Formal verification of a realistic compiler

Xavier Leroy
CACM 2009

CS 7194: Great Works in Programming Languages

Presenter: Irene Yoon | Mentor: Ryan Doenges
Building robust compilers is Hard.
Bugs

We will now judge the bugs you’ve caught.
Bugs

- Random testing finds bugs in 11 C compilers [Yang et al 2011]
Bugs

• Random testing finds bugs in 11 C compilers

• **Hundreds** of previously unknown bugs

[Yang et al 2011]
Bugs

- Random testing finds bugs in 11 C compilers [Yang et al 2011]
- Hundreds of previously unknown bugs
Bugs

- Random testing finds bugs in 11 C compilers
- **Hundreds** of previously unknown bugs
- LLVM has a large test suite

[Yang et al 2011]
Building compilers is hard
✅ Building compilers is hard

✅ Testing sucks
Building compilers is hard

Testing sucks

Formalisms are good
✅ Building compilers is hard

✅ Testing sucks

✅ Formalisms are good

Formal verification of a compiler
First Published Proof of Compiler Correctness

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS*

JOHN McCARTHY and JAMES PAINTER
1967

- arithmetic expressions → stack machine code
- prototype for proving usable compilers
First **Mechanized** Proof of Compiler Correctness

1972

- ALGOL-like language → elementary assembly language
- Stanford LCF
Compiler Verification

- 100+ papers on compiler verification since 1967
Compiler Verification

- 100+ papers on compiler verification since 1967
Compiler Verification

- 100+ papers on compiler verification since 1967
1. The Verifying Compiler: A Grand Challenge for Computing Research

TONY HOARE

Microsoft Research Ltd., Cambridge, UK
CompCert
CompCert

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

- 42k Coq, 3 person years
Verified, Validated, Certifying
1. Verified transformation [Compiler Correctness]
Verified, Validated, Certifying

2. Translation validation [Translation Verification]
Verified, Validated, Certifying

3. **Certifying compiler** [*Proof-carrying Code*]
1. **Verified transformation**  
   \[\textbf{Compiler Correctness}\]

![Diagram showing source, compiler, and target]

*total and correct

2. **Translation validation**  
   \[\textbf{Translation Verification}\]

![Diagram showing source, compiler, and validator]

3. **Certifying compiler**  
   \[\textbf{Proof-carrying Code}\]

![Diagram showing source, certifying prover, proof, and proof checker]
1. Verified transformation [Compiler Correctness]

2. Translation validation [Translation Verification]

3. Certifying compiler [Proof-carrying Code]

[LeRoy '06]
=> External solver with verified validation

1. **Verified transformation** [*Compiler Correctness*]

   ![Diagram 1](image1)

   *total and correct*

2. **Translation validation** [*Translation Verification*]

   ![Diagram 2](image2)

3. **Certifying compiler** [*Proof-carrying Code*]

   ![Diagram 3](image3)
=> External solver with verified validation

1. Verified transformation \textit{[Compiler Correctness]}  
   \begin{itemize}
   \item source \rightarrow \text{COMPILER} \rightarrow \text{target}
   \end{itemize}
   \begin{itemize}
   \item \textit{*total and correct}
   \end{itemize}

2. Translation validation \textit{[Translation Verification]}
   \begin{itemize}
   \item source \rightarrow \text{COMPILER} \rightarrow \text{VALIDATOR} \rightarrow \text{target}
   \end{itemize}

3. Certifying compiler \textit{[Proof-carrying Code]}
   \begin{itemize}
   \item source \rightarrow \text{CERTIFYING PROVER} \rightarrow \text{PROOF CHECKER}
   \end{itemize}

\[\text{[Tristan and Leroy `10]}\]
Figure 1:Compilation passes and intermediate languages.
Figure 1: Compilation passes and intermediate languages.
CompCert

formal specification
→ semantic analysis tools [Appel ’11]

Figure 1: Compilation passes and intermediate languages.
Figure 1: Compilation passes and intermediate languages.
CompCert

Figure 1: Compilation passes and intermediate languages.
Semantic Preservation

- \( \text{Spec}(B) \): functional specification of observable behavior

\( \text{Spec}(B) \) cannot go wrong.

All behaviors \( B \) satisfy \( \text{Spec} \).
Semantic Preservation

- $Spec(B)$: functional specification of observable behavior

- $B$: observable behavior (trace properties of I/O)
  - “going wrong” (run-time error), termination, divergence
Semantic Preservation

- $Spec(B)$: functional specification of observable behavior
- $B$: observable behavior (trace properties of I/O)
  - “going wrong” (run-time error), termination, divergence
- $C \models Spec$ if
  - A. $C$ cannot go wrong
  - B. All behaviors $B$ satisfy $Spec$
Correctness Property

\[ S \models Spec \implies C \models Spec \]

Compiled code \( C \) preserves the fact that the source code \( S \) satisfies the specification.
Proving Semantic Preservation

\[
\begin{array}{ccc}
S_1 & \sim & S'_1 \\
\downarrow t & & \downarrow t \\
S_2 & \sim & S'_2
\end{array}
\]
Proving Semantic Preservation

\[
\begin{array}{cccc}
S_1 & \sim & S'_1 \\
\downarrow t & & \downarrow t^* \\
S_2 & \sim & S'_2
\end{array}
\]
Safety Precondition

- Compilation result will match the semantics of the input if the program is "safe" (no runtime errors)
Safety Precondition

- Compilation result will match the semantics of the input if
  if program is “safe” (no runtime errors)
- Need to prove that input program is safe
Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON
Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

You can’t trust code that you did not totally create yourself.

KEN THOMPSON
Figure 1: Compilation passes and intermediate languages.
CompCert

Figure 1: Compilation passes and intermediate languages.
Correctness Weakness

- Only runs after the preprocessing step
Correctness Weakness

- Only runs after the preprocessing step
  - *Astrée* [Cousot et al ’05], *Verasco* [Jourdan et al ’15]
Correctness Weakness

- Only runs after the preprocessing step
  - Astrée [Cousot et al '05], Verasco [Jourdan et al '15]
- Reliant on less verifiable assumptions
Correctness Weakness

- Only runs after the preprocessing step
  - *Astrèe* [Cousot et al ’05], *Verasco* [Jourdan et al ’15])
- Reliant on less verifiable assumptions
  - Coq’s correctness (*CertiCoq* [Anand et al ’17])
Correctness Weakness

- Only runs after the preprocessing step
  - *Astrèe* [Cousot et al ’05], *Verasco* [Jourdan et al ’15]
- Reliant on less verifiable assumptions
  - Coq’s correctness (*CertiCoq* [Anand et al ’17])
- Formal specification of C & PowerPC assembly
Performance
Performance

competitive with gcc -01
Bugs, revisited.

[Yang et al 2011]
Bugs, revisited.

- CompCert: errors only found in **unverified parts** (parser and model of machine)

[Yang et al 2011]
Bugs, revisited.

[Yang et al 2011]

- CompCert: errors only found in **unverified parts** (parser and model of machine)

- Other compilers: errors everywhere
Bugs, revisited.

- CompCert: errors only found in unverified parts (parser and model of machine)

- Other compilers: errors everywhere

“The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent”
Critical Use Cases

- AirBus
Critical Use Cases

- AirBus
- MTU Friedrichshafen (nuclear energy)
Critical Use Cases

- AirBus

- MTU Friedrichshafen (nuclear energy)

- High-Assurance Cyber Military Systems (HACMS) [Fisher et al, ’17]
Critical Use Cases

- AirBus
- MTU Friedrichshafen (nuclear energy)
- High-Assurance Cyber Military Systems (HACMS) [Fisher et al, ’17]
- PhD Theses
Critical Use Cases

“a realistic compiler”

- AirBus
- MTU Friedrichshafen (nuclear energy)
- High-Assurance Cyber Military Systems (HACMS) [Fisher et al, ’17]
- PhD Theses
Concluding Remarks

- Still some correctness and safety weaknesses
- Useful for safety critical code (that doesn’t have to run fast)
Concluding Remarks

- Still some correctness and safety weaknesses
- Useful for safety critical code (that doesn’t have to run fast)

Future work -

[Logos and icons indicating various software tools and projects related to verified software development]
Concluding Remarks

- Still some correctness and safety weaknesses
- Useful for safety critical code (that doesn’t have to run fast)

Future work -

- Verified Software Toolchain
- CertiCoq
- Vellvm verified LLVM
Concluding Remarks

- Still some correctness and safety weaknesses
- Useful for safety critical code (that doesn’t have to run fast)

Future work -

Type Preserving Compilation

- Verified Software Toolchain
- CertiCoq
- Vellvm
Concluding Remarks

- Still some correctness and safety weaknesses
- Useful for safety critical code (that doesn’t have to run fast)

Type Preserving Compilation

Future work -

Principle 1: Erase the types! Compiler correctness is a stronger property than type preservation, anyway.
Thanks!