Progress in Formal Verification of Compilers: A Survey

Andrew Tolmach
Portland State University
What?

• **Compiler:**
  
  ![Diagram](source_language_\rightarrow \text{Compiler} \rightarrow target_language)

• **Correctness:**
  
  if \( t = \text{compile}(s) \)
  
  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

Formal Verification of Compilers
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)
Verify or Check?

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

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

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

• Goal: A verified production-quality C compiler usable for critical embedded software
• Source language: a (large) subset 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

(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:

\[ \sigma, S \xrightarrow{t} \sigma' \quad S = \text{src prog} \]
\[ \rho, T \xrightarrow{t} * \rho' \quad \sigma = \text{src state} \quad \rho = \text{target state} \]

• Core of proof is defining state relation \( \sim \)
• Each phase preserves the trace \( t \) of observable events (e.g. system calls)
• This strategy relies on languages being deterministic

Formal Verification of Compilers
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. 50K lines of Coq proof, 8K lines of program, 4 person years [as of 2011; more now]
• Some industrial users (e.g. Airbus)
• Many research groups have built on CompCert framework
  • optimizations
  • weak memory models
  • verified program analysis tools
  • richer front-end languages
A Few CompCert Extensions

• Complex optimizations[Tristan09]
  • Lazy code motion, software pipelining
  • Uses (verified) translation validation
    • White box approach needed

• Static Single Assignment form[Demange12]
  • (Verified) translation validation for SSA minimization algorithm
  • Key lemma: variable definitions are equations that are valid within region dominated by definition
LLVM Work (1)

• Vellvm (Verified LLVM) [Zhao+12,13]
  • LLVM IR formalized in Coq
    • Testable against “real” IR by extracting interpreter
  • Verified version of mem2reg (generates minimal pruned SSA)
    • written in Coq, extracted, and plugged into real LLVM
  • General framework for reasoning about SSA form
  • About 50K lines of Coq proof script
LLVM Work (2)

- LLVM-MD [Tristan+11]
  - Translation validation on standard LLVM system
  - Based on normalization of value-graphs using heuristic collection of rewrite rules (gray box)
  - Gets about 80% checking success
  - (Unverified)
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).
High-Assurance Run-Time System

• PSU project to build RTS for high-level, safe languages (Java, Haskell, Habit, ...) with small trusted computing base
• Goal: make it credible for high-assurance systems to use these languages
• Implement minimalist RTS with essential services: garbage collection, concurrency, foreign function interfacing
Language-based approach

• Use compiler intermediate languages to package RTS services
• Language formal semantics specify intended behavior of services and clients
• Use semantics-preserving compilation to guarantee behavior of RTS implementation
  • building on CompCert
• Use type systems selectively to help guarantee that client code is well-behaved
CompCert-based RTS strategy

- High-level language source code
  - New language constructs for:
    - Managed heap allocation
    - Stacks and synchronization
    - Foreign calls
    - etc.

- Enriched Cminor-like intermediate code
  - Support code for:
    - Garbage collection
    - Thread scheduling
    - etc.

- Cminor code

- RTS library code (Cminor)

- Semantics-preserving transformations
Front-end assurance

High-level language source code

Typedness-preserving transformations

Strongly-typed intermediate code

Guarantees safety, not full behavior

Enriched Cminor-like intermediate code

RTS library code (Cminor)

Semantics-preserving transformations

Flexible but unsafe interfaces

Safe but restricted interfaces

Assembly code
Verifying GC service correctness

1. Prove correctness of GC algorithm and its implementation.
   • This is a proof about a particular program (in Cminor). Uses separation logic toolkit.

2. Prove that mutator and collector are correctly integrated:
   • agree about the set of roots and the locations of pointers within objects
   • respect each others’ private data structures

[McCreight+10]
GCminor

- Language formalizes mutator-collector interface with
  - `alloc` primitive
  - explicit GC root declarations
- Abstracts away details of GC implementation
- Compiler correctness guarantees apply
Summary

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


References (2)


