Verifying a compiler:
Why?  How?  How far?

Borrowed from
Xavier Leroy and Sandrine Blazy

Xavier Leroy
INRIA Paris-Rocquencourt

CGO 2011
Compiler verification:

Why?
Can you trust your compiler?

The miscompilation issue: Bugs in the compiler can lead to incorrect machine code being generated from a correct source program.
NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated.

http://www.nullstone.com/htmls/category/divide.htm

We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables.

E. Eide & J. Regehr, EMSOFT 2008

To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input.

X. Yang, Y. Chen, E. Eide & J. Regehr, PLDI 2011
Exhibit A: GCC bug #323

Title: optimized code gives strange floating point results.

#include <stdio.h>

void test(double x, double y)
{
    double y2 = x + 1.0; // computed in 80 bits, not rounded to 64 bits
    if (y != y2) printf("error_n");
}

void main()
{
    double x = .012;
    double y = x + 1.0; // computed in 80 bits, rounded to 64 bits
    test(x, y);
}

Why it is a bug: C99 allows intermediate results to be computed with excess precision, but requires them to be rounded at assignments.
Exhibit A: GCC bug #323

Reported in 2000.

Dozens of duplicates.

More than 150 comments.

Still not acknowledged as a bug.


Responsible for PHP’s `strtod()` function not terminating on some inputs...

...causing denial of service on many Web sites.
Are miscompilation bugs a problem?

For ordinary software:

- Compiler-introduced bugs are negligible compared with the bugs in the program itself.
- Programmers rarely run into them.
- When they do, debugging is very hard.
Are miscompilation bugs a problem?

For critical software validated by testing only:

- Good testing should find all bugs, even those compiler-introduced.
- Optimizations can complicate test plans.
Are miscompilation bugs a problem?

For critical software validated by review, analysis & testing:
(e.g. DO-178 in avionics)

- Manual reviews of (representative fragments of) generated assembly.
- Turning all optimizations off to get traceability.
- Reduced usefulness of formal verification.
The guarantees obtained (so painfully!) by source-level formal verification may not carry over to the executable code . . .
A solution? Verified compilers

With a regular compiler:

\[
\text{Source program} \quad \rightarrow \quad \text{Formal verification} \\
\downarrow \\
? \\
\downarrow \\
\text{compiler} \\
\downarrow \\
\text{Executable}
\]
A solution? Verified compilers

With a formally verified compiler:

The properties formally established on the source program carry over to the executable.
Formal verification of compilers

A radical solution to the miscompilation problem:

Apply program proof to the compiler itself to prove that it preserves the semantics of the source code.

After all, compilers are complicated programs with a simple specification:

*If compilation succeeds, the generated code should behave as prescribed by the semantics of the source program.*
An old idea...

John McCarthy
James Painter

CORRECTNESS OF A COMPILER
FOR ARITHMETIC EXPRESSIONS

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.
Compiler verification:

How far are we today?

(X. Leroy, *Formal verification of a realistic compiler*, CACM 07/2009)
The CompCert project
(X.Leroy, S.Blazy, et al)

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

- Source language: a very large subset of C.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code
  \[\Rightarrow\] careful code generation; some optimizations.

Note: compiler written from scratch, along with its proof; not trying to prove an existing compiler.
The subset of C supported

Supported natively:

- **Types**: integers, floats, arrays, pointers, `struct`, `union`.
- **Expressions**: all of C, including pointer arithmetic.
- **Control**: `if/then/else`, loops, `goto`, regular `switch`.
- **Functions**, including recursive functions and function pointers.
- **Dynamic allocation** (`malloc` and `free`).
- **Volatile accesses**.
The subset of C supported

Not supported at all:

- The long long and long double types.
- Unstructured switch (Duff’s device), longjmp/setjmp.
- Variable-arity functions.

Supported through (unproved!) expansion after parsing:

- Block-scoped variables.
- typedef.
- Bit-fields.
- Assignment between struct or union.
- Passing struct or union by value.
The formally verified part of the compiler

CompCert C → Clight → C#minor

- side-effects out of expressions
- type elimination loop simplifications
- stack allocation of “&” variables

RTL → CminorSel → Cminor

- CFG construction expr. decomp.
- instruction selection
- (Instruction scheduling)

LTL → LTLin → Linear

- linearization of the CFG
- spilling, reloading calling conventions
- layout of stack frames

LTLin → Asm → Mach

- register allocation (IRC)
- instruction selection
- (Instruction scheduling)

Optimizations: constant prop., CSE, tail calls, (LCM), (Software pipelining)
Formally verified in Coq

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

Theorem transf_c_program_is_refinement:
  forall p tp,
  transf_c_program p = OK tp ->
  (forall beh, exec_C_program p beh -> not_wrong beh) ->
  (forall beh, exec_asm_program tp beh -> exec_C_program p beh).

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

- **C source**
  - Parsing, construction of an AST
  - Type-checking, de-sugaring

- **AST C**
  - Type reconstruction
  - Graph coloring
  - Code linearization heuristics

- **Assembly**
  - Assembling
  - Linking
  - Printing of asm syntax

- **Executable**

- **AST Asm**

**Verified compiler**

- **Proved in Coq** (extracted to Caml)
- **Not proved** (hand-written in Caml)

**Part of the TCB**
- Proved in Coq

**Not part of the TCB**
- Not proved
Performance of generated code
(On a PowerPC G5 processor)
Complete source & proofs available for evaluation and research purposes:

http://compcert.inria.fr/

(+ research papers)

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

Tested on small benchmarks (up to 3000 LOC), real-world avionics codes, and by random testing.
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
Proof assistants

- Implementations of well-defined mathematical logics.
- Provide a specification language to write definitions and state theorems.
- Provide ways to build proofs in interaction with the user. (Not fully automated proving.)
- Check the proofs for soundness and completeness.

Formal semantics for realistic programming languages are large (but shallow) formal systems.

Computers are better than humans at checking large but shallow proofs.
Compiler verification:

How?
Contents

Using the IMP toy language as an example, we will review and show how to mechanize (see the associated Coq development):

1. Operational semantics.
2. Compilation to virtual machine code and its correctness proof.
3. The last part will be devoted to the CompCert compiler.
Part I

Operational semantics
 Operational semantics

1. Warm-up: expressions and their denotational semantics

2. The IMP language and its reduction semantics
Warm-up: symbolic expressions

A language of expressions comprising

- variables $x, y, \ldots$
- integer constants $0, 1, -5, \ldots, n$
- $e_1 + e_2$ and $e_1 - e_2$
  where $e_1, e_2$ are themselves expressions.

Objective: mechanize the syntax and semantics of expressions.
Syntax of expressions

Modeled as an \textit{inductive type} (see the lecture on inductive data types).

\textbf{Definition} ident := nat.

\textbf{Inductive} expr : Type :=
\begin{itemize}
\item \texttt{Evar} : ident \to expr \quad (* \texttt{Evar} (v:ident) *)
\item \texttt{Econst} : \mathbb{Z} \to expr \quad (* \texttt{Econst} (i:Z) *)
\item \texttt{Eadd} : expr \to expr \to expr \quad (* \texttt{Eadd} (e1 e2: expr) *)
\item \texttt{Esub} : expr \to expr \to expr \quad (* \texttt{Esub} (e1 e2: expr) *).
\end{itemize}

\texttt{Evar}, \texttt{Econst}, etc. are functions that construct terms of type \texttt{expr}.

All terms of type \texttt{expr} are finitely generated by these 4 functions → enables case analysis and induction.
Denotational semantics of expressions

Define $[e]_s$ as the denotation of expression $e$ (the integer it evaluates to) in state $s$ (a mapping from variable names to integers).

In ordinary mathematics, the denotational semantics is presented as a set of equations:

\[
\begin{align*}
[x]_s &= s(x) \\
[n]_s &= n \\
[e_1 + e_2]_s &= [e_1]_s + [e_2]_s \\
[e_1 - e_2]_s &= [e_1]_s - [e_2]_s
\end{align*}
\]
Mechanizing the denotational semantics

In Coq, the denotational semantics is presented as a recursive function (≈ a definitional interpreter).

Definition state := ident -> Z.

Fixpoint eval_expr (s: state) (e: expr) {struct e} : Z :=
match e with
| Evar x => s x
| Econst n => n
| Eadd e1 e2 => eval_expr s e1 + eval_expr s e2
| Esub e1 e2 => eval_expr s e1 - eval_expr s e2
end.
Using the denotational semantics (1/3)

As an interpreter, to evaluate expressions.

Definition initial_state: state := fun (x: ident) => 0.

Definition update (s: state) (x: ident) (n: Z) : state :=
  fun y => if eq_ident x y then n else s y.

Eval compute in (let x : ident := O in let s : state := update initial_state x 12 in eval_expr s (Eadd (Evar x) (Econst 1))).

Coq prints = 13 : Z.
Using the denotational semantics (1/3, cont’d)

Can also generate Caml code automatically (Coq’s extraction mechanism).

Extraction eval_expr.

(** val eval_expr : state -> expr -> z **)  
let rec eval_expr s = function
  | Evar x -> s x
  | Econst n -> n
  | Eadd (e1, e2) -> zplus (eval_expr s e1) (eval_expr s e2)
  | Esub (e1, e2) -> zminus (eval_expr s e1) (eval_expr s e2)
Using the denotational semantics (1/3, cont’d)

Can also generate Caml code automatically (Coq’s extraction mechanism).

Recursive Extraction eval_expr.

... type expr = Evar of ident | Econst of z
    | Eadd of expr * expr | Esub of expr * expr
...

let zplus x y = ...
...

(** val eval_expr : state -> expr -> z **)  
let rec eval_expr s = function  
    | Evar x -> s x  
    | Econst n -> n  
    | Eadd (e1, e2) -> zplus (eval_expr s e1) (eval_expr s e2)  
    | Esub (e1, e2) -> zminus (eval_expr s e1) (eval_expr s e2)
Using the denotational semantics (2/3)

To reason symbolically over expressions.

Lemma expr_add_pos:
  forall s x,
  s x >= 0 -> eval_expr s (Eadd (Evar x) (Econst 1)) > 0.
Proof.
  simpl.
  (* goal becomes: forall s x, s x >= 0 -> s x + 1 > 0 *)
  intros. omega.
Qed.
Using the denotational semantics (3/3)

To prove “meta” properties of the semantics. For example: the denotation of an expression is insensitive to values of variables not mentioned in the expression.

Lemma eval_expr_domain:
  forall s1 s2 e,
  (forall x, occurs_in x e -> s1 x = s2 x) ->
  eval_expr s1 e = eval_expr s2 e.

where the predicate occurs_in is defined by

Fixpoint occurs_in (x: ident) (e: expr) {struct e} : Prop :=
  match e with
  | Evar y => x = y
  | Econst n => False
  | Eadd e1 e2 => occurs_in x e1 \/ occurs_in x e2
  | Esub e1 e2 => occurs_in x e1 \/ occurs_in x e2
  end.
Variant 1: interpreting arithmetic differently

Example: signed, modulo $2^{32}$ arithmetic (as in Java).

```plaintext
Fixpoint eval_expr1 (s: state) (e: expr) {struct e} : Z :=
    match e with
    | Evar x => s x
    | Econst n => n
    | Eadd e1 e2 => normalize(eval_expr1 s e1 + eval_expr1 s e2)
    | Esub e1 e2 => normalize(eval_expr1 s e1 - eval_expr1 s e2)
end.
```

where normalize $n$ is $n$ reduced modulo $2^{32}$ to the interval $[-2^{31}, 2^{31})$.

```plaintext
Definition normalize (x : Z) : Z :=
    let y := x mod 4294967296 in
    if Z_lt_dec y 2147483648 then y else y - 4294967296.
```
Variant 2: accounting for undefined expressions

In some languages, the value of an expression can be undefined:

- if it mentions an undefined variable;
- in case of arithmetic operation overflows (ANSI C);
- in case of division by zero;
- etc.

Recommended approach: use option types, with None meaning “undefined” and Some $n$ meaning “defined and having value $n$”.

Inductive option (A: Type): A -> option A :=

| None: option A
| Some: A -> option A.
Variant 2: accounting for undefined expressions

Definition ostate := ident -> option Z.

Fixpoint eval_expr2 (s: ostate) (e: expr) {struct e} : option Z :=
  match e with
  | Evar x => s x
  | Econst n => Some n
  | Eadd e1 e2 =>
    match eval_expr2 s e1, eval_expr2 s e2 with
    | Some n1, Some n2 => Some (n1 + n2)
    | _, _ => None
  end
  | Esub e1 e2 =>
    match eval_expr2 s e1, eval_expr2 s e2 with
    | Some n1, Some n2 => Some (n1 - n2)
    | _, _ => None
  end
end.
Summary

The “denotational semantics as a Coq function” is natural and convenient…

… but limited by a fundamental aspect of Coq: all Coq functions must be total (= terminating).

✗ Cannot use this approach to give semantics to languages featuring general loops or general recursion.

✓ Use relational presentations “predicate state term result” instead of functional presentations “result = function state term”.
Operational semantics

1. Warm-up: expressions and their denotational semantics

2. The IMP language and its reduction semantics
The IMP language

A prototypical imperative language with structured control.

Expressions:
\[ e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \]

Boolean expressions (conditions):
\[ b ::= e_1 = e_2 \mid e_1 < e_2 \]

Commands (statements):
\[ c ::= \text{skip} \quad \text{(do nothing)} \\
\quad | \quad x ::= e \quad \text{(assignment)} \\
\quad | \quad c_1; c_2 \quad \text{(sequence)} \\
\quad | \quad \text{if } b \text{ then } c_1 \text{ else } c_2 \quad \text{(conditional)} \\
\quad | \quad \text{while } b \text{ do } c \text{ done} \quad \text{(loop)} \]
Abstract syntax

Inductive expr : Type :=
  | Evar: ident -> expr
  | Econst: Z -> expr
  | Eadd: expr -> expr -> expr
  | Esub: expr -> expr -> expr.

Inductive bool_expr : Type :=
  | Bequal: expr -> expr -> bool_expr
  | Bless: expr -> expr -> bool_expr.

Inductive cmd : Type :=
  | Cskip: cmd
  | Cassign: ident -> expr -> cmd
  | Cseq: cmd -> cmd -> cmd
  | Cifthenelse: bool_expr -> cmd -> cmd -> cmd
  | Cwhile: bool_expr -> cmd -> cmd.
Reduction semantics

Also called “structured operational semantics” (Plotkin) or “small-step semantics”.

Like the $\lambda$-calculus: view computations as sequences of reductions

\[ M \xrightarrow{\beta} M_1 \xrightarrow{\beta} M_2 \xrightarrow{\beta} \ldots \]

Each reduction $M \rightarrow M'$ represents an elementary computation. $M'$ represents the residual computations that remain to be done later.
Reduction semantics for IMP

Reductions are defined on \((\text{command, state})\) pairs (to keep track of changes in the state during assignments).

Reduction rule for assignments:

\[(x := e, s) \rightarrow (\text{skip}, \text{update } s \times n) \quad \text{if } \llbracket e \rrbracket s = n\]
Reduction semantics for IMP

Reduction rules for sequences:

\[
\begin{align*}
((\text{skip}; c), s) & \rightarrow (c, s) \\
((c_1; c_2), s) & \rightarrow ((c'_1; c_2), s') \quad \text{if } (c_1, s) \rightarrow (c'_1, s')
\end{align*}
\]

Example

\[
\begin{align*}
((x := x + 1; x := x - 2), s) & \rightarrow ((\text{skip}; x := x - 2), s') \\
& \rightarrow (x := x - 2, s') \\
& \rightarrow (\text{skip}, s'')
\end{align*}
\]

where \(s' = \text{update } s \times (s(x) + 1)\) and \(s'' = \text{update } s' \times (s'(x) - 2)\).
Reduction semantics for IMP

Reduction rules for conditionals and loops:

\[
\text{(if } b \text{ then } c_1 \text{ else } c_2, s) \rightarrow (c_1, s) \quad \text{if } \llbracket b \rrbracket s = \text{true}
\]

\[
\text{(if } b \text{ then } c_1 \text{ else } c_2, s) \rightarrow (c_2, s) \quad \text{if } \llbracket b \rrbracket s = \text{false}
\]

\[
\text{(while } b \text{ do } c \text{ done, } s) \rightarrow (\text{skip, } s) \quad \text{if } \llbracket b \rrbracket s = \text{false}
\]

\[
\text{(while } b \text{ do } c \text{ done, } s) \rightarrow ((c; \text{while } b \text{ do } c \text{ done}), s) \quad \text{if } \llbracket s \rrbracket b = \text{true}
\]

with

\[
\llbracket e_1 = e_2 \rrbracket s = \begin{cases} 
\text{true} & \text{if } \llbracket e_1 \rrbracket s = \llbracket e_2 \rrbracket s; \\
\text{false} & \text{if } \llbracket e_1 \rrbracket s \neq \llbracket e_2 \rrbracket s
\end{cases}
\]

and likewise for \( e_1 < e_2 \).
Reduction semantics as inference rules

\[
(x := e, s) \rightarrow (\text{skip}, s[x \leftarrow \llbracket e \rrbracket s])
\]

\[
((\text{skip}; c), s) \rightarrow (c, s)
\]

\[
(c_1, s) \rightarrow (c'_1, s')
\]

\[
((c_1; c_2), s) \rightarrow ((c'_1; c_2), s')
\]

\[
[\llbracket b \rrbracket s = \text{true}
\]

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

\[
[\llbracket b \rrbracket s = \text{false}
\]

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

\[
[\llbracket b \rrbracket s = \text{true}
\]

\[
((\text{while } b \text{ do } c \text{ done}), s) \rightarrow ((c; \text{while } b \text{ do } c \text{ done}), s)
\]

\[
[\llbracket b \rrbracket s = \text{false}
\]

\[
((\text{while } b \text{ do } c \text{ done}), s) \rightarrow (\text{skip}, s)
\]
Expressing inference rules in Coq

Step 1: write each rule as a proper logical formula

\[(x := e, s) \rightarrow (\text{skip}, \ s[x \leftarrow [e] \ s])\]

\[(c_1, s) \rightarrow (c'_1, s)\]

\[((c_1; c_2), \ s) \rightarrow ((c'_1; c_2), \ s')\]

forall x e s,
red (Cassign x e, s) (Cskip, update s x (eval_expr s e))

forall c_1 c_2 s c_1' s',
red (c_1, s) (c_1', s') \rightarrow
red (Cseq c_1 c_2, s) (Cseq c_1' c_2, s')

Step 2: give a name to each rule and wrap them in an inductive predicate definition.
Inductive red: cmd * state -> cmd * state -> Prop :=
  | red_assign: forall x e s,
    red (Cassign x e, s) (Cskip, update s x (eval_expr s e))
  | red_seq_left: forall c1 c2 s c1’ s’,
    red (c1, s) (c1’, s’) ->
    red (Cseq c1 c2, s) (Cseq c1’ c2, s’)
  | red_seq_skip: forall c s,
    red (Cseq Cskip c, s) (c, s)
  | red_if_true: forall s b c1 c2,
    eval_bool_expr s b = true ->
    red (Cifthenelse b c1 c2, s) (c1, s)
  | red_if_false: forall s b c1 c2,
    eval_bool_expr s b = false ->
    red (Cifthenelse b c1 c2, s) (c2, s)
  | red_while_true: forall s b c,
    eval_bool_expr s b = true ->
    red (Cwhile b c, s) (Cseq c (Cwhile b c), s)
  | red_while_false: forall b c s,
    eval_bool_expr s b = false ->
    red (Cwhile b c, s) (Cskip, s).
Using inductive definitions

Each case of the definition is a theorem that lets you conclude \( \text{red} \ (c, s) \ (c', s') \) appropriately.

Moreover, the proposition \( \text{red} \ (c, s) \ (c', s') \) holds only if it was derived by applying these theorems a finite number of times (smallest fixpoint).

✓ Reasoning principles: by case analysis on the last rule used; by induction on a derivation.

Example

Lemma red_deterministic: 
\[
\forall cs \ cs1, \ \text{red} \ cs \ cs1 \rightarrow \forall cs2, \ \text{red} \ cs \ cs2 \rightarrow cs1 = cs2.
\]

Proved by induction on a derivation of \( \text{red} \ cs \ cs1 \) and a case analysis on the last rule used to prove \( \text{red} \ cs \ cs2 \).
Sequences of reductions

The behavior of a command $c$ in an initial state $s$ is obtained by forming sequences of reductions starting at $(c, s)$:

- **Termination** with final state $s'$ ($c, s \downarrow s'$): finite sequence of reductions to skip.
  
  $$(c, s) \rightarrow \cdots \rightarrow (\text{skip}, s')$$

- **Divergence** ($c, s \uparrow$): infinite sequence of reductions.
  
  $$\forall (c', s'), (c, s) \rightarrow \cdots \rightarrow (c', s') \Rightarrow \exists c'', s'', (c', s') \rightarrow (c'', s'')$$

- **Going wrong** ($c, s \downarrow \text{wrong}$): finite sequence of reductions to an irreducible state that is not skip.
  
  $$(c, s) \rightarrow \cdots \rightarrow (c', s') \not\rightarrow \text{ with } c' \neq \text{skip}$$
Sequences of reductions

The Coq presentation uses a generic library of closure operators over relations $R : A \rightarrow A \rightarrow \text{Prop}$:

- $\text{star } R : A \rightarrow A \rightarrow \text{Prop}$ (reflexive transitive closure)
- $\text{infseq } R : A \rightarrow \text{Prop}$ (infinite sequences)
- $\text{irred } R : A \rightarrow \text{Prop}$ (no reduction is possible)

Definition terminates (c: cmd) (s s’: state) : Prop :=
  star red (c, s) (Cskip, s’).
Definition diverges (c: cmd) (s: state) : Prop :=
  infseq red (c, s).
Definition goes_wrong (c: cmd) (s: state) : Prop :=
  exists c’, exists s’,
  star red (c, s) (c’, s’) \land c’ <> Cskip \land \text{irred red (c’, s’)}.