EECS 483: COMPILERS

Announcements

- Garter Part I:
  - Grades are out
  - If you had a major issue on your assignment, discuss with course staff at office hours and/or private piazza posts.
- Garter Part II:
  - Include updated spec as well as your executable tests/any other tests you saw fit to write.

POLL

IS IMPLEMENTING A CORRECT COMPILER HARD?

Empirical Evidence that Compiling is Hard

Egg-eater: 5/57 submissions passed 100% of autograder tests.

Not very scientific…

PLDI

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

Compiler Bugs

[Regehr's group: Yang et al. PLDI 2011]

Csmith
random
test-case generation

Source
Programs

GCC

79 bugs
(25 critical)

LLVM

202 bugs

…8 other C
compilers

325 bugs
in total

More recently:

- ALIVE/ALIVE2 projects
- miscompilation of C, Rust sources [Lee et al. OOPSLA 2018]

Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - "lint" tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound programming languages tools
  - "Formal" verification

Less "formal": Techniques may miss problems in programs

This isn't a tradeoff… all of these methods should be used.

Even "formal" methods can have holes:

- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth. "Beware of bugs in the above code; I have only proved it correct, not tried it."

More "formal": eliminate with certainty as many problems as possible.

Goal: Verified Software Correctness

- Social
  - Code reviews
  - Extreme/Pair programmin
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - “lint” tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound programming languages tools
  - “Formal” verification

Q: How can we move the needle towards mathematical software correctness properties?

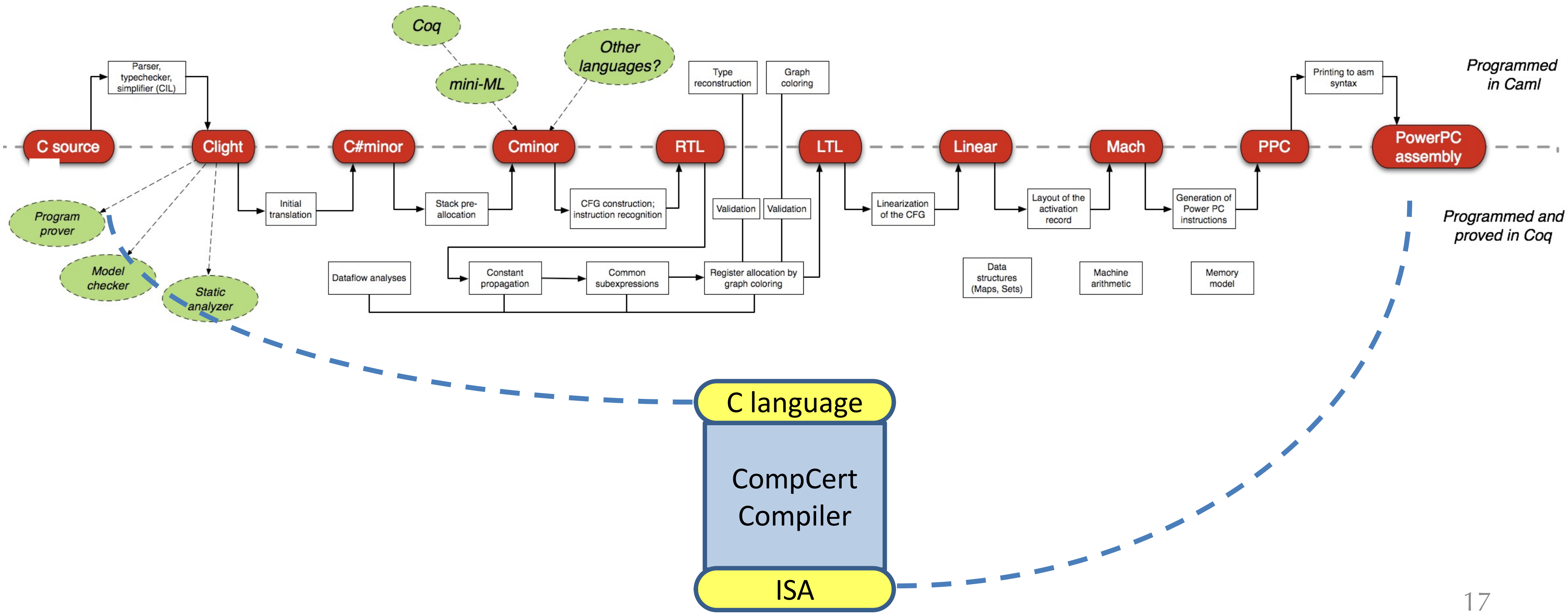
Taking advantage of advances in computer science:

- Moore's law
- improved programming languages & theoretical understanding
- better tools: interactive theorem provers

CompCert – A Verified C Compiler

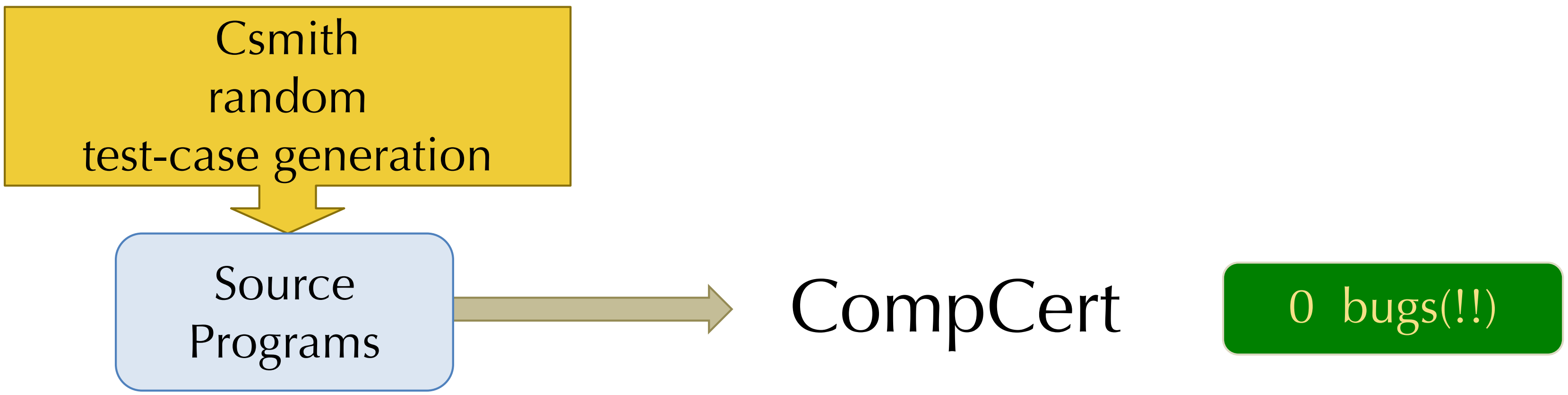
Xavier Leroy
INRIA

Optimizing C Compiler,
proved correct end-to-end
with machine-checked proof in Coq



Csmith on CompCert?

[Yang et al. PLDI 2011]



Verification Works!

"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.***"

– Regehr et. al 2011

Compiler Verification

Several components:

1. Specification: come up with precise specifications for when a compiler is correct.
   1. Functional correctness
   2. Security preservation, robustness to side-channel attacks
2. Proof: prove that a compilation technique satisfies the specification
3. Verification: computer-checked proof that a particular *implementation* is correct

Compiler Verification

Several components:

1. Specification: come up with precise specifications for when a compiler is correct.
    1. Functional correctness
    2. Security preservation, robustness to side-channel attacks
2. Proof: prove that a compilation technique satisfies the specification
3. Verification: computer-checked proof that a particular *implementation* is correct

PROVING BOA-- CORRECT

WRAPPING UP 483

What have we learned?

- Different phases of the compiler
  - Lexing/Parsing/Type Checking
- Different intermediate representations
- Interesting programming language features
  - Dynamic typing, heap-allocation, closures
- Meta lessons
  - How to work with an evolving codebase
  - Implementing programs with rich specifications

What we didn't get to cover

- Much more on parsing
  - PEGs, Earley Parsing,
- Macro systems
  - Preprocessors, LISP/Scheme/Rust-style of generative parsing
- Static Typing
  - Overloading, Traits/Typeclasses
- Interesting programming language features
  - Objects/Classes, concurrency/parallelism
- Interesting compilation techniques
  - JIT compilation, bytecode interpreters
- Other intermediate representations
  - SSA, Continuation-passing style
- Efficient data structures for compilation
- Runtime System features
  - Garbage collection, exceptions, debuggers

Where to learn more?

Classes at UM:

- EECS 583:
  - Graduate compilers. More focus on practical use of LLVM, reading research papers, implementing optimizations
- EECS 490 and 590:
  - Programming languages courses. More focus on Type Systems, programming language features, mathematical reasoning about programs

Open source projects

- Language implementations (e.g., Rust of course)
- Common compiler backends: LLVM, Cranelift, MLIR
- Compiler frontends: Tree-sitter, LALRPOP

Where to learn more?

Research at UM:

- Michigan Programming Languages and Software Engineering (MPLSE): mplse.org

Academic conferences

- PLDI (Programming Language Design and Implementation)
- POPL (Principles of Programming Languages)
- ICFP (Functional Programming)
- OOPSLA (Object-oriented …)
- CC (Compiler Construction)
- …and many more

Thanks!

- To course staff: Steven, Daniel
- To *you* for taking the class
- Feedback wanted:
  - Please fill out course evaluations so we can improve the course in the future