The CompCert verified C compiler

Compiler built and proved by Xavier Leroy et al.

Talk given by David Pichardie - Harvard University / INRIA Rennes

Slides largely inspired by Xavier Leroy own material
Critical embedded software
Critical embedded software

High degree of assurance is required
• is the program critical_prog.ppc safe?
Critical embedded software

High degree of assurance is required

- is the program critical_prog.ppc safe?

2 options:

- qualify the PPC program as if hand-written (intensive testing, painful manual review...)

- qualify the program at the source level (static analysis, model checking, or program proof)
Critical embedded software

High degree of assurance is required
- is the program critical_prog.ppc safe?

2 options:
- qualify the PPC program as if hand-written (intensive testing, painful manual review...)
- qualify the program at the source level (static analysis, model checking, or program proof)

2nd option is preferred in practice
- can you trust your compiler?
- this talk: apply formal verification techniques to the compiler itself!
Miscompilation happens
NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated.

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

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. So far, we have reported more than 290 previously unknown bugs to compiler developers. Moreover, every compiler that we tested has been found to crash and also to silently generate wrong code when presented with valid inputs.

J. Regehr et al, 2010
How do we verify a compiler?
How do we verify a compiler?

A simple idea:
How do we verify a compiler?

A simple idea:

Program and prove your compiler in the same language!
How do we verify a compiler?

A simple idea:

Program and prove your compiler in the same language!

Which language?
How do we verify a compiler?

A simple idea:

Program and prove your compiler in the same language!

Which language?

Coq
Coq: an animal with two faces
Coq: an animal with two faces

First face:
Coq: an animal with two faces

First face:
- a proof assistant that allows to interactively build proof in constructive logic
Coq: an animal with two faces

First face:
- a proof assistant that allows to interactively build proof in constructive logic

Second face:
Coq: an animal with two faces

First face:
  • a proof assistant that allows to interactively build proof in constructive logic

Second face:
  • a functional programming language with a very rich type system
Coq: an animal with two faces

First face:
- a proof assistant that allows to interactively build proof in constructive logic

Second face:
- a functional programming language with a very rich type system
  example:

```coq
sort: \forall l: list int, \{ l’: list int | 
  Sorted l’ \land PermutationOf l l’ \}
```
Coq: an animal with two faces

First face:
- a proof assistant that allows to interactively build proof in constructive logic

Second face:
- a functional programming language with a very rich type system
  example:
  \[
  \text{sort: } \forall l: \text{list int}, \{ l': \text{list int} | \begin{array}{l}
    \text{Sorted } l' \land \text{PermutationOf } l \ l'
  \end{array} \}
  \]
- with an extraction mechanism to Ocaml
  \[
  \text{sort: int list } \to \text{ int list}
  \]
Verifying a compiler in Coq
A simple recipe
Verifying a compiler in Coq
A simple recipe

Program the compiler inside Coq

Definition compiler (p:source) : option target := ...

Compiler

Logical Framework (here Coq)
Verifying a compiler in Coq
A simple recipe

Program the compiler inside Coq

**Definition** compiler
(p : source) : option target := ...

State its correctness wrt. a formal specification of the language semantics

**Theorem** compiler_is_correct :
\( \forall p, \text{compiler } p = \text{Some } p' \rightarrow \text{behaviors}(p') \subseteq \text{behaviors}(p) \)

If compilation succeeds, the generated code should behave as prescribed by the semantics of the source program.
Verifying a compiler in Coq
A simple recipe

Program the compiler inside Coq

\textbf{Definition} \texttt{compiler}:
\begin{verbatim}
(p:source) : option target := ...
\end{verbatim}

State its correctness wrt. a formal specification of the language semantics

\textbf{Theorem} \texttt{compiler_is_correct}:
\begin{verbatim}
\forall p, \text{compiler} p = \text{Some} p' \rightarrow \text{behaviors} (p') \subseteq \text{behaviors} (p)
\end{verbatim}

We interactively and mechanically prove this theorem

\textbf{Proof}. ... (* few days later *) ... \textbf{Qed}.
Verifying a compiler in Coq
A simple recipe

Program the compiler inside Coq

**Definition** compiler
(p:source) : option target := ...

State its correctness wrt. a formal specification of the language semantics

**Theorem** compiler_is_correct :
∀ p, compiler p = Some p’ →
behaviors(p’) ⊆ behaviors(p)

We interactively and mechanically prove this theorem

**Proof.** ... (* few days later *) ... Qed.

We extract an OCaml implementation of the compiler

**Extraction** compiler.

Compiler
Source & Target Language Semantics
Soundness Proof
Logical Framework (here Coq)

parser.ml  compiler.ml  linker.ml
Trusted Computing Base (TCB)

1. Formal specification of the programming language semantics
   • already (informally) shared between any end-user programmer, compiler, static analyzer
   • must itself be rigorously proofread/animated/tested

2. Logical Framework
   • only the proof checker needs to be trusted
   • we don’t trust sophisticated decision procedures
Trusted Computing Base (TCB)

1. Formal specification of the programming language semantics
   • already (informally) shared between any end-user programmer, compiler, static analyzer
   • must itself be rigorously proofread/animated/tested

2. Logical Framework
   • only the proof checker needs to be trusted
   • we don’t trust sophisticated decision procedures
Trusted Computing Base (TCB)

1. Formal specification of the programming language semantics
   • already (informally) shared between any end-user programmer, compiler, static analyzer
   • must itself be rigorously proofread/animated/tested

2. Logical Framework
   • only the proof checker needs to be trusted
   • we don’t trust sophisticated decision procedures

Still a large code base but at least a foundational code base: logic & semantics
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
  ⇒ careful code generation; some optimizations.

Note: compiler written from scratch, along with its proof; not trying to prove an existing compiler (otherwise see Zdancewic et al’s Verified LLVM project).
The formally verified part of the compiler

- Compcert C → side-effects out of expressions → Clight
- Clight → type elimination loop simplifications → C#minor
- RTL → CFG construction expr. decomp. → CminorSel
- CminorSel → instruction selection → Cminor
- RTL → register allocation → LTL
- LTL → linearization of the CFG → LTLin
- LTLin → spilling, reloading calling conventions → Linear
- Linear → layout of stack frames → Mach
- Mach → asm code generation → ASM
- ASM → asm code generation

Optimizations: constant prop., CSE, tail calls
Formally verified in Coq

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

**Theorem** transf_c_program_is_refinement:
\[
\forall p \ tp, \\
\text{transf_c_program } p = \text{OK } tp \rightarrow \\
(\forall \beh, \text{exec_C_program } p \beh \rightarrow \text{not_wrong } \beh) \rightarrow \\
(\forall \beh, \text{exec_asm_program } tp \beh \rightarrow \text{exec_C_program } p \beh).
\]

Behaviors \( \beh = \) termination / divergence / going wrong 
+ trace of I/O operations (syscalls, volatile accesses).
Compiler verification patterns
(for each pass)
Compiler verification patterns
(for each pass)

Verified transformation

transformation
Compiler verification patterns
(for each pass)

Verified transformation

Verified translation validation

= formally verified

= not verified
Compiler verification patterns
(for each pass)

Verified transformation

External solver with verified validation

Verified translation validation

= formally verified

= not verified
External solver with verified validation

Example: register allocation
Verified translation validation
Example: SSA generation (in CompCert SSA extension)
Verified translation validation
Example: SSA generation (in CompCert SSA extension)

The untrusted generator can rely on advanced graph algorithms as Lengauer and Tarjan’s dominator tree construction and frontier dominance computation.
Verified translation validation
Example: SSA generation (in CompCert SSA extension)

The untrusted generator can rely on advanced graph algorithms as Lengauer and Tarjan’s dominator tree construction and frontier dominance computation.

We prove the validator is complete with respect to this family of algorithms.
The whole CompCert compiler

- C source → AST C (parsing, construction of an AST, type-checking, de-sugaring)
- Type reconstruction → Assembly
- Graph coloring → Assembly
- Code linearization heuristics → Assembly
- executable → Assembly (assembling, linking)
- Assembly → AST Asm (printing of asm syntax)
- AST Asm → Executable

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

Part of the TCB
Not part of the TCB
Performance of generated code
(On a PowerPC G5 processor)

Execution time

AES
Almabench
Binarytrees
Fannkuch
FFT
Knucleotide
Nbody
Qsort
Raytracer
Spectral
VMach

gcc -O0
Compcert
gcc -O1
gcc -O3
Conclusions

The formal verification of realistic compilers is feasible.
(Within the limitations of contemporary proof assistants)

Much work remains:

• Shrinking the TCB
  (e.g. verified parsing, validated assembling & linking).

• More optimizations
  (see CompCert SSA).

• Front-ends for other languages

• Concurrency
  (see Sevcik et al’s CompCert TSO and Appel and al’s Verified Software Toolchain).

• Connections with source-level verification
  (ongoing french project on a verified C static analyzer)