A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language

Adam Chlipala
University of California, Berkeley
PLDI 2007
Why Certified Compilers?

Manual Code Auditing

Source Code

Static Analysis

Formal Methods

Compiled Machine Code

How do we know that these programs have the same behavior?

Compiler
$ cat tests/id.src
(\x : Nat, x)

$ bin/ctpc tests/id.src
block1:
r3 := r2.0
r4 := r2.1
r7 := r3
r6 := r1
r5 := r4
r1 := r6
r0 := r5
jump r7

main:
r1 := new([r0], [1])
r2 := 1
r3 := new([r1,r0], [r2,r1])
r4 := r0.0
r5 := r0.1
r8 := r4
r7 := r3
r6 := r5
r1 := r7
r0 := r6
jump r8

End Product

Simply-Typed Lambda Calculus program

Compiler

Idealized Assembly Language program

Contribution #1: Mechanizing proofs about type-preserving compilation

The Big Theorem:
(proved mechanically)

This commutative diagram holds for EVERY input program.
Coq Formalization

- Mechanized syntax and semantics for source language, target language, and 5 intermediate languages
- Type preservation
- Semantics preservation
Engineering a Correctness Proof

- Source Language Semantics
- Intermediate Language #1 Semantics
- Intermediate Language #2 Semantics
- Target Language Semantics

- Phase #1 Source
- Phase #2 Source
- Phase #N Source

- Phase #1 Proof Script
- Phase #2 Proof Script
- Phase #N Proof Script

- Automated Theorem Prover
- Hint Database
- Compiler Proof Library
- Correct runtime system

- Overall Correctness Proof
An Example

\[
\begin{align*}
type \ & ty = \\
& \quad \text{Int} \\
& \quad | \ Arrow \ of \ ty \ * \ ty
\end{align*}
\]

\[
\begin{align*}
type \ & exp = \\
& \quad \text{Const} \ of \ int \\
& \quad | \ Var \ of \ var \\
& \quad | \ Lambda \ of \ var \ * \ exp \\
& \quad | \ Apply \ of \ exp \ * \ exp
\end{align*}
\]

\[
\begin{align*}
type \ & lty = \\
& \quad \text{LInt} \\
& \quad | \ LArrow \ of \ lty \ * \ lty
\end{align*}
\]

\[
\begin{align*}
type \ & lexp = \\
& \quad \text{LConst} \ of \ int \\
& \quad | \ LVar \ of \ var \\
& \quad | \ LLambda \ of \ var \ * \ lexp \\
& \quad | \ LApply \ of \ exp \ * \ lexp \\
& \quad | \ LLet \ of \ var \ * \ lexp \ * \ lexp
\end{align*}
\]

Objective:

Compile \( lexps \) into \( exps \) using this identity:
\[
\text{let } x = e_1 \ \text{in } e_2 \ \simeq \ (\lambda x. \ e_2) \ e_1
\]
First Attempt

let rec compile e =
  match e with
  | LConst n -> Const n
  | LVar x -> Var x
  | LLambda (x, e') -> Lambda (x, compile e')
  | LApply (e1, e2) -> Apply (compile e1, compile e2)
  | LLet (x, e1, e2) ->

Proving that compile outputs well-typed programs....

I discovered a counterexample!
(after hours of frustration)

Compiler Source → Coq compiler → Compiler Binary → Certified compilation → Output Source #1

Input Source #2 → Output Source #2

Type Error!
Stating Our Assumptions

```plaintext
type ty =
    Int |
    Arrow of ty * ty

type (Γ, τ) exp =
    Const : int -> (Γ, Int) exp
    | Var : (Γ, τ) var -> (Γ, τ) exp
    | Lambda : ((Γ, x : τ1), τ2) exp
        -> (Γ, Arrow (τ1, τ2)) exp
    | Apply : (Γ, Arrow (τ1, τ2)) exp
        -> (Γ, τ1) exp -> (Γ, τ2) exp
```

Idea:

Represent expressions as their (strongly-typed) typing derivations
Second Attempt

```ocaml
let rec compile e =
  match e with
  | LConst n -> Const n
  | LVar x -> Var x
  | LLambda e' -> Lambda (compile e')
  | LApply (e1, e2) -> Apply (compile e1, compile e2)
  | LLet (e1, e2) -> Apply (Lambda (compile e1), compile e2)
```

This expression doesn't have the right type!
Dynamic Semantics

“Let” Language

\[ \text{LLet } (x, \text{Const } 1, \text{Var } x) \]

Base Language

\[ \text{LApp } ((\text{LLambda } (x, \text{Var } x), \text{Const } 1) \cdot x) 1 \]

Denotation Function

Common Core Language

Contribution #2:
Denotational semantics for proofs in type theory
(drawing on ideas related to GADTs and tagless interpreters)

Provable by the laws of the core language!
Inside a Proof

Correctness Theorem

Prove by induction on e.

Inductive step for "let":

IH1: \([\text{compile } e_1] \approx [e_1]\)

IH2: \([\text{compile } e_2] \approx [e_2]\)

\[\text{IH1}: [\text{compile } (\text{LLet } (x, e_1, e_2))] \approx [\text{App } (\text{Lambda } (x, \text{compile } e_2), \text{compile } e_1)]\]

Two simple operations form a base for automation: partial evaluation and rewriting

Contrast with, for example, CompCert project (Leroy 2006)
Implementation Statistics

Shift from lambda calculus to “three-address code”

Bottom bars show LoC that would remain in ML-style implementation

First reasoning about garbage collector interaction

Total: ~600 LOC uncertified vs. ~5000 LOC certified

(just implementation) (implementation + proofs)
Conclusion

- One more step toward mostly-automated correctness proofs for all of our compilers. :-)  

Code and documentation on the web at:
http://ltamer.sourceforge.net/
Key Innovations of This Work

• Proofs about a type-preserving compiler

• Dependently-typed abstract syntax
  – Static type checking ensures that compiler phases produce well-typed terms.

• Denotational semantics
  – ...as opposed to operational semantics used in most mechanized proofs

• Proof automation
Certified CPS Translation

in 250 lines
“Build Process”

Phase #1 Source → Phase #2 Source → ... → Phase #N Source

Coq Program Extraction

OCaml source of main compiler

OCaml source of parser

OCaml source of pretty-printer

OCaml compiler

Compiler Binary
Quick Tour of Useful Tricks

- Dependently-typed abstract syntax
- Denotational semantics
- Generic programming of variable-munging operations
Semantics by "Definitional Compilers"

Language #N

Phase #N

Language #(N+1)

Computable Denotation Function

Coq

Equivalent?

Coq
Generic Programming of Variable Manipulation

- **Abstract Syntax Tree**
- **Datatype**
- **Reflected Description**
- **Specialized Functions**
- **Generic Functions** (substitution, free variable calculation, etc.)
- **Generic Proofs** (commutativity of different operations, etc.)

Implemented in Coq such that static type checking guarantees compatibility for any original datatypes!

Static types show compatibility!
# Code/Proof Size Summary

<table>
<thead>
<tr>
<th>Component</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>31</td>
</tr>
<tr>
<td>...to...</td>
<td>116</td>
</tr>
<tr>
<td>Linear</td>
<td>56</td>
</tr>
<tr>
<td>...to...</td>
<td>115</td>
</tr>
<tr>
<td>CPS</td>
<td>87</td>
</tr>
<tr>
<td>...to...</td>
<td>646</td>
</tr>
<tr>
<td>CC</td>
<td>185</td>
</tr>
<tr>
<td>...to...</td>
<td>1321</td>
</tr>
<tr>
<td>Alloc</td>
<td>217</td>
</tr>
<tr>
<td>...to...</td>
<td>658</td>
</tr>
<tr>
<td>Flat</td>
<td>141</td>
</tr>
<tr>
<td>...to...</td>
<td>868</td>
</tr>
<tr>
<td>Asm</td>
<td>111</td>
</tr>
<tr>
<td>Dictionaries</td>
<td>119</td>
</tr>
<tr>
<td>Traces</td>
<td>96</td>
</tr>
<tr>
<td>GC Safety</td>
<td>741</td>
</tr>
<tr>
<td>Glue code</td>
<td>119</td>
</tr>
</tbody>
</table>

**PL Formalization Library:**
3520 lines of Coq
2716 lines of OCaml
Greedy Quantifier Instantiation

Expressions appearing in proof state

- \( n : \text{int} \)
- \( t1 : \text{type} \)
- \( t2 : \text{type} \)
- \( e1 : t1 \exp \)
- \( e3 : (t1 \ast t2) \exp \)
- \( e2 : t2 \exp \)

\[ \exists x : t2 \exp, \text{foo}(x) \]
Good News

Step 1: Simplify using the definition of compile

Step 2: Simplify using the defn. of [] for “let” language

Step 3: Simplify using the defn. of [] for base language

Step 4: Simplify using core language semantics

Step 5: Apply IH2

Step 6: Use known fact $\sigma \sim \sigma'$

Step 7: Apply IH1

Partial Evaluation

This is one of the fundamental operations of theorem proving in Coq!

Logic Programming

Rich types make this relatively easy to automate!
Wish List

• Semantics approach with better support for “impure” features
  – Mutable references and arrays
  – Non-termination
  – General recursive types
• Easier *dependently-typed programming*
• Better *proof automation*
  – (Probably mostly domain-specific)