Verified compilation
An introduction to CompCert

Sandrine Blazy

Université de Rennes

UMR IRISA

OPLSS, Eugene, 2023-07-05
Deductive verification

SOFTWARE

CORRECT

in the sense of

PROOF

conducted with

MATHEMATICAL RIGOUR

SPECIFICATION

in

LOGIC
From early intuitions …

A. M. Turing.
Checking a large routine. 1949.
... to deductive-verification and automated tools
Floyd 1967, Hoare 1969

SOFTWARE
  written in
    LANGUAGE
      defined by
        SEMANTICS
          e.g.
            INTERPRETER

CORRECT
  in the sense of
    VERIFIED SOFTWARE

PROOF
  produces
    PROOF CERTIFICATE
      defined by
        MATHEMATICAL RIGOUR

SOFTWARE TOOL
  conducted with
    PROOF
      enforces
        SOFTWARE TOOL

INPEATRANTS
  either or
    FUNCTIONS LANGUAGE

SPECIFICATION
  including
    LOGIC
      in
Another historical example

Boyer-Moore’s majority. 1980

Given N votes, determine the majority if any

\[ \text{majority} = A \]
\[ \text{cpt\_delta} = 3 \]

**MJRTY—A Fast Majority Vote Algorithm**

Robert S. Boyer and J Strother Moore

Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

**Abstract**

A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election. The number of comparisons required is at most twice the number of votes. Furthermore, the algorithm uses storage in a way that permits an efficient use of magnetic tape. A Fortran version of the algorithm is exhibited. The Fortran code has been proved correct by a mechanical verification system for Fortran. The system and the proof are discussed.
Another historical example

Boyer-Moore’s majority. 1980

Given N votes, determine the majority if any

```
A A A C C B B C C C B C C
```

majority = A
cpt_delta = 3

```
A A A C C B B C C C B C C
```

majority = A
cpt_delta = 1

---

**MJRTY—A Fast Majority Vote Algorithm**

*Robert S. Boyer and J Strother Moore*

Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

**Abstract**

A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election. The number of comparisons required is at most twice the number of votes. Furthermore, the algorithm uses storage in a way that permits an efficient use of magnetic tape. A Fortran version of the algorithm is exhibited. The Fortran code has been proved correct by a mechanical verification system for Fortran. The system and the proof are discussed.
Part 1: summary

SOFTWARE

written in

C LANGUAGE

defined by

SEMANTICS

e.g.

INTERPRETER

VERIFIED

COMPILER

in the sense of

CORRECT

PROOF

conducted with

COQ PROOF

ASSISTANT

enforces

SPECIFICATION

including

INTEGRANTS

VERIFIED

COMPILER

including

conducted with

enforces

SPECIFICATION

including
Lecture material

These slides
(including some slides borrowed from by Xavier Leroy)

Reused Coq developments


Mechanized semantics, second lecture

Traduttore, traditore:
formal verification of a compiler

Xavier Leroy
2019-12-12
Collège de France, chair of software sciences

Mechanized semantics: the Coq development

This repository contains the Coq sources for the course “Mechanized semantics” given by Xavier Leroy at Collège de France in 2019-2020.

This is the English version of the Coq sources. La version commentée en français est disponible ici.

An HTML pretty-printing of the commented sources is also available:

1. The semantics of an imperative language
   - Module Imp: the imperative language Imp and its various semantics.
   - Library Sequences: definitions and properties of reduction sequences.

2. Formal verification of a compiler
   - Module Compile: compiling Imp to a virtual machine.
   - Library Simulation: simulation diagrams between two transition systems.
Part 2: early intuitions
The miscompilation risk

Compilers still contain bugs!

We found and reported hundreds of previously unknown bugs [...]. Many of the bugs we found cause a compiler to emit incorrect code without any warning. 25 of the bugs we reported against GCC were classified as release-blocking.

[Yang, Chen, Eide, Regehr. Finding and understanding bugs in C compilers. PLDI’11]
Verified compilation

Compilers are complicated programs, but have a rather simple end-to-end specification:

The generated code must behave as prescribed by the semantics of the source program.

This specification becomes mathematically precise as soon as we have formal semantics for the source language and the machine language.

Then, a formal verification of a compiler can be considered.
An old idea …

Mathematical Aspects of Computer Science, 1967

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
Now taught as an exercise
(Mechanized semantics: when machines reason about their languages, X.Leroy)
(Software foundations, B.Pierce et al.: exercise stack_compiler_correct)

Definition state := string → nat.

Inductive aexp := ANum(n:nat) | AId(x:string) | APlus(a1 a2:aexp) | ...

Inductive sinstr := SPush(n:nat) | SLoad(x:string) | SPlus | SMinus | SMult.

Fixpoint s_execute(s:state)(stack:list nat)(prog:list sinstr):list nat :=
  match (prog, stack) with
  | (nil, _ ) => stack
  | ... end.

Fixpoint aeval(s:state)(e:aexp):nat := ...

Fixpoint s_compile(e:aexp): list sinstr := ...

Fixpoint s_execute(s:state)(stack:list nat)(prog:list sinstr):list nat :=
  match (prog, stack) with
  | (nil, _ ) => stack
  | ... end.

semantics (aeval, s_execute)  
compiler (s_compile)
Fixpoint \( \text{aeval}(s:\text{state})(e:\text{aexp}):\text{nat} := \ldots \)
Now taught as an exercise
(Mechanized semantics: when machines reason about their languages, X.Leroy)
(Software foundations, B.Pierce et al.: exercise stack_compiler_correct)

Fixpoint \( aeval(s:\text{state})(e:\text{aexp}):\text{nat} := \ldots \)

Fixpoint \( s\_compile(e:\text{aexp}):\text{list sinstr} := \ldots \)

Fixpoint \( s\_execute(s:\text{state})(\text{stack:list nat})(\text{prog:list sinstr}):\text{list nat} := \ldots \)

Theorem execute_app : \( \forall \text{st} \ p1 \ p2 \ \text{stack}, \)
\( s\_execute \ \text{st} \ \text{stack} \ (p1 ++ p2) = s\_execute \ \text{st} \ (s\_execute \ \text{st} \ \text{stack} \ p1) \ p2. \)
Proof.
(* ... *)
Qed.

Theorem s_compile_correct_aux : \( \forall \text{s} \ e \ \text{stack}, \)
\( s\_execute \ \text{s} \ \text{stack} \ (s\_compile \ e) = aeval \ e :: \text{stack}. \)
Proof.
induction e; (* ... *)
Qed.

Theorem s_compile_correct : \( \forall \text{s} \ e, \)
\( s\_execute \ \text{s} \ [] \ (s\_compile \ e) = [aeval \ s \ e]. \)
Proof.
intros. apply s compile correct aux.
Qed.

Extraction s compile.
Course outline

Formal verification in Coq of a non-optimizing compiler for a simple imperative language (from IMP language to VM language)

Extension of these ideas to CompCert, a realistic C compiler
The CompCert formally verified compiler
(X.Leroy, S.Blazy et al.)
https://compcert.org

A moderately optimizing C compiler

Targets several architectures (PowerPC, ARM, RISC-V and x86)

Programmed and verified using the Coq proof assistant

Shared infrastructure for ongoing research

Used in commercial settings (for emergency power generators and flight control navigation algorithms) and for software certification - AbsInt company

Improved performances of the generated code while providing proven traceability information

ACM Software System award 2021
ACM SIGPLAN Programming Languages Software award 2022
Part 3: basics of verified compilation
Compiling IMP instructions
Already seen in Imp.v

**Denotational** style for the semantics of IMP expressions

**Big-step** (operational) style for commands: relation $c/s \Rightarrow s'$

## Fixpoint aeval

```latex
\text{Fixpoint aeval}(s:\text{state}) (e:\text{aexp}) : \text{nat} := ...
```

```latex
\text{Definition example: } \text{com} := \langle\{ X := X + 1 \}\rangle.
```

```latex
\text{Definition same_example: } \text{com} := \text{CAss } X \text{ (APlus (AId } X \text{ ) (ANum 1))}.
```

## boolean expressions

```latex
\text{Definition example: } \text{com} := \langle\{ X := X + 1 \}\rangle.
```

```latex
\text{Definition same_example: } \text{com} := \text{CAss } X \text{ (APlus (AId } X \text{ ) (ANum 1))}.
```

## Inductive com

```latex
\text{Inductive com} :=
| \text{CSkip} |
| \text{CAss } (x:\text{ string}) (a:\text{ aexp}) |
| \text{CSeq } (c1 \ c2: \text{ com}) |
| \text{CIf } (b:\text{ bexp}) (c1 \ c2: \text{ com}) |
| \text{CWhile } (b:\text{ bexp}) (c: \text{ com}). |
```

## Inductive ceval

```latex
\text{Inductive ceval} : \text{com} \rightarrow \text{state} \rightarrow \text{state} \rightarrow \text{Prop} :=
| \text{E_Skip} : \forall \text{ st, st } = [\text{ skip }] \Rightarrow \text{ st} |
| \text{E_WhileFalse} : \forall \text{ b st c, beval st b } = \text{ false } ->
  \text{ st } = [\text{ while b do c end }] \Rightarrow \text{ st} |
| \text{E_WhileTrue} : \forall \text{ st st' st'' b c, beval st b } = \text{ true } ->
  \text{ st } = [\text{ c }] \Rightarrow \text{ st' } ->
  \text{ st' } = [\text{ while b do c end }] \Rightarrow \text{ st'' } ->
  \text{ st } = [\text{ while b do c end }] \Rightarrow \text{ st''} |
| \text{ ...} |
```
Extending the VM language: instruction set

```
Inductive instr: Type :=
| Iconst (n: Z) (* formerly SPush *)
| Ivar (x: ident) (* formerly SLoad *)
| Iadd
| Isetvar (x: ident) (* pop an integer and assign it to variable *)
| Ibranch (d: Z) (* skip forward or backward d instructions *)
| Iopp (* pop one integer, push its opposite *)
| Ibeq (d1 d0: Z) (* pop 2 integers, skip d1 instructions if =, d0 if ≠ *)
| Ible (d1 d0: Z) (* pop 2 integers, skip d1 instructions if ≤, if > *)
| Ihalt. (* stop execution *)

Definition code := list instr.
```

```
Definition ex_code1:code := Ivar "x" :: Iconst 1 :: Iadd :: Isetvar "x" :: nil.
Definition ex_code2:code :=
  Ivar "x" :: Iconst 1 :: Iadd :: Isetvar "x" :: Ibranch (-5) :: nil.
```

```
x := x + 1
```
Small-step semantics, given by a transition relation $s \rightarrow s'$

Definition stack := list Z.
Definition store := ident → Z.
Definition config := (Z * stack * store).

Inductive transition (C:code): config → config → Prop :=
  | trans_const: ∀ pc stack s n,
    instr_at C pc = Some(Iconst n) →
    transition C (pc, stack, s) (pc + 1, n :: stack, s)
  | trans_setvar: ∀ pc stack s x n,
    instr_at C pc = Some(Isetvar x) →
    transition C (pc, n :: stack, s) (pc + 1, stack, update x n s)
  | trans_branch: ∀ pc stack s d pc',
    instr_at C pc = Some(Ibranch d) →
    pc' = pc + 1 + d →
    transition C (pc, stack, s) (pc', stack, s)
  | ...

branch instructions increment by 1+d
increments pc by 1
position of the currently executing instruction
formerly called state

fixed list of instructions
Execution of VM programs

Small-step (operational) semantics

Definition **transitions** \( (C: \text{ code}) : \text{ config} \rightarrow \text{ config} \rightarrow \text{ Prop} \) :=
\[
\text{star} \ (\text{transition } C).
\]

Definition **machine_terminates** \( (C: \text{ code}) \ (s_\text{init} \ s_\text{final}: \text{ store}) \) :=
\[
\exists \ pc, \ \text{transitions} \ C \ (0, \text{nil}, s_\text{init}) \ (pc, \text{nil}, s_\text{final}) \\
\land \ \text{instr_at} \ C \ pc = \text{Some Ihalt}.
\]
Sequences of transitions and their properties

Sequences.v

\[ S \rightarrow S' \]

Variable `A`: Type. (* type of states *)
Variable `R`: `A \rightarrow A \rightarrow \text{Prop}`. (* transition relation between states *)

\[ S \rightarrow \ast S' \]

Inductive `\ast`: `A \rightarrow A \rightarrow \text{Prop}` :=
- `\ast_refl`: `\forall` `a`, `\ast` `a` `a`
- `\ast_step`: `\forall` `a` `b` `c`, `R` `a` `b` \rightarrow `\ast` `b` `c` \rightarrow `\ast` `a` `c`.

\[ S \rightarrow + S' \]

Inductive `+`: `A \rightarrow A \rightarrow \text{Prop}` :=
- `+\text{\_left}`: `\forall` `a` `b` `c`, `R` `a` `b` \rightarrow `\ast` `b` `c` \rightarrow `+` `a` `c`.

\[ \text{Definition } \text{irred} (a:A): \text{Prop} := \quad (* \text{stuck states} *) \]
\[ \forall b, \neg(R \ a \ b). \]
Compilation of commands

- Code for (CIf b c1 c2)
  - Code for b
  - Code for c1
  - Ibranch
  - Code for c2

- Code for (CWhile b c)
  - Code for b
  - Code for c
  - Ibranch
Compiler correctness

**Lemma** compile_com_correct_terminating:
\[ \forall s \, c \, s', \]
\[ \text{cexec } s \, c \, s' \rightarrow \]
\[ \forall C \, \text{pc stack}, \]
\[ \text{code_at } C \, \text{pc (compile_com } c) \rightarrow \]
\[ \text{transitions } C \]
\[ (\text{pc}, \text{stack}, s) \]
\[ (\text{pc} + \text{codelen (compile_com } c), \text{stack}, s'). \]

**Definition** compile_program \( (p: \text{com}) : \text{code} := \)
\[ \text{compile_com } p \, \text{++ } \text{Ihalt} \, \text{:: nil}. \]

**Theorem** compile_program_correct_terminating:
\[ \forall s \, c \, s', \]
\[ \text{cexec } s \, c \, s' \rightarrow \]
\[ \text{machine_terminates} \, (\text{compile_program } c) \, s \, s'. \]

**Definition** machine_terminates \( (C: \text{code}) \, (s_{\text{init}} \, s_{\text{final}}: \text{store}) := \)
\[ \exists \, \text{pc}, \text{transitions } C \, (0, \text{nil}, s_{\text{init}}) \, (\text{pc}, \text{nil}, s_{\text{final}}), \]
\[ \land \text{instr_at } C \, \text{pc = Some Ihalt}. \]

**proof by induction on the derivation of**
cexec \( s \, c \, s' \)
Part 3: summary

« The generated code must behave as prescribed by the semantics of the source program. »

Theorem \texttt{s\_compile\_correct}: \( \forall s e, \)

\[ s\_execute(s\_compile(e)) = [aeval(e)]. \]

Theorem \texttt{compile\_program\_correct\_terminating}:

\[ \forall s c s', \]

\[ \texttt{cexec}\ s\ c\ s' \rightarrow \]

\texttt{machine\_terminates}(\texttt{compile\_program}\ c)\ s\ s'.

This is not enough to conclude that the compiler is correct!
Part 4: semantic preservation and compiler verification
What should be preserved?
Observable behaviors

\[
S \rightarrow S' \qquad S \rightarrow ^+ S' \quad \quad S \xrightarrow{\infty}
\]

normal termination

\[
x := 1; \quad \text{Ihalt} \quad \text{return 0;}
\]

going wrong

\[
\text{finite sequence of transitions to a state that is stuck and not final}
\]

infinite sequence of transitions

\[
\text{while true do skip end} \quad \text{Ibranch (-1)} \quad \text{for(;;) { }}
\]

observable behaviors

\[
\text{finite sequence of transitions to a final state}
\]
Summary of yesterday’s lecture

Expressions: big-step semantics

Commands: big-step semantics

Instructions: small-step semantics

Theorem compile_program_correct_terminating:
\[ \forall s \ c \ s', \ c\text{exec} \ s \ c \ s' \rightarrow \ \text{machine
terminates} \ (\text{compile}\_\text{program} \ c) \ s \ s'. \]
Summary of yesterday’s lecture

We need to equip IMP with a small-step semantics $S \rightarrow S'$

What about diverging programs?

How do we compare the behaviors of two programs?

Compiler correctness theorem for terminating programs

What about diverging programs?

How do we compare the behaviors of two programs?

Instructions: small-step semantics

Expressions: big-step semantics

Commands: small-step semantics

Compiler

Small-step semantics

Behaviors

Termination

Divergence

Instructions: small-step semantics

Expressions: big-step semantics

Compiler semantics

Behaviors

Termination

Divergence

$S \rightarrow ^* S'$

$S \rightarrow ^+ S'$

$S \Rightarrow$
Should «going wrong» behaviors be preserved?

Compilers routinely optimize away going-wrong behaviors.

This program goes wrong.

However, the compiler eliminates \( x=1/0; \) as it is dead code.

Thus, the generated code always terminates.

Justifications

- We know that the program does not go wrong (e.g. by static analysis).
- It is the programmer’s responsibility to avoid going-wrong behaviors (C standards).
Should «going wrong» behaviors be preserved?

This program goes wrong.

However, the code generated by the compiler does not check the array bounds.

The generated code may crash but in general it prints an arbitrary integer and terminates normally.

This out-of-bound access is an example of an undefined behavior (according to the ISO C standard).

```c
#include <stdio.h>
int main()
{
    int x[2] = { 12, 34 };  
    printf("x[2] = %d\n", x[2]);
    return 0;
}
```
Notions of semantic preservation: bisimulation

The source program S and the compiled program have exactly the same behaviors.

• Every possible behavior of S is a possible behavior of C.
• Every possible behavior of C is a possible behavior of S.

Example for the IMP to VM compiler

• (compile_com c) terminates if and only if c terminates, with the same final store
• (compile_com c) diverges if and only if c diverges
• (compile_com c) never goes wrong
Forward simulation

Forward simulation from a source program S to a compiled code C: every possible behavior of S is a possible behavior of C

Example:
- theorem compile_program_correct_terminating
- If C diverges, (compile_com C) diverges

This looks insufficient: what if C has more behaviors than S? For instance, if C can terminate or go wrong?
Reducing non-determinism during compilation

A language is deterministic if every program has only one behavior.

The C language is not deterministic: the evaluation order is partially unspecified.

The expression \( f() + g() \) can evaluate either to:

- 1 if \( f() \) is evaluated first (returning 1), then \( g() \) (returning 0);
- -1 if \( g() \) is evaluated first (returning 1), then \( f() \) (returning 0).

Every C compiler chooses one evaluation order at compile-time. The compiled code therefore has fewer behaviors than the source program (1 instead of 2). Forward simulation and bisimulation fail.
Backward simulation

Backward simulation from a source program S to a compiled code C: every possible behavior of C is a possible behavior of S. However, C may have fewer behaviors than S.

If the target language is deterministic, forward simulation implies backward simulation, and therefore bisimulation.
Simulations for safe programs

A program is **safe** when it either terminates or diverges.

**Safe forward simulation**: any behavior of the source program S other than « going wrong » is a possible behavior of the compiled code C.

**Safe backward simulation**: for any behavior b of the compiled code C, the source program S can either have behavior b or go wrong.
Simulation diagrams

Behaviors are defined in terms of sequences of transitions.

Forward simulation from a source program S to a compiled code C can be proved as follows:

• show that every transition in S is simulated by some transitions in C

• while preserving an invariant ≈ between the states of S and C

Backward simulation is similar but simulates transitions of C by transitions of S.
Lock-step simulation

Every transition in the source $S$ is simulated by exactly one transition in the compiled code $C$

Further show that initial states are related: $S_{\text{init}} \approx C_{\text{init}}$

and final states are related: $S \approx C \land S \in \text{Final} \Rightarrow C \in \text{Final}$
From lock-step simulation to forward simulation

\[ S_{\text{init}} \approx C_{\text{init}} \]
\[ S_1 \approx C_1 \]
\[ S_{n-1} \approx C_{n-1} \]
\[ \text{Final } \ni S_n \approx C_n \in \text{Final} \]

Likewise if \( S_{\text{init}} \) makes an infinity of transitions
Plus simulation

Example: compilation of \( x := x + 1 \) into

\[
\text{Ivar } "x" :: \text{Iconst 1 :: Iadd :: Isetvar } "x" :: \text{nil}
\]

(already seen on this slide)

Forward simulation still holds
Incorrect star simulation

Forward simulation is not guaranteed:

- terminating executions are preserved,
- but diverging executions may not be preserved
The problem of infinite stuttering

The source program diverges but the compiled code can terminate normally or by going wrong.

This denotes an incorrect optimization of a diverging program, e.g. compiling (while true skip) into skip.
Corrected star simulation

\[ S_1 \approx C_1 \approx S_2 \]

or

\[ S \approx C \]

with \[ 0 \leq \text{measure}(S') < \text{measure}(S) \]

\[ \text{measure}(S) : \text{nat} \] from source states (could be to a well-founded set)

If the source program diverges, it must perform infinitely many non-stuttering steps, so the compiled code executes infinitely many transitions.
Coq library for star simulations: from star simulation to forward simulation

Variable C1: Type.  (* the type of configurations for the source program *)
Variable step1: C1 → C1 → Prop.  (* its transition relation *)

Variable C2: Type.  (* the type of configurations for the transformed program *)
Variable step2: C2 → C2 → Prop.  (* its transition relation *)

Variable inv: C1 → C2 → Prop.  (* the invariant ≈ *)
Variable measure: C1 → nat.  (* the measure that prevents infinite stuttering *)

Hypothesis simulation: ∀ c1 c1', step1 c1 c1' → ∀ c2, inv c1 c2 → ∃ c2', (plus step2 c2 c2' ∨ (star step2 c2 c2' ∧ measure c1' < measure c1)) ∧ inv c1' c2'.

or

\[ S \approx C \quad \text{with} \quad 0 \leq \text{measure}(S') < \text{measure}(S) \]
We need to equip IMP with a small-step semantics:

\[ S \rightarrow S' \]

The correctness theorem is about small-step semantics, which is proved by observing behaviors. These behaviors include termination and divergence.

Reasoning about these behaviors is strengthened with anti-stuttering measure. Using simulation diagrams, not yet fully proved!
Part 5: small-step semantics and compiler verification
A small-step semantics for IMP

Relation: \[ c / s \rightarrow c' / s' \]

- \[ x := a / s \rightarrow \text{skip} / x \mapsto (\text{aeval } a / s) ; s \]
- \[ (c ; \text{skip}) / s \rightarrow c / s \]
- \[ \text{eval } s b = \text{true} \]
- \[ (\text{if } b \text{ then } c_1 \text{ else } c_2) / s \rightarrow c_1 / s \]
- \[ \text{eval } s b = \text{false} \]
- \[ (\text{while } b \text{ do } c \text{ end}) / s \rightarrow \text{skip} / s \]

Big-step semantics for expressions:

- \[ c_1 / s_1 \rightarrow c_2 / s_2 \]
- \[ (c_1 ; c) / s_1 \rightarrow (c_2 ; c) / s_2 \]
- \[ \text{eval } s b = \text{false} \]
- \[ (\text{if } b \text{ then } c_1 \text{ else } c_2) / s \rightarrow c_2 / s \]
- \[ \text{eval } s b = \text{true} \]
- \[ (\text{while } b \text{ do } c \text{ end}) / s \rightarrow c ; \text{while } b \text{ do } c \text{ end} / s \]
Equivalence with big-step semantics

A classic result:

\[ c/s \Rightarrow s' \quad \text{if and only if} \quad c/s \rightarrow^* \text{skip}/s' \]

This proof is useful to build confidence in both semantics
Spontaneous generation of commands

Some rules generate fresh commands that are not subterms of the source program.

\[(\text{if } b \text{ then } c_1 \text{ else } c_2); c / s \rightarrow (c_1; c) / s\]

Raises two issues when using simulation diagrams:

• impractical to reason on the execution relation
• difficult to define the measure
Small-step semantics with continuations

Instead of rewriting whole commands: \[ c / s \rightarrow c' / s' \]

rewrite pairs of (subcommand under focus, continuation):

\[ c / k / s \rightarrow c' / k' / s' \]

Continuation

- remainder of command
- context in which it occurs (control stack)
  - \texttt{Kstop} nothing remains to be done
  - \texttt{c \& k} execution of a sequence of two commands
  - \texttt{Kwhile b c k} execution of a loop
Small-step semantics with continuations

No generation of fresh commands: c’ is always a subterm of c

\[
\frac{c}{k/s} \rightarrow \frac{c’}{k’/s’}
\]

New kinds of rules for dealing with continuations

\[
\frac{(\text{if } b \text{ then } c_1 \text{ else } c_2)}{k/s} \rightarrow \frac{c_1}{k/s} \quad \text{when } \text{eval } s \ b = \text{true}
\]

\[
\frac{(c_1;c_2)}{k/s} \rightarrow \frac{c_1}{c_2} \bullet k/s 
\quad \text{Focus (on the left of a sequence)}
\]

\[
\frac{\text{skip} / c}{\bullet k/s} \rightarrow \frac{c}{k/s} \quad \text{Resume (the remaining computations)}
\]
A small-step semantics for IMP

\[ x := a / k / s \rightarrow \text{skip} / k / x \mapsto (\text{aeval} a s); s \]

\[ (c_1 ; c_2) / k / s \rightarrow c_1 / c_2 \bullet k / s \]

\[ \text{eval } s \ b = \text{true} \]
\[ \frac{ (\text{if } b \text{ then } c_1 \text{ else } c_2) / k / s \rightarrow c_1 / k / s }{ \text{eval } s \ b = \text{false} } \]

\[ \text{eval } s \ b = \text{false} \]
\[ \frac{ (\text{while } b \text{ do } c \text{ end}) / k / s \rightarrow \text{skip} / k / s }{ \text{eval } s \ b = \text{true} } \]

\[ \text{skip} / c \bullet k / s \rightarrow c / k / s \]

\[ \text{skip} / \text{Kwhile } b \ c \ k / s \rightarrow \text{while } b \text{ do } c \text{ end} / k / s \]
Program execution

Termination

Definition \texttt{kterminates} (s: store) (c: com) (s': store) :=
star step (c, K\texttt{stop}, s) (\texttt{SKIP}, K\texttt{stop}, s').

Divergence

Definition \texttt{kdiverges} (s: store) (c: com) :=
infseq step (c, K\texttt{stop}, s).

Equivalence between small-step semantics

Theorem \texttt{equiv\_smallstep\_terminates}:
\[ \forall s \ c \ s', \ \texttt{terminates} s \ c \ s' \leftrightarrow \texttt{kterminates} s \ c \ s'. \]

Theorem \texttt{equiv\_smallstep\_diverges}:
\[ \forall s \ c, \ \texttt{diverges} s \ c \leftrightarrow \texttt{kdiverges} s \ c. \]
Full proof of compiler correctness
Simulation diagram

Difficulties

• find the invariant ≈ between source and target states
• find the measure from source states to a natural number

\[
\begin{align*}
C \vdash c_1/k_1/s_1 \approx (pc_1, [], s'_1) & \quad (pc_1, [], s'_1) \\
C \vdash c_2/k_2/s_2 \approx (pc_2, [], s'_2) & \quad (pc_2, [], s'_2)
\end{align*}
\]

\[
C \vdash c_1/k_1/s_1 \approx (pc_1, [], s'_1) \quad (pc_1, [], s'_1)
\]

\[
C \vdash c_2/k_2/s_2 \approx (pc_1, [], s'_2) \quad (pc_1, [], s'_2)
\]

\[
C \vdash c_1/k_1/s_1 \approx (pc_1, [], s'_1) \quad (pc_1, [], s'_1)
\]

\[
C \vdash c_2/k_2/s_2 \approx (pc_1, [], s'_2) \quad (pc_1, [], s'_2)
\]

with \(0 \leq \text{measure}(c_2, k_2) < \text{measure}(c_1, k_1)\)
When do the source program stutter? When no VM instruction is executed.

\[
\begin{align*}
(c1 \; c2) / k / s & \rightarrow c1 / c2 \bullet k / s \\
\text{skip} / c \bullet k / s & \rightarrow c / k / s \\
(\text{if true then c1 else c2}) / k / s & \rightarrow c1 / k / s \\
(\text{while true do c end}) / k / s & \rightarrow c; \text{while b do c end} / K\text{while b c k} / s
\end{align*}
\]

\textbf{measure}(c,k): sum of the sizes of c and all the commands appearing in k
Full proof of compiler correctness
The simulation invariant

Remember this slide:

\[ C \vdash c/k/s \approx (pc, stack, s') \] is defined as:

- \( s = s' \)
- \( stack = [] \)
- \( code_{at} C \ pc (compile_{com} c) \)
- \( C \) contains compiled code matching \( k \) at \( pc + code_{len}(compile_{com} c) \)

\[ \forall s c s', ceval s c s' \rightarrow \forall C \ pc \ stack, \]
\[ \text{code}_{at} C \ pc (\text{compile}_{com} c) \rightarrow \]
\[ \text{transitions} C (pc, stack, s) \]
\[ (pc + code_{len}(\text{compile}_{com} c), stack, s'). \]
Compiler correctness: wrapping up

**Theorem** \( \text{compile\_program\_correct\_terminating}: \forall s \ c \ s', \)
\[ \text{ceval} \ s \ c \ s' \rightarrow \text{machine\_terminates} \ (\text{compile\_program} \ c) \ s \ s'. \]

**Theorem** \( \text{compile\_program\_correct\_terminating\_2}: \forall s \ c \ s', \)
\[ \text{star} \ \text{step} \ (c, \ K\text{stop}, \ s) \ (\text{SKIP}, \ K\text{stop}, \ s') \rightarrow \text{machine\_terminates} \ (\text{compile\_program} \ c) \ s \ s'. \]

**Theorem** \( \text{compile\_program\_correct\_diverging}: \forall c \ s, \)
\[ \text{infseq} \ \text{step} \ (c, \ K\text{stop}, \ s) \rightarrow \text{machine\_diverges} \ (\text{compile\_program} \ c) \ s. \]
Part 5: summary

Correctness theorem for terminating and diverging programs is about small-step semantics. Is proved by observing emissions of traces about behaviors. Belong to termination and divergence is facilitated by continuations. Reasoning is strengthened with simulation diagrams using anti-stuttering measure.

Alternate proof for terminating programs.
Part 6
How to turn CompCert from a prototype in a lab into a real-world compiler?
CompCert compiler: 11 languages, 18 passes

Optimisations:
- constant prop., CSE, tail calls
- (LCM), (software pipelining)
- register allocation (IRC)
- linearisation of the CFG
- instruction selection
- spilling, reloading
- calling conventions
- layout of stack frames
- ASM code generation
- instruction scheduling
- stack allocation of «&» variables
- no side-effect determinization
- type elimination
Multiplicity of source behaviors
Reducing non-determinism during compilation

The C language is not deterministic: the evaluation order is partially unspecified.

The expression \( f() + g() \) can evaluate either to:

• 1 if \( f() \) is evaluated first (returning 1), then \( g() \) (returning 0);
• -1 if \( g() \) is evaluated first (returning 1), then \( f() \) (returning 0).

Every C compiler chooses one evaluation order at compile-time. The compiled code therefore has fewer behaviors than the source program (1 instead of 2). **Forward simulation fails.**
Back to simulations

S: source program   C: compiled program

Backward simulation: every possible behavior of C is a possible behavior of S

**Safe backward simulation:** for any behavior $b$ of C, S can have either behavior $b$ or go wrong

If the target language is deterministic, forward simulation implies backward simulation (and therefore bisimulation)
Handling multiple compilation passes

Theorem transf_c_program_correct:
∀ p tp,
transf_c_program p = OK tp →
backward_simulation (Csem.semantics p)
(Asm.semantics tp).

Compiler.v
Verification patterns

Verified transformation
- transformation

Verified translation validation
- transformation
- validator

Verified validator
- Less to prove (if validator simpler than transformation)
- Validator reusable for several variants of an optimization
- Can be efficient (cheap enough to be invoked on every compiler run)

Example: register allocation with advanced spilling and splitting
CompCert compiler: 11 languages, 18 passes

- **CompCertC**
  - Optimisations: constant prop., CSE, tail calls, (LCM), (software pipelining)
  - no side-effect determinization

- **Clight**
  - type elimination
  - CFG construction expr. decomp.
  - instruction selection

- **C#minor**
  - stack allocation of «&» variables

- **CminorSel**
  - basic block structuring

- **Cminor**
  - (instruction scheduling)
  - spilling, reloading calling conventions

- **LTL**
  - register allocation (IRC)
  - linearisation of the CFG

- **LTLin**
  - ASM code generation
  - layout of stack frames

- **ASM**
  - 66

- **Mach**
  - no side-effect determinization

- **RTL**
CompCert compiler: 11 languages, 18 passes

Small-step semantics

$S \xrightarrow{t} S'$

$S \xrightarrow{\ast} S'$

$S \xrightarrow{+} S'$

$S \xrightarrow{n} S'$

termination

$S \xrightarrow{\infty}$

divergence

Behaviors

going wrong

execL P b

Smallstep.v library

CompCertC

Clight

C#minor

RTL

CminorSel

Cminor

LTL

LTLin

Linear

ASM

Mach
Observable behaviors

Behaviors.v and Events.v

Inductive program_behavior :=
  | Terminates (t: trace) (n: int)
  | Diverges (t: trace)
  | Reacts (tinf: traceinf)
  | Goes_wrong (t: trace).

trace = list of I/O events
traceinf = infinite list of I/O events

I/O event
- call to an external function (e.g. printf)
- memory accesses to global volatile variables (hardware devices)
General form of small-step semantics

**Smallstep.v**

\[
G \vdash S \xrightarrow{t} S'
\]

**Semantic states** include a memory state, mapping addresses to values.

\[
\text{initial\_state}(S)
\]

\[
\text{final\_state}(S,n)
\]

Does not change during transitions

Observed events

\(G\) maps:

- each name of a function or global variable to a memory address
- each function pointer to a function definition

Return value
The CompCert memory model

Memory.v

Shared by all the languages of the compiler

An abstract view of memory refined into a concrete memory layout

In the semantics:

Memory operations (load, store, alloc, free) over values (machine integers, pointers, floating-point numbers)

Memory safety preserved by CompCert (good variable properties)

Generic memory injections and memory extensions
Semantic states

Exemple: Clight

Inductive state :=
  | Callstate (fd: fundef)(args: list val)(k: cont)(m: mem)

Exception: assembly languages, where a state is a pair of a memory and a mapping from processor registers to values.
CompCert C source language  
(see chapter 4 of the user’s manual)

**Expressions** are annotated with their type

Overloading and implicit conversions between types

Expressions have side-effects

- Assignments are expressions

Non-deterministic evaluation of expressions (e.g., see this slide)

Numerous semantic rules in small-step style

**Commands**

All C constructs: loops, switch, goto, break, continue, return

Numerous semantic rules in small-step style

```c
Eval(int(5), Tint(I32, Signed)): expr
```
Clight language

Expressions are annotated with their type
No overloading and explicit conversions between types and arithmetic operators
Expressions are pure
Temporary variables do not reside in memory
19 semantic rules in big-step style

Commands
Assignments are commands
Single syntax for loops, continue command
  ▶ C loops are derived forms
25 semantic rules in small-step style
+ numerous rules for unary and binary operators, memory loads and stores
The CompCert C reference interpreter

Outcome:

- normal termination or aborting on an undefined behavior
- observable effects (I/O events: `printf`, `malloc`, `free`)

Faithful to the formal semantics of CompCert C; the interpreter displays all the behaviors according to the semantics

```
step: state → trace → state → Prop
```

```
do_step: world → state → list (trace * state)
```

external world: uniquely determines the results of external calls
Using the reference interpreter: exhaustive exploration
Using the reference interpreter: randomized exploration

First choice

Randomized

S0
S1
S4
S0
S1
S4
S0
S3
S7
S7
S9
Sb
Sb
Using the reference interpreter
A first example

```c
int main(void)
{   int x[2] = { 12, 34 };
    printf("x[2] = %d\n", x[2]);
    return 0; }
```

Stuck state: in function main, expression
`<printf>(<ptr __stringlit_1>, <loc x+8>)`

Stuck subexpression: `<loc x+8>`

**ERROR: Undefined behavior**
Using the reference interpreter
A second example: randomized exploration

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

int main () { printf("%d\n", a() + (b() + c())); return 0;  }
```

State 45.9: returning 3
State 45.10: returning 2
State 45.11: returning 1
State 55.1: returning 0
Time 55: program terminated (exit code = 0)
RTL language

RTL.v

Each function is represented by its CFG
Instructions only
Unlimited supply of pseudo-registers

```c
int f(int n) {
    int x = 1;
    for (int i = 1; i < n; i++)
        if (x < 9) x = x + 2;
        else if (x > 50) x = x + 1;
        else x = 2 * x;
    return x;
}
```
Part 6: summary

Proving a compiler pass mainly amounts to proving a simulation diagram

Many reusable libraries:
- simulations, memory model, C semantics, Clight and RTL languages
- machine integers, dataflow solver

Some compilation options
- using the CompCert C interpreter: `-interp` (`-trace`, `-all`, `-random`)
- tracing options: `-dc`, `-dclight`, `-drtl`, ...
- show the time spent in compiler passes: `-timing`
Part 7: Compiling critical embedded software with CompCert
Fly-by-wire software

Execute pilot's commands
Flight assistance: keep aircraft within safe flight envelope
Mostly control-command code (Scade) + a minimalistic OS (C)

100k - 1M LOC code, but mostly generated from block diagrams (Simulink, Scade)

Fly-by-wire software
Rigorous validation: review (qualitative), analysis (quantitative), testing (huge amounts)
Conducted at multiple levels, from design to final product
Meticulous development process; extensive documentation

The qualification process (DO-178)
From block diagrams to assembly

Delay symbol
Delay macro
Variable stored in RAM
Input
Output
Observation point
Compiler
Code generator

/* Sheet Number: 1EEEEE7 */
#include "other_includes.h"
#include "delay.mac"
#include "csci.mac"

M_START_NUM_INPUT_ZONE
M_INPUT(SENSORDATA, NUM)
M_END_NUM_INPUT_ZONE

M_START_NUM_OUTPUT_ZONE
M_OUTPUT(OUTPUTZONE, NUM)
M_END_NUM_OUTPUT_ZONE

M_START_NUM_OBS_ZONE
M_OBS_NUM(POINT_PIQ)
M_END_NUM_OBS_ZONE

M_START_CST_INPUT_ZONE
M_CST(c_DELAY_F3_2_1EEEEE7, ENT, 1)
M_END_CST_INPUT_ZONE

M_START_STATIC_ARRAY_ZONE
M_STATIC_ARRAY(t_DELAY_F3_2_1EEEEE7, 1, NUM)
M_END_STATIC_ARRAY_ZONE

M_START_CEINT(1E, EE, X7)
M_CONNECTION(P1EEEZ72, 1EEEEE7, NUM)
M_CONNECTION(loc_c_DELAY_F3_2_1EEEEE7, ENT)
M_CONNECTION(P1EEEZ72, 1EEEEE7, NUM)
M_CONNECTION(P1EEEZ72, 1EEEEE7, NUM)
M_VTS(SENSORDATA, 1EEEEE7)
M_CST(c_DELAY_F3_2_1EEEEE7, loc_c_DELAY_F3_2_1EEEEE7)

COS (0_1EEEEE7, P1EEEZ72, 1EEEEE7, P1EEEZ72, 1EEEEE7)
P_0 (0_1EEEEE7, P1EEEZ72, 1EEEEE7, POINT_PIQ)
DELAY(c_DELAY_F3_2_1EEEEE7, loc_c_DELAY_F3_2_1EEEEE7, t_DELAY_F3_2_1EEEEE7)

M_END_CEINT

%!_1EEEEE7:
; annotation: Symbol DELAY number 2, 1EEEEE7 ,
; inputs: f3, r31 and one static
100 addh r12, 0, (c_DELAY_F3_2_1EEEEE7, r2)
104 luh r4, (c_DELAY_F3_2_1EEEEE7, r2)
108 mr r7, r4
110 add r8, r12, (t_DELAY_F3_2_1EEEEE7)
114 rlvimm r10, r7, r8, 0, 28 ; 0x8f0000
118 add r10, r6, r10
11c lfd r2, (r10)
; annotation: Variable to search: loc_c_DELAY_F3_2_1EEEEE7
; annotation: DELAY, t entered with r4 = from 0 to
120 mr r6, r4
124 add r12, 0, (c_DELAY_F3_2_1EEEEE7)
128 add r6, r10, (t_DELAY_F3_2_1EEEEE7)
12c rlvimm r5, r6, r8, 3, 0, 28 ; 0x800000
130 add r9, r6, r5
134 urc r3, (r9)
138 add r4, r4, 1
13c cmr cr0, r4, r31
140 bt 0, .L101
144 add r4, 0, 0
.L101:
148 add r12, 0, (c_DELAY_F3_2_1EEEEE7)
14c stv r4, (c_DELAY_F3_2_1EEEEE7)
; annotation: End of DELAY number 2, 1EEEEE7 , output: j2
Program annotations

A mechanism to attach annotations to program points
• Mark specific program points
• Provide information about the location of C variables
• Ensure that some variables are preserved (e.g. x must be kept in a register)

Annotations are preserved during compilation.
• Each annotation generates an observable event
• The correctness theorem ensures preservation of the sequencing of 1) symbols, and 2) of accesses to hardware devices (volatile variables)

_annot("Begin of a loop");
...
x = 1;
_annot("Here x is at %d", x);
...
_annot("End of a loop");

; annotation: Begin of a loop
... addi r3, 0, 1
; annotation: Here x is at r3
... ; annotation: End of a loop
Conformance to the qualification process

A formally verified compiler gives traceability guarantees.

Simplified example

• The semantics preservation theorem ensures preservation of:
  • the sequencing of symbols,
  • the sequencing of accesses to hardware devices (volatile variables).

Remember the main theorem: If the source program can not go wrong, then the behavior of the generated assembly code is exactly one of the behaviors of the source program.
How good is the compiled code?

Trade-off between
- traceability guarantees
- and efficiency of the generated code

Low-level verifications
- reviews of the assembly
- computation of a WCET estimation
Compiling critical embedded software

Improved performances of the generated code, while providing proven traceability guarantees thanks to annotations

FCGU A380: 3600 files, 3.96 MB of assembly code
- Estimated WCET for each file
- Average improvement per file: 13.5%
- Compiled with CompCert 1.10, 2012
Overall assessment

The improvement mainly comes from the register allocation pass.

- From: no register allocation
- To: sharing of local variables among available registers

Traceability guarantees

- From: tracking of all program variables
- To: tracking of meaningful variables (used in block diagrams)
Part 8: CompCert, a shared infrastructure for ongoing research
The Verasco abstract interpreter

A holistic effect with compiler verification

**CompCert compiler**

Theorem csharpminor_compiler_correct_alt:
\[ \forall \ p \ tp \ b, \]
\[ \text{transf\_c\_program} \ p = \text{OK} \ tp \rightarrow \]
\[ \text{execC} \ p \ b \rightarrow \]
\[ \text{execASM} \ tp \ b. \]

**Verasco abstract interpreter**

Theorem analyzer_is_correct:
\[ \forall \ p \ b, \]
\[ \text{static\_analyzer} \ p = \text{Success} \rightarrow \]
\[ \text{execC} \ p \ b. \]

Theorem csharp Compiler correct stronger:
\[ \forall \ p \ tp \ b, \]
\[ \text{transf\_c\_program} \ p = \text{OK} \ tp \rightarrow \]
\[ \text{execASM} \ tp \ b. \]
Verasco architecture

- **CompCert C** → **Clight** → **C#minor** → ... → **CompCert compiler**

**States**
- Abstract interpreter
- Control flow
- Alarms

**Numerical Abstraction**
- Symbolic conditional expressions (improve precision of assume commands)
- Intervals
- Congruences
- Symbolic equalities
- Polyhedra
- Linearization
- Octagons

**Integer and floating-point arithmetic**
- Conjunctions of linear inequalities $\sum ai \cdot xi \leq c$

**Communication channels**
- Crucial to analyze the safety of memory accesses (memory alignment)

- Integer and floating-point arithmetic
  - Crucial to analyze the safety of memory accesses (memory alignment)
  - Conjunctions of linear inequalities $\pm x \pm y \leq c$

- Symbolic equalities
  - Symbolic conditional expressions (improve precision of assume commands)

- Algebraic integers (abstract interpreter)

- Algebraic integers (abstract interpreter)
Cryptographic constant-time (CCT) programming discipline

unsigned nok-function (unsigned x, unsigned y, bool secret)
{ if (secret) return y; else return x; }

unsigned ok-function (unsigned x, unsigned y, bool secret)
{ return x ^ ((y ^ x) & -((unsigned)secret)); }

How to turn CompCert into a formally-verified secure compiler?

**Theorem** compiler-correct:
\[ \forall \ S \ C \ b, \]
\[ \text{compiler} \ S = \text{OK} \ C \rightarrow \]
\[ \text{execCompCertC} \ S \ b \rightarrow \]
\[ \text{execASM} \ C \ b. \]

**Theorem** compiler-preserves-CCT:
\[ \forall \ S \ C, \]
\[ \text{compiler} \ S = \text{OK} \ C \rightarrow \]
\[ \text{isCCT} \ S \rightarrow \]
\[ \text{isCCT} \ C. \]
Which proof technique for the isCCT policy?

**Observational non-interference:** observing program leakage (boolean guards and memory accesses) during execution does not reveal any information about secrets

**Theorem compiler-preserves-CCT:**
\[
\forall S, C, \quad \text{compiler } S = \text{OK } C \rightarrow \text{isCCT } S \rightarrow \text{isCCT } C.
\]

**Indistinguishability property** \( \varphi(S_i, S'_i) \): share public values, but may differ on secret values

**Difficulty:** tricky proofs!
Proving CCT preservation: back to simulation diagrams

Proof-engineering: leverage the existing proof scripts as much as possible

must predict the number of steps at target level

\[ t' = t \]

or \( t' = \varepsilon \) and \( t \) is leak only

\[ S_1 \approx C_1 \approx S_2 \approx C_2 \]

\[ S'_1 \approx C'_1 \approx t' \approx S'_2 \approx C'_2 \]

\[ n_1 \]

\[ n_2 \]
Verifying just-in-time (JIT) compilation [Barrière’s PhD 12/2022]
[Barrière, Blazy, Flückiger, Pichardie, Vitek, POPL’21] and [Barrière, Blazy, Pichardie, POPL’23]

A JIT compiler **interleaves** the execution of a program with its optimizations.

Dynamic speculation: specializes functions, requires deoptimization.

Non-deterministic semantics: either deoptimize to the source program or continue to the next instruction in the optimized program.
Proving semantics preservation: the simulation approach

Both the program and the execution state are evolving
Nested simulations for JIT verification

Both the program and the execution state are evolving.

Invariant $\approx_{\text{JIT}}$: at any point during JIT execution

- the current state $C_i$ corresponds to a source state $S_i$
- the curent JIT program $P_i$ is equivalent to the source program $P_0$

Nested simulation: this equivalence is expressed with another simulation.
Work in progress

new font-end

Catala

CompCertC

new optimizations

new back-end

new target

Clight

C#minor

Cminor

CminorSel

RTL

LTL

LTLin

Linear

Mach

ASM

SSA

GSA

FPGA
Gated SSA (static single assignment) intermediate representation

```c
int f(int n) {
    int x = 1;
    for (int i = 1; i < n; i++)
        if (x < 9) x = x + 2;
        else if (x > 50) x = x + 1;
        else x = 2 * x;
    return x;
}
```

C program

Program in SSA form

Program in GSA form

Path predicates

\[ \mathcal{P}_2: x_1 < 9 \]
\[ \mathcal{P}_3: x_1 \geq 9 \land x_1 > 50 \]
\[ \mathcal{P}_4: x_1 \geq 9 \land x_1 \leq 50 \]
Conclusion and perspectives

CompCert is a shared infrastructure for ongoing research

- **compilation**: ProbCompCert (Boston College, USA), L2C (Tsinghua, China), Velus (DIENS, Fr), CompCertO (Yale, USA), VeriCert (Imperial College, GB), CompCert-KVX (Verimag, Fr)

- **program logics**: VST (Princeton, USA), Gillian (Imperial College, GB), VeriFast (KUL, Be)

- **static analysis**: Verasco (Inria, Fr)

Opens the way to the trust of development tools

From early intuitions to fundamental formalisms … verification tools that automate these ideas … actual use in the critical software industry
Questions?
Bibliography

- Yang, Chen, Eide, Regehr. Finding and understanding bugs in C compilers. PLDI’11.
- Kumar, Myreen, Norrish, Owens. CakeML: a verified implementation of ML. POPL’14.
- Barrière, Blazy, Pichardie. Formally verified native code generation in an effectful JIT - or: Turning the CompCert backend into a formally verified JIT compiler. POPL’23.
- Barthe, Demange, Pichardie. Formal Verification of an SSA-based middle-end for CompCert. TOPLAS’14.
- Herklotz, Demange, Blazy. Mechanised semantics for gated static single assignment. CPP’23.