Progress in Formal Verification of Compilers: A Survey

CS410P/510 Programming Language Compilation
Winter 2024
What?

• Compiler:

```
Source Language ➔ Compiler ➔ Target Language
```

• Correctness:

```
if t = compile(s)
then behavior(t) matches behavior(s)
```

• for suitable definition of behavior and matching

• (Mechanized) Verification:

give a mechanically checked proof of correctness on all programs
Why?

• Real compilers have **bugs**, but verified ones have fewer:

  The striking thing about our CompCert results is that the middle-end 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

  - [Yang+11]
Why? (2)

• Verifying algorithms helps us **understand** them much better
  • Especially useful to tame the “optimization zoo”

• Formal verification requires formal **specification** of language semantics (behavior) and semantic preservation (matching)
  • Not easy to get right!
  • Useful for many other tasks...
Compiler Verification in Context

Possible goals involving formal semantics of L:

• Verifying “meta-properties” of language L
  • e.g. well-typed L programs don’t crash at runtime

• Verifying properties of particular L programs
  • e.g. this L function computes square roots correctly

• Verifying properties of transformations on L
  • e.g. this compiler from L to assembly code is correct

• In practice, there is overlap, e.g. language RTS.
Two Schools of Mechanized Proof

• **Interactive Provers** ("proof assistants")
  - Finding proof is not fully automated
  - Checking is fully automated (and trustworthy)
  - Logics can be very expressive
  - Examples: Coq, Isabelle, ACL2, PVS, HOL etc.

• **Automatic Provers**
  - Finding proof (or refutation) is fully automated
  - Logics strictly limited in power (e.g. no quantifiers)
  - Can handle very large problems
  - Examples: Z3, CVC, Simplify etc.
Defining Compiler Correctness

• Key idea: **observable** properties of source behavior should also be properties of target
  • e.g. trace of IO system calls
  • note: internal behavior is generally not preserved!
• Hence, target code should only do things source code might do (**simulation/refinement**)
• In practice, many tricky technical issues:
  • non-termination, error behaviors, granularity of comparison, etc.
Two approaches to verification:

- **verified transformations**
  - are directly proven to preserve observable behavior
  - typically by showing they preserve (internal) invariants
  - compiler must be a “white box” (probably one we wrote)
Two approaches to verification:

- (verified) **translation validation**
  - on each run, check that compiler output is correct; otherwise fail-stop
  - we must hope it seldom fail-stops!
  - compiler can be a “black box” (maybe) or a “gray box”
  - (must prove checker is correct)

- most clearly a win if checking output is easier than generating it
Toy Example in Coq

• To make these ideas concrete, consider an extremely simple “compiler” from arithmetic expressions

\[ e := x \mid n \mid e + e \mid e - e \mid e \times e \]

to stack-machine code

\[ i := \text{Push } n \mid \text{Load } x \mid \text{Plus} \mid \text{Minus} \mid \text{Mult} \]

• See compver.v
The CompCert C Compiler

- [Leroy+06] See [http://compcert.inria.fr](http://compcert.inria.fr)
- Goal: A verified production-quality C compiler usable for critical embedded software
- Source language: (most of) C
- Target language: PPC, ARM, or X86 assembler
- Coq is used for proof and to implement (most of) the compiler itself (using extraction)
- Generates respectable target code, but does little optimization
Compiler Pass Structure

CompCert C → side-effects out of expressions → Clight → type elimination loop simplifications → C#minor

Optimizations: constant prop., CSE, inlining, tail calls, dead code

RTL → CFG construction expr. decomp. → CminorSel → instruction selection → Cminor

register allocation (IRC) calling conventions

LTL → linearization of the CFG → Linear → layout of stack frames → Mach

Asm x86 → asm code generation → Asm ARM → Asm PPC

(form from CompCert web site)
CompCert Proof Structure

• Formal semantics for each IR
  • “adequacy” is a concern at endpoints

• Composition of preservation proofs for individual pipeline stages

• Mostly directly verified transformations, but some phases use translation validation
  • e.g. register allocation: much easier to validate an allocation solution (and prove the validator correct) than to prove precise spec for allocator
Forward Simulation Proofs

• Correctness of most phases is proven by establishing a simulation relation like this:

\[
\begin{align*}
\sigma, S \xrightarrow{t} \sigma' \\
\sim \quad \text{src} \\
\rho, T \xrightarrow{t} \rho' \\
\sim \\
\end{align*}
\]

S = src prog
T = target prog
σ = src state
ρ = target state

• Core of proof is defining state relation ~

• Each phase preserves the trace t of observable events (e.g. system calls)

• This strategy relies on languages being deterministic
CompCert Memory Model

• An important simplifying idea is to use the same memory model for all phases

• Memory is unbounded set of distinct blocks, each with individual bounds
  • each global, stack frame, and alloc gets own block
  • pointer arithmetic allowed only within blocks

• Although this simplification is a strength, it means that assembler semantics are less concrete than we might like...
CompCert status

• ca. 100K lines of Coq program and proof, 6 person years [as of 2018; somewhat more now]
• Some industrial users (e.g. Airbus)
• Many research groups have built on CompCert framework
  • optimizations
  • weak memory models
  • verified program analysis tools
Decompilation (1)

• Decompiling machine code [Myreen09, etc]
  • Build (certifiably) equivalent functional program
    • Each instruction becomes a sequence of updates and a collection of side conditions
    • Control flow is analyzed to discover loops
  • Can use to build a translation validator
    • Assuming we have effective automated equivalence checking between source & decompiled programs
    • Favors gray box approach
    • Limited support for optimization
Decompilation (2)

• Translation validation of seL4 [Sewell+13]
  • Used to transfer functional correctness proof from C to ARM machine code
  • Validated gcc compilation of 9500 C line kernel
    • almost 100% at –O1 (1 hour); about 55% at –O2 (4.5 hours)
  • C code and decompiled machine code both converted to a graph IR (unverified)
  • Equivalence of graph IRs checked by external SMT solvers (Z3 and SONOLAR).
Summary

• Verification of (new) production-quality compilers is well within reach today
• Verified translation validation is a promising technique for use with existing compilers
• Many foundational and engineering research challenges remain
• Why verify? To understand what you’re doing!
References (1)


References (2)


