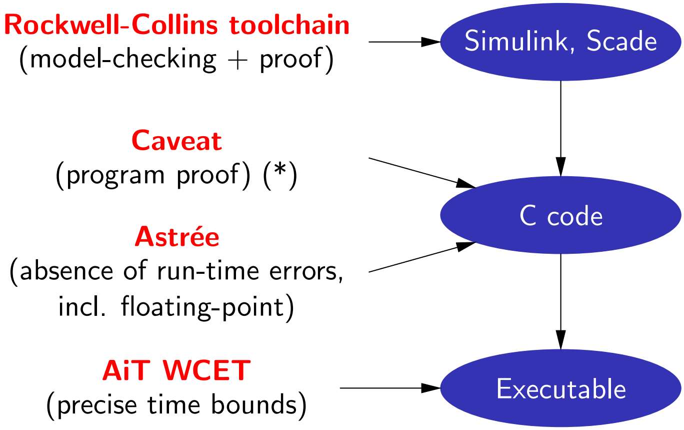


Examples of uses for avionics software



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

Examples of uses for avionics software

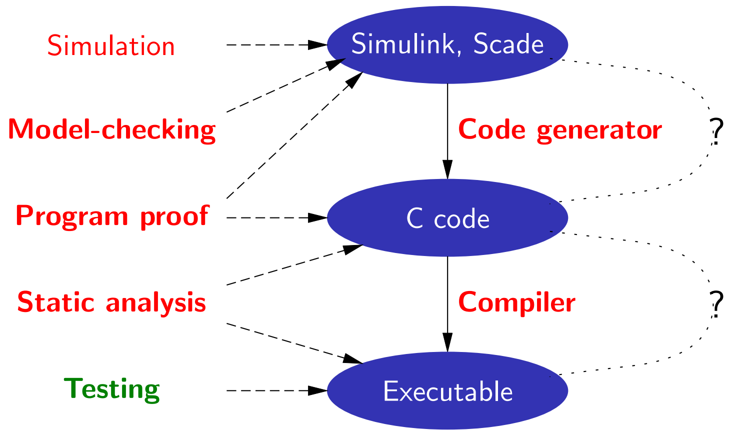


(*) Motto: “unit proofs as a replacement for unit tests”

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

Why is it so hard to compile correctly?

Misunderstandings of the definition of the source language.

(In particular, the C and C++ standards have many subtle points.)

Ambitious optimizations to try to increase performance.

(Example next.)

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 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)

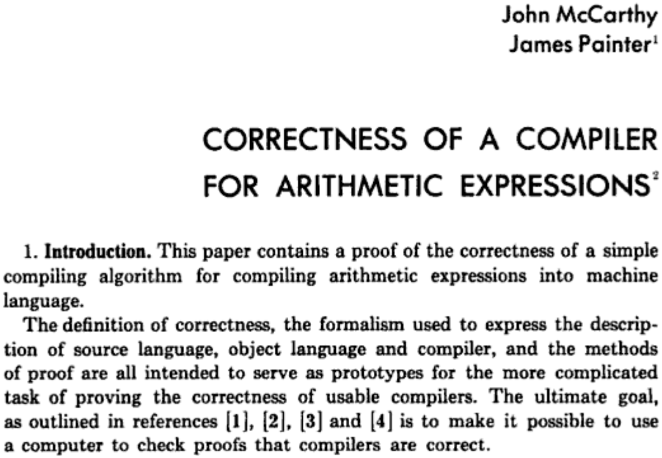
After all, these tools have simple specifications:

> *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.*

An old idea. . .



John McCarthy
James Painter[1]

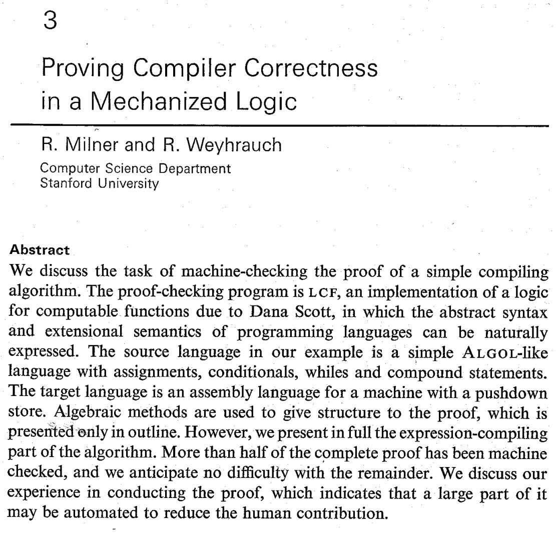
CORRECTNESS OF A COMPILER
FOR ARITHMETIC EXPRESSIONS[2]

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

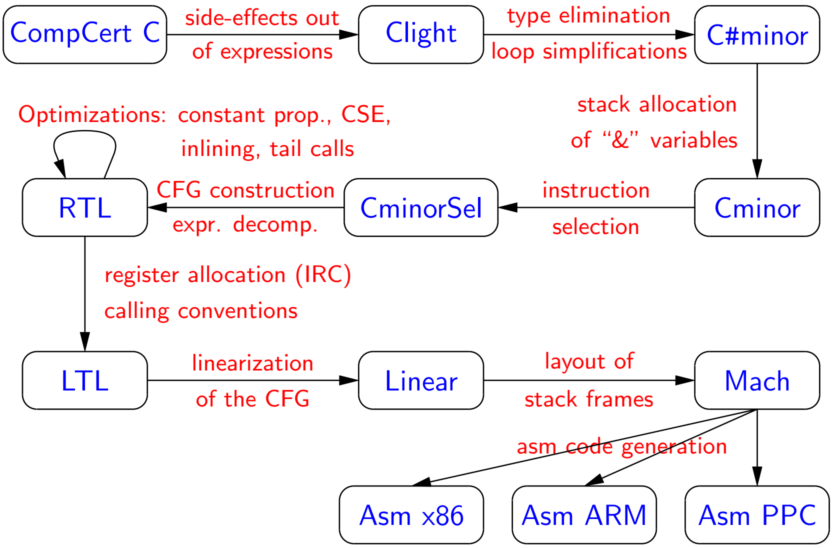
The CompCert project
(X. Leroy, S. Blazy, et al)

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.

```
Theorem transf_c_program_correct:
  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.
```

A fairly large proof: 60 000 lines, 6 person.years, low automation.

Programmed (mostly) in Coq

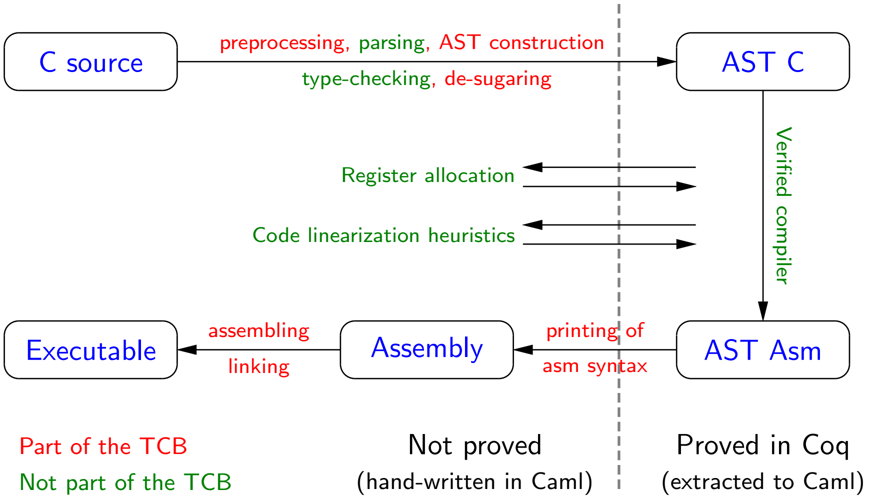
All the verified parts of the compiler are programmed directly in Coq's specification language, using pure functional style.

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

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

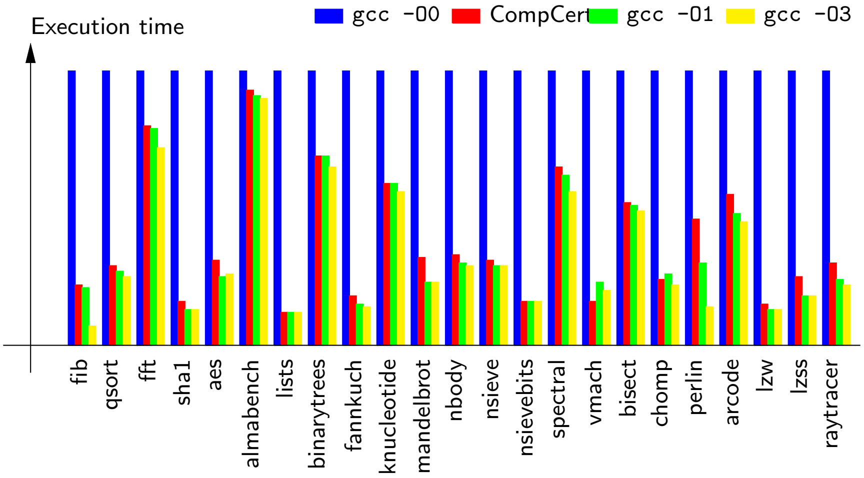
Claim: purely functional programming is the shortest path to writing and proving a program.

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

What have we proved, exactly?

Trusting a proof

Did we prove it right?

- Pencil & paper proof: proof reviews, social consensus.
- Mechanized proof: trust the proof assistant.
- Axioms used, if any.

Did we prove the right thing?

- Does the statement of the theorem say what we think it says?
- Are the definitions it uses correct?

Mathematical proofs

For some mathematical proofs, the final statement is immediately understandable:

```
Theorem Fermat's_last:
  forall (a b c n: nat),
  a >= 1 /\ b >= 1 /\ c >= 1 /\ n >= 3 ->
  power a n + power b n <> power c n.
```

The proof itself would involve higher mathematics, of course. But they do not show up in the statement of the final theorem.

For the skeptics. . .

We can even avoid fancy notations and provide all the definitions used in the statement of the theorem:

```
Inductive nat : Type := O | S (n: nat).
Fixpoint add (x y: nat) :=
  match x with O => y | S x' => S (add x' y) end.
Fixpoint mul (x y: nat) :=
  match x with O => O | S x' => add (mul x' y) y end.
Fixpoint power (x y: nat) :=
  match y with O => S O | S y' => mul x (power x y') end.
Fixpoint ge (x y: nat) :=
  match x, y with _, O => True | O, S _ => False
              | S x', S y' => ge x' y' end.
Theorem Fermat's_last:
  forall (a b c n: nat),
  ge a (S O) /\ ge b (S O) /\ ge c (S O) /\ ge n (S(S(S O))) ->
  add (power a n) (power b n) <> power c n.
```

A more realistic example: The Feit-Thompson theorem

As verified in Coq by G. Gonthier et al:

```
Theorem Odd_Order:
  forall T mul one inv (G : T -> Type) (n : natural),
  group_axioms T mul one inv -> group T mul one inv G ->
  finite_of_order T G n -> odd n ->
  solvable_group T mul one inv G.
```

The definitions required by this statement fit in one page.

CompCert's correctness statement

```
Theorem transf_c_program_correct:
  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.
```

Not obvious that we proved the right thing!

CompCert's correctness statement

```
Theorem transf_c_program_correct:
  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.
```

We need to understand:

- Program behaviors and the ∀-∃-`improves` dance.
- The operational semantics for the source language (`Csem.semantics`, 2500 lines)
- The operational semantics for the target language (`Asm.semantics`, 400 lines)
- Supporting libraries: machine integers (1000 lines), floating-point numbers (2000), memory states (1500).

CompCert's correctness statement

```
Theorem transf_c_program_correct:
  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.
```

However, there is no need to understand:

- The code generation and optimization algorithms (`transf_c_program`, about 10 000 lines)
- The operational semantics for the intermediate languages.

In particular, adding a new compilation pass or improving an existing pass does not change the final correctness statement.

How to build confidence?

Review the definitions and final statement.

Analyze the definitions by proving more properties about them:

- The Flocq formalization of floating-point is used in several other verification projects.
- The CompCert Clight semantics and memory model is exercised in the VST verified separation logic.
- The CompCert C#minor semantics is exercised in the Verasco verified static analyzer.

Test executable forms of the specifications.

- CompCert provides a reference interpreter for its C semantics.
- Frameworks for executable semantics: K, PLT Redex, Ott, ...

A peek under the hood: CompCert's operational semantics and notion of semantic preservation

The Asm semantics

Very naturally, a Labeled Transition System:

$$(\text{registers}, \text{memory}) \xrightarrow{\ell} (\text{registers}', \text{memory'})$$

One transition = execute the instruction pointed by register PC.

Broadly similar to the instruction set manuals of the target architecture.

More abstract in some respects:

- The code is immutable and not stored in memory.
- No bit-level encoding of instructions.

add*x*

Add (x’7C00 0214’)

add	**r**D**,r**A**,r**B	(OE = 0 Rc = 0)
add.	**r**D**,r**A**,r**B	(OE = 0 Rc = 1)
addo	**r**D**,r**A**,r**B	(OE = 1 Rc = 0)
addo.	**r**D**,r**A**,r**B	(OE = 1 Rc = 1)

31	D	A	B	OE	266	Rc
0 5	6 10	11 15	16 20	21	22 30	31

```
rD ← (rA) + (rB)
```

The sum (**r**A) + (**r**B) is placed into **r**D.

The **add** instruction is preferred for addition because it sets few status bits.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (If Rc = 1)

  NOTE: CR0 field may not reflect the infinitely precise result if overflow occurs (see next bullet item.

- XER:

  Affected: SO, OV (If OE = 1)

  NOTE: For more information on condition codes see Section 2.1.3, “Condition Register,” and Section 2.1.5, “XER Register.”

Observable transitions

Most transitions (labeled τ) are internal computations.

Some transitions (labeled $c!v$ or $c?v$) are input/output operations that can be observed from the outside of the program:

- Calls to functions representing OS services (e.g. `getchar`, `putchar`).
- Reads and writes to "volatile" memory areas corresponding to hardware I/O devices.

Compilation must preserve the observable I/O actions but can change the internal computation steps.

From transitions to whole-program behaviors

Normal termination with trace $a_1 \ldots a_k$:

$$\texttt{initial} \ni s \xrightarrow{\tau} s_1 \xrightarrow{a_1} s_2 \xrightarrow{\tau} \cdots \xrightarrow{a_k} s_n \in \text{final}$$

Abnormal termination with trace $a_1 \ldots a_k$:

$$\texttt{initial} \ni s \xrightarrow{\tau} s_1 \xrightarrow{a_1} s_2 \xrightarrow{\tau} \cdots \xrightarrow{a_k} s_n \in \text{error}$$

Reactive divergence with infinite trace $a_1 \ldots a_k \ldots$:

$$\texttt{initial} \ni s \xrightarrow{\tau} \cdots \xrightarrow{a_i}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} \cdots \xrightarrow{a_j}\xrightarrow{\tau}\xrightarrow{\tau} \cdots$$

Silent divergence with trace $a_1 \ldots a_k$:

$$\texttt{initial} \ni s \xrightarrow{\tau} \cdots \xrightarrow{a_k} s_n \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} \cdots$$

The CompCert C semantics

Early versions of CompCert used big-step (natural) semantics, with some twists:

- normal, inductive big-step semantics with traces of I/O (for terminating behaviors);
- co-inductive big-step semantics with traces of I/O (for diverging behaviors).

Eventually we switched to Labeled Transition Systems so as to correctly handle

- Nondeterminism in expression evaluation order.
- Unstructured control such as `goto`.

Transitions with continuations

(A. Appel and S. Blazy)

$$(c, k, m) \xrightarrow{\ell} (c', k', m')$$

c is the statement or expression under focus.

k is the continuation of c: what to do "after" c?
(e.g. "loop one more time" or "return to caller")

m is the current memory state and environments.

Transitions are either computational steps or refocusing steps (updating c,k to change the focus, without doing actual work):

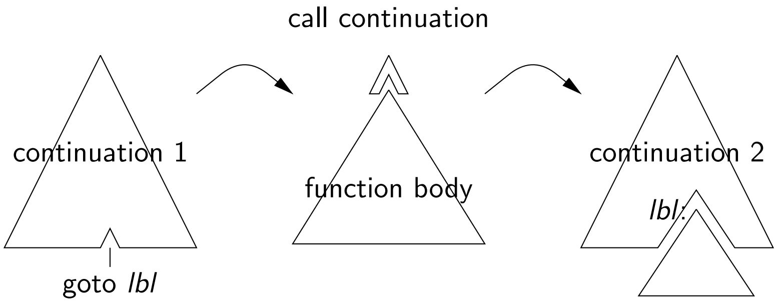
$$(s_1; s_2), k, m \xrightarrow{\tau} s_1, \mathtt{kseq}(s_2, k), m \qquad \text{(focusing on } s_1\text{)}$$

$$(loc = val), k, m \xrightarrow{\tau} \mathtt{skip}, k, m\{loc \leftarrow val\} \qquad \text{(computing)}$$

$$\mathtt{skip}, \mathtt{kseq}(s, k), m \xrightarrow{\tau} s, k, m \qquad \text{(focusing out)}$$

Handling goto by continuation surgery

A search function that finds a subcommand labeled *lbl* while manufacturing the corresponding continuation:



Implements the transition `goto` $lbl/k_1/m \xrightarrow{\tau} lbl : c/k_2/m$.

Relating the C and Asm semantics through compilation

First try: bisimulation

> *Any possible behavior of the generated Asm code is a possible behavior of the source C code, and conversely.*

Bisimulation is too strong, because:

- The C source can have several possible evaluation orders, leading to internal nondeterminism, and the compiler can pick one specific evaluation order.
- The C source can exhibit undefined behaviors (run-time errors), which the compiler need not preserve exactly.

Evaluation orders in C expressions

```
int a(void) { printf("a"); return 1; }
int b(void) { printf("b"); return 2; }
int c(void) { printf("c"); return 3; }
int main(void) { return a() + b() + c(); }
```

The subexpressions `a()` and `b()` and `c()` can be evaluated in any order that the compiler chooses.

$\Rightarrow$ any of the 6 permutations `abc`, `acb`, `bac`, `bca`, `cab`, `cba` is a valid output for this program.

“Undefined behavior” in C

Systems programs can encounter run-time errors:

- Integer division by zero.
- Accessing an array outside of its bounds.
- Dereferencing the null pointer.
- Possibly: overflow in signed integer arithmetic.

High-level languages such as Java define those behaviors, typically as aborting the computation by raising an exception.

The C standard treats those run-time errors as undefined behavior, where anything can happen:

- Execution can be aborted on a fatal error.
- Execution can continue with any value the hardware provides.
- Other parts of the program can misbehave.

The pros and cons of undefined behaviors

Pros: more efficient machine code can be generated:

- No need to check for null pointers, array bounds, zero divisor.
- The compiler can optimize under the assumption that no undefined behavior occurs:

```
int x = *p;                                int x = *p;
if (p == NULL) return ERROR;   --->        // redundant test
...                                        ...
```

Cons: undefined behaviors open security holes.

Compiling undefined behaviors

Strict interpretation of the C standards: if the source code has undefined behaviors, the compiled code can do anything.

More useful interpretation: if the source code does some I/O t then runs into undefined behavior, the compiled code should do the same I/O t and then can do anything.

Source code:	$i_1.o_1.o_2.i_2.o_3$	$i_1.o_1.\dagger$ run-time error
Compiled code:	$i_1.o_1.o_2.i_2.o_3$	$i_1.o_1.o_2\ldots$
	(same behavior)	("improved" undefined behavior)

Improving whole-program behaviors

To capture the “more useful” interpretation, we define the improvement relation $b \preceq b'$ as

- either $b = b'$,
- or b is “run-time error after performing an I/O trace t”, and b' is any behavior whose trace starts with t.

Compiler correctness as refinement with behavior improvement

We can finally state the correctness property of CompCert:

> *If compilation succeeds, for every observable behavior b of the compiled code, there exists a possible observable behavior b' of the source program such that $b' \preceq b$.*

```
Theorem transf_c_program_correct:
  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.
```

Backward simulation diagrams

Original program Transformed program

Simulation:

$$\begin{tikzcd}
\texttt{safe} \ni S_1 \arrow[r, dash, "\mathcal{R}"] \arrow[d, "t", "{+ \text{ or } |S_2'| < |S_2| \text{ and } *}"'] & S_2 \arrow[d, "t"] \\
S_1' \arrow[r, dash, "\mathcal{R}"] & S_2'
\end{tikzcd}$$

Progress:

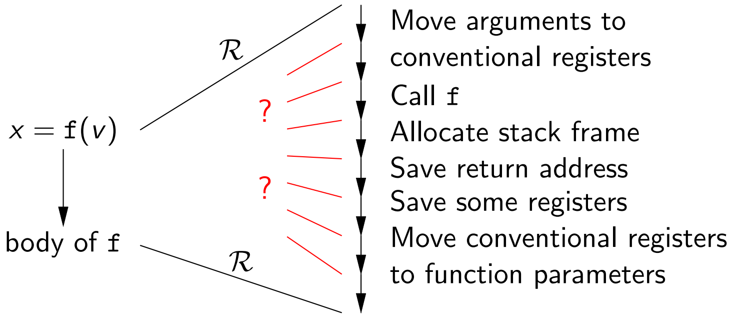
$$\begin{tikzcd}
\texttt{safe} \ni S_1 \arrow[r, dash, "\mathcal{R}"] & S_2 \notin \texttt{final} \arrow[d, "t"] \\
 & S_2'
\end{tikzcd}$$

$S \in \texttt{safe}$ means $\neg\, S \xrightarrow[\tau]{*} \texttt{error}$ (cannot crash silently).

Backward simulation woes

It is hard to build backward simulation relations in the frequent case where a single, atomic transition of the original program is implemented by several, simpler steps in the transformed program.

Example: function calls.



Alternative proof: Forward simulation

For passes the do not reduce nondeterminism, it's easier to reason "in the other direction":

Original program   Transformed program

$$\begin{tikzcd}
\texttt{error} \notin S_1 \arrow[rr, dash, "\mathcal{R}"] \arrow[d, "t"] & & S_2 \arrow[d, "t"', "{+ \text{ or } * \text{ and } |S_1'| < |S_1|}"] \\
S_1' \arrow[rr, dash, "\mathcal{R}"] & & S_2'
\end{tikzcd}$$

Theorem (Ševčík, Vafeiadis, Zappa Nardelli, Jagannathan, Sewell)

A backward simulation can be constructed from a forward simulation if the source language is receptive and the target language is determinate.

Concluding remarks

On tool verification

CompCert is still an ongoing project, but it demonstrates that the formal verification of realistic compilers is feasible (within the limitations of today's proof assistants).

See also the related Verasco project: formal verification of a C static analyzer based on abstract interpretation.

Much work remains:

- Handle other source languages: functional, reactive.
- More optimizations, esp. loop optimizations.
- Increase confidence even further.
- Shared-memory concurrency.

On trusting the specifications

A difficult problem, faced by all kinds of formal verifications.

A small simplification in the specifications is worth a large increase in proof effort.

Executable specifications (e.g. reference interpreters) can help:

- For testing the specifications.
- To discuss with standard committees.

On mechanized semantics

A need shared by many verification efforts, not just verified compilers.

A difficult task, especially for realistic programming languages (i.e. Java and the JVM; C; Javascript).

Machine assistance is a necessity to scale up to realistic programming languages.

The sensitivity is disturbingly high: adding one language feature can deeply impact the whole semantics.

The unreasonable effectiveness of Labeled Transition Systems (despite looking more like abstract machines than high-level specs).

Determinacy and receptiveness

Two labels are compatible $\ell_1 \asymp \ell_2$ if they differ only by input values.
(I.e. $\ell_1 = \ell_2 = \tau$ or $\ell_1 = \ell_2 = c!v$ or $\ell_1 = c?v_1, \ell_2 = c?v_2$.)

A language is determinate if:

- $s \xrightarrow{\ell_1} s_1$ and $s \xrightarrow{\ell_2} s_2$ imply $\ell_1 \asymp \ell_2$.
- $s \xrightarrow{\ell} s_1$ and $s \xrightarrow{\ell} s_2$ imply $s_1 = s_2$.

In other words: the only nondeterminism comes from the input values in labels. This is the case for CompCert Asm.

A language is receptive if:

- $s \xrightarrow{\ell_1} s_1$ and $\ell_1 \asymp \ell_2$ implies $\exists s_2, s \xrightarrow{\ell_2} s_2$.

In other words: a transition with an input is possible regardless of the value of the input. This is the case for all CompCert languages.

Putting it all together

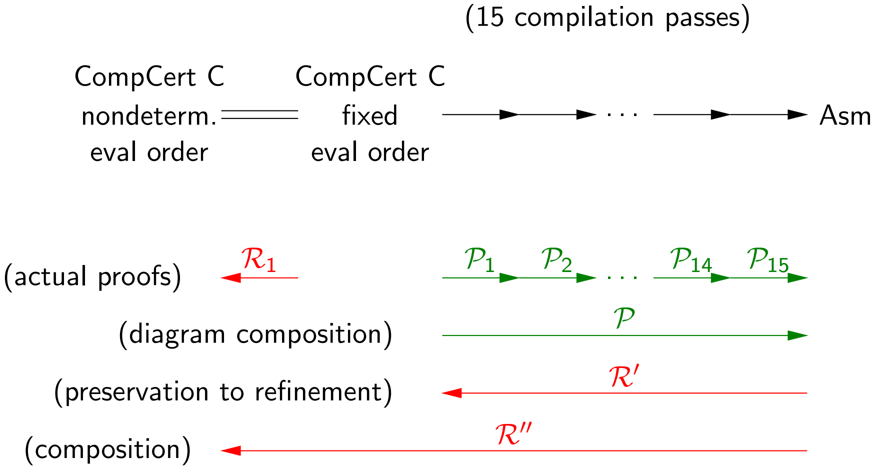
$$\begin{tikzcd}[column sep=small]
\begin{array}{c}\text{CompCert C}\\\text{nondeterm.}\\\text{eval order}\end{array} \arrow[r, equal] & \begin{array}{c}\text{CompCert C}\\\text{fixed}\\\text{eval order}\end{array} \arrow[r, "\text{(15 compilation passes)}"] & {} \arrow[r] & \cdots \arrow[r] & {} \arrow[r] & \text{Asm}
\end{tikzcd}$$

Putting it all together

(15 compilation passes)

$$\begin{tikzcd}
\text{CompCert C nondeterm. eval order} \arrow[r, equal] & \text{CompCert C fixed eval order} \arrow[r] & {} \arrow[r] & \cdots \arrow[r] & {} \arrow[r] & \text{Asm} \\
\text{(actual proofs)} & {} \arrow[l, "\mathcal{R}_1"'] \arrow[r, "\mathcal{P}_1"] & {} \arrow[r, "\mathcal{P}_2"] & \cdots \arrow[r, "\mathcal{P}_{14}"] & {} \arrow[r, "\mathcal{P}_{15}"] & {}
\end{tikzcd}$$

Putting it all together


(15 compilation passes)
CompCert C nondeterm. eval order
CompCert C fixed eval order
. . .
Asm
(actual proofs)
$\mathcal{R}_1$
$\mathcal{P}_1$
$\mathcal{P}_2$
. . .
$\mathcal{P}_{14}$
$\mathcal{P}_{15}$
(diagram composition)
$\mathcal{P}$
(preservation to refinement)
$\mathcal{R}'$
(composition)
$\mathcal{R}''$