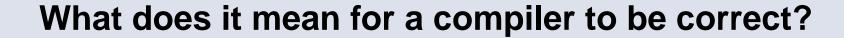
# **Proving Compilers Correct**

Formal Certification of a Compiler Back-end X. Leroy; POPL '06

Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules
S. Lerner, T. Millstein, E. Rice, and C. Chambers; POPL '05

Dan Licata Jake Donham



# **Compiler Soundness**

For all source programs, if property holds for source program and source compiles to target then analogous property holds for target

# **Compiler Soundness: Simulation**

For all source programs S, if S runs to a final memory M and S compiles to target program T then T runs to M

# **Compiler Soundness: Memory safety**

For all source programs S, if S is well-typed in source language and S compiles to target program T then T is memory-safe

# **Compiler Soundness: Memory safety**

For all source programs S, if (no assumptions about S) and S compiles to target program T then T is memory-safe

# **Prerequisites for Proving Soundness**

- Formal definition of target language (always)
- Formal definition of source language (most properties)
- Formal definition of compiler itself (algorithm or implementation)
- Proof environment: paper-and-pencil or computer-assisted

# Why Prove Compilers Correct?

- Eliminates bugs
- Can then prove properties of individual programs
- Cost is highly amortized

# Formal Certification of a Compiler Back-end X. Leroy; POPL '06

# Leroy '06: Summary

#### Certified compiler from Cminor to PowerPC:

- Operational semantics for Cminor and PPC
- Compiler implemented in Coq programming language/proof assistant
- Coq proof of simulation:
   if S runs to M and S compiles to T then T runs to M

#### **Cminor**

#### Designed to support most of C:

- All arithmetic types and operators, arrays, pointers, function pointers, pointer arithmetic, stack allocation, structured control
- No malloc/free (possible to add) or unstructured control (goto, switch, longjmp)

# Coq

Essentially an effect-free version of ML

Dependent types for specifications:

Theorem corresponds to a type:

$$\forall S, M, T. run_s(S, M) \times (compile(S) = T) \rightarrow run_t(T, M)$$

- Proof is another functional program
- Coq helps programmer write it

# **Compiler and Proof Architecture**

- Compiler passes compose
- So do proofs:
   if source is simulated by intermediate
   and intermediate is simulated by target
   then source is simulated by target

Compiler and proofs are decomposed into independent passes.

# Certified vs. Certifying

#### Certified compiler (this paper):

A compiler with a proof that it is sound on all inputs

#### Certifying compiler (Necula and Lee last week):

- On each run, compiler produces a proof that output satisfies soundness property
- Certificate can be checked by an external verifier

# Certified vs. Certifying

#### Certified compiler (this paper):

A compiler with a proof that it is sound on all inputs

#### Certifying compiler (Necula and Lee last week):

- On each run, compiler produces a proof that output satisfies soundness property
- Certificate can be checked by an external verifier

Sound different, right?

#### Not a Dime's Worth of Difference ...

Certifying compiler pass and certified compiler pass are interchangeable:

- -fying to -fied: run the compiler, then run the verifier on the certificate—works if you have a certified verifier.
- -fied to -fying: use the proof of correctness of the whole compiler to produce a certificate.

Freedom to choose certified or certifying for each pass.

# Okay, Maybe a Nickel or Two ....

- May be easier to prove verifier correct than to prove the algorithm correct
- Proving the algorithm correct may reveal other bugs (certified compiler might compile more programs)

Balance trade-offs for each pass.

#### **Passes**

- 1. Macro expansion
- 2. Block structure to CFG with infinite temporary registers
- 3. Optimizations (constant prop., CSE)
- 4. George-Appel register allocation
- 5. Linearize CFG
- 6. Lay out stack frames
- 7. Generate code

#### **Passes**

- 1. Macro expansion
- 2. Block structure to CFG with infinite temporary registers
- 3. Optimizations (constant prop., CSE)
- 4. George-Appel register allocation (Certifying)
- 5. Linearize CFG
- 6. Lay out stack frames
- 7. Generate code

#### **Numbers**

- Implementation: 6000 lines of Coq and Caml for compiler; 30,000 lines of Coq for specifications and proofs.
- Compilation time (a few small examples): half to double gcc -01.
- Performance (a few small examples): comparable to gcc -01.

# Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules

S. Lerner, T. Millstein, E. Rice, and C. Chambers POPL '05

# **Lerner '05: Summary**

Rhodium: domain-specific language for writing dataflow analyses and transformations

- Specify program by local propagation and transformation rules
- Automatically proved sound

Used to implement Andersen points-to analysis, arithmetic-invariant detection, loop-induction-variable strength reduction, ...

# **Rhodium Example: Analysis**

Relation mustNotPointTo(X : Var, Y : Var): the value of variable X is not a pointer to Y.

```
edge fact mustNotPointTo(X : Var, Y : Var) means \sigma(X) \neq \sigma(\&Y)
```

if  $stmt(X := \&Z) \land Y \neq Z$ then mustNotPointTo(X, Y)@out

if  $mustNotPointTo(X, Y)@in \land doesNotDef(X)$  then mustNotPointTo(X, Y)@out

#### **Local and Global Soundness**

#### Soundness of analyses is proved in two parts:

- A rule is sound if the meaning of the premise implies the meaning of the conclusion
  - checked with a decision procedure
- If all the rules are sound, then their fixed point is too
  - proved for all Rhodium programs once, by hand

#### **Rhodium Transformations**

Rules for replacing nodes of control-flow graph

Local soundness: for all states  $\sigma$  where transformation applies, new node has same meaning (action on  $\sigma$ ) as old node

 Somewhat restrictive; may be hard to express some non-local transformations

Global soundness proved by hand

#### Free stuff

- Flow-insensitive analyses
- Interprocedural analyses
- Execution engine: run Rhodium programs in compiler

All are automatically proved correct!

# **Big Picture**

A technique for writing dataflow analyses and transformations, not whole compilers

- Restricted language
- Smaller proof burden (fully automated)

Integrate into Leroy's compiler (?):

- implement Rhodium in Coq
- mechanize proofs of global soundness
- use certified/certifying decision procedure for local conditions

#### **Conclusion**

# Summary

- Proving a compiler correct requires formal definitions of the source and target languages
- Writing compilers in well-behaved languages is feasible
- ...and doing so enables you to prove theorems about them