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

An experiment with variable binding, denotational semantics, and logical relations in Coq

Adam Chlipala
University of California, Berkeley
Certifying compilation:
Source and target programs are observationally equivalent.

Certified compiler:
For any valid input, the compiler produces an observationally equivalent output.

Transformations:
CPS conversion, closure conversion, explicit heap allocation, register allocation, ...
Type-Preserving Compilation

• Preserve static type information in some prefix of the compilation process.

• Taken all the way, you end up with typed assembly language, proof-carrying code, etc..

• More modestly, implement nearly tag-free garbage collection.
  – Replace tag bits, boxing, etc., with static tables mapping registers to types.
  – Used in the MLton SML compiler.
What's tricky?

- Nested variable scopes
- Relational reasoning
- Proof management and automation

This is what the **POPLmark Challenge** is all about!
Design Decision #1: Dependently-Typed ASTs

Use dependent types to make the compiler type-preserving by construction!

Type Preservation Theorem. If the input program has type $T$, then the output program has type $C(T)$.

Semantics Preservation Theorem. If the input program has meaning $M$, then the output program has meaning $C(M)$. 
Design Decision #2: Denotational Semantics

Denotational Semantics Version:
1. Compile the input program to CIC.
2. Compile the output program to CIC.
3. The two results must be equal.

Semantics Preservation Theorem.
If the input program has meaning $M$, then the output program has meaning $C(M)$. 
Secret Weapons

Programming with dependent types is hard!

The trickiest bits deal with "administrative" operations that adjust variable bindings... but these are still routine and hardly language-specific!

Writing formal proofs is hard!

The combination of **dependent types** and **denotational semantics** enables some very effective **decision procedures** to be coded in Coq's **Ltac language**.

"Put the rooster to work!"
Rest of the Talk...

- Summary of compilation
- Dependently-typed ASTs
- Denotational semantics in Coq
- Writing compiler passes
  - …including generic programming of helper functions
- Proving semantics preservation
Source and Target Languages

Source language: **simply-typed lambda calculus**

\[ \tau ::= N \mid \tau \rightarrow \tau \]
\[ e ::= n \mid x \mid e \ e \mid \lambda x : \tau, e \]

Target language: **idealized assembly language**

\[ o ::= r \mid n \mid \text{new}(R, R) \mid \text{read}(r, n) \]
\[ i ::= r := o; i \mid \text{jump} \ r \]
\[ p ::= (l, i) \]
Compiler Stages

\[ \lambda x, f x \]

**CPS conversion**

\[ k_{top}(\lambda x, \lambda k, f x k) \]

**Closure conversion**

\[
\text{let } F = \lambda e, \lambda x, \lambda k, e.1 x k \text{ in } k_{top}(\langle F, [f] \rangle) \]

**Explicit heap allocation**

\[
\text{let } F = \lambda e, \lambda x, \lambda k, e.1.1 e.1.2 x k \text{ in } \text{let } r1 = [f] \text{ in let } r2 = [F, r1] \text{ in } k_{top}(r2) \]

**Flattening**

\[
F: r4 := r1.1; r1 := r4.2; r4 := r4.1; \textbf{jump } r4 \\
main: r3 := r1.1; r1 := r1.2; \\
\text{r2 := new } [f]; r2 := \text{new } [F, r2]; \textbf{jump } r3
\]
Correctness Proof

- Compiler and proof implemented entirely within **Coq 8.0**

- Axioms:
  - Functional extensionality:
    \[ \forall f, g, (\forall x, f(x) = g(x)) \Rightarrow f = g \]
  - Uniqueness of equality proofs:
    \[ \forall \tau, \forall x, y : \tau, \forall P1, P2 : x = y, P1 = P2 \]

- The compiler is *almost* runnable as part of a proof.
Denotational Semantics of the Source Language

\[ [\tau] : \text{types} \rightarrow \text{sets} \]
\[ [N] = \mathbb{N} \]
\[ [\tau_1 \rightarrow \tau_2] = [\tau_1] \rightarrow [\tau_2] \]

\[ [\Gamma] : \text{contexts} \rightarrow \text{sets} \]
\[ [\cdot] = \text{unit} \]
\[ [\Gamma, x : \tau] = [\Gamma] \times [\tau] \]

\[ [e] : [\Gamma \vdash e : \tau] \rightarrow [\Gamma] \rightarrow [\tau] \]
\[ [n]_\sigma = \overline{n} \]
\[ [x]_\sigma = \sigma(x) \]
\[ [e_1 \ e_2]_\sigma = [e_1]_\sigma [e_2]_\sigma \]
\[ [\lambda x : \tau, e]_\sigma = \lambda x : [\tau], [e](\sigma, x) \]
For Types...

**Inductive** ty : Set :=
  | Nat : ty
  | Arrow : ty -> ty -> ty.

**Fixpoint** tyDenote (t : ty) : Set :=
  match t with
  | Nat => nat
  | Arrow t1 t2 => tyDenote t1 -> tyDenote t2
  end.
Representing Terms

Nominal syntax

\textbf{Inductive} \texttt{term} : \texttt{Set} :=

| Const : nat -> term |
| Var : name -> term |
| Lam : name -> term -> term |
Representing Terms

*De Bruijn syntax*

**Inductive** \( \text{term} : \text{Set} := \)**

\[
\begin{align*}
| & \text{Const} : \text{nat} \to \text{term} \\
| & \text{Var} : \text{nat} \to \text{term} \\
| & \text{Lam} : \text{term} \to \text{term} \\
| & \text{App} : \text{term} \to \text{term} \to \text{term}.
\end{align*}
\]
Representing Terms

Dependent de Bruijn syntax

Inductive term : nat -> Set :=
| Const : forall n, nat -> term n
| Var : forall n x, x < n -> term n
| Lam : forall n, term (S n) -> term n
| App : forall n, term n -> term n -> term n.
Representing Terms

Dependent de Bruijn syntax with typing

**Inductive** `term : list ty -> ty -> Set :=`

| Const : forall `G : nat -> term G Nat` |
| Var : forall `G t : var G t -> term G t` |
| Lam : forall `G dom ran : term (dom :: G) ran -> term G (Arrow dom ran)` |
| App : forall `G dom ran : term G (Arrow dom ran) -> term G dom -> term G ran` |
Term Denotations

**Fixpoint** termDenote \( (G : \text{list ty}) \ (t : \text{ty}) \ (e : \text{term G t}) \) \{struct e\} : subst tyDenote \( G \rightarrow\) tyDenote \( t := \)

**match** \( e \ \text{in} \ (\text{term} \ G \ t) \)

**return** (subst tyDenote \( G \rightarrow\) tyDenote \( t \)) **with**

1. Const _ _ \( n \rightarrow \text{fun} _ \rightarrow n \)
2. Var _ _ \( x \rightarrow \text{fun} s \rightarrow \text{varDenote} \ x \ s \)
3. Lam _ _ \( e' \rightarrow \text{fun} s \rightarrow \text{fun} x \rightarrow \text{termDenote} \ e' \ (\text{SCons} \ x \ s) \)
4. App _ _ _ \( e1 \ e2 \rightarrow \text{fun} s \rightarrow \) (termDenote \( e1 \ s \)) (termDenote \( e2 \ s \))

**end.**
Definition of “Values” for Free Operational

\[ n \text{ value} \]

\[ \lambda x : \tau, \ e \text{ value} \]

Syntactic characterization used throughout definitions and proofs

Inherit any “canonical forms” properties of the underlying Coq types.

“A natural number is either zero or a successor of another natural number.”

Caveat: We don't get the same kind of property for functions!

\[
\begin{align*}
[\tau] : \text{types} & \rightarrow \text{sets} \\
[N] &= \mathbb{N} \\
[\tau_1 \rightarrow \tau_2] &= [\tau_1] \rightarrow [\tau_2]
\end{align*}
\]
No Substitution Function!

Operational

\[
\begin{align*}
n[x := e] &= n \\
x[x := e] &= e \\
y[x := e] &= y \\
(e_1 e_2)[x := e] &= e_1[x := e] e_2[x := e] \\
(\lambda x : \tau, e')[x := e] &= \lambda x : \tau, e' \\
(\lambda y : \tau, e')[x := e] &= \lambda y : \tau, e'[x := e]
\end{align*}
\]

Customized syntactic substitution function written for each object language

\[
(\lambda x : \tau, e_1) e_2 \rightarrow e_1[x := e_2]
\]

Reduction rules defined using substitution

Denotational

\[
\begin{align*}
[n]\sigma &= \overline{n} \\
[x]\sigma &= \sigma(x) \\
[e_1 e_2]\sigma &= [e_1]\sigma [e_2]\sigma \\
[\lambda x : \tau, e]\sigma &= \lambda x : [\tau], [e](\sigma, x)
\end{align*}
\]

\[
[(\lambda x : \mathbb{N}, x) 1]() = [\lambda x : \mathbb{N}, x]() [1]() \\
= (\lambda x : \mathbb{N}, x) 1 \\
= 1
\]

Coq's operational semantics provides the substitution operation for us!
Free Metatheorems

Operational

For each object language, give customized, syntactic proofs of properties like:

- Type safety – preservation
- Type safety – progress
- Confluence
- Strong normalization
- ...

Denotational

Meta-theorems proved once and for all about CIC

The majority of programming language theory mechanization experiments only look at proving these sorts of theorems!
Free Theorems

Theorem 1 For any $n$, $\llbracket (\lambda x : \mathbb{N}, x) \ n \rrbracket () = n$.

Proof. By \textit{reflexivity of equality}.

Coq's proof checker identifies as \textbf{equivalent} terms that reduce to each other! This means that both \textbf{compilation of terms into CIC} and \textbf{evaluation of the results} are “zero cost” operations.
But Wait!

Doesn't that only work for languages that are:

- Strongly normalizing
- Purely functional
- Deterministic
- Single-threaded
- ...etc...

(In other words, a lot like Coq)
Monads to the Rescue!

- Summary rebuttal: Take a cue from Haskell.
- Use **object language agnostic** "embedded languages" to allow expression of "effectful" computations
- Keep using Coq's definitional equality to handle reasoning about pure sublanguages, and even some of the mechanics of impure pieces.
Non-Strongly-Normalizing Languages

For closed, first-order programs with basic block structure (e.g., structured assembly)

A total denotation function that executes a basic block, determining the next program counter and memory state.

A function runs basic blocks repeatedly to build a lazy list describing an execution trace. (no “non-computational” definitions required)
Co-inductive Traces

\[ T ::= n \mid \bot \mid \star, T \]

By keeping only these summaries of program executions, we enable effective equality reasoning.

Example: Garbage collection safety
Equality of traces is a good way to characterize the appropriate effect on programs from rearranging the heap and root pointers to a new, isomorphic configuration.
Example Compilation Phase: CPS Transform

Recall that terms are represented as typing derivations.

We need a syntactic helper function equivalent to a weakening lemma.
Dependently-Typed Syntactic Helper Functions?

• Could just write this function from scratch for each new language.
  – Probably using tactic-based proof search
  – The brave (and patient) write the CIC term directly.
    • My recipe for writing generic substitution functions involves three auxiliary recursive functions!

• Much nicer to automate these details using generic programming!
  – Write each function once, not once per object language.
What Do We Need?

1. The helper function itself

\[
\text{weaken} : \text{forall} \ (G : \text{list ty}) \ (t : \text{ty}), \ \text{term} \ G \ t \to \text{forall} \ (t' : \text{ty}), \ \text{term} \ (t' :: G) \ t
\]

2. Lemmas about the function

For any term \(e\), properly-typed substitution \(\sigma\), and properly-typed value \(\nu\):

\[
\left[\text{weaken}(e)\right](\sigma, \nu) = \left[e\right]_\sigma
\]

Can prove this generically for any compositional denotation function! For example, for simply-typed lambda calculus, there must exist \(f_{\text{var}}\), \(f_{\text{app}}\), and \(f_{\text{lam}}\) such that:

\[
\left[x\right]_\sigma = f_{\text{var}}(\sigma(x))
\]

\[
\left[e_1 \ e_2\right]_\sigma = f_{\text{app}}(\left[e_1\right]_\sigma, \left[e_2\right]_\sigma)
\]

\[
\left[\lambda x : \tau, \ e\right]_\sigma = f_{\text{lam}}(\lambda x : \left[\tau\right], \left[e\right](\sigma, x))
\]
Reflection-Based Generic Programming

**Language Definition**
(Coq inductive type)

**Coq plug-in**
(outside the logic)

**Reflected Language Definition**
(term of CIC)

**Denotation Function**

**Coq plug-in**

**Reflected Denotation Function**

**Generic proof**

**Specific function**
(type-compatible with original language definition)

**Generic function**

**Specific proof**
What to Prove?

*Overall correctness theorem:* The compilation of a program of type \( \mathbb{N} \) runs to the same result as the original program does.

What do we prove about individual phases?

Prove that input/output pairs are in an appropriate *logical relation*. E.g., for the CPS transform:

\[
\begin{align*}
    n_1 \simeq_{\mathbb{N}} n_2 &= n_1 = n_2 \\
    f_1 \simeq_{\tau_1 \rightarrow \tau_2} f_2 &= \forall x_1 : [\tau_1]^S, \forall x_2 : [\tau_1]^L, x_1 \simeq_{\tau_1} x_2 \\
    &\quad \rightarrow \exists v : [\tau_2]^L, \forall k : [\tau_2]^L \rightarrow \mathbb{N}, \\
    f_1 x_1 \simeq_{\tau_2} v
\end{align*}
\]

This function space contains many functions not representable in our object languages!
Easy first step: Use introduction rules for forall's and implications at the start of the goal.
In the Trenches

Now we're blocked at the tricky point for automated provers: proving existential facts and applying universal facts.

Key observation: The quantified variables have very specific dependent types.

We can use greedy quantifier instantiation!

Now we're blocked at the tricky point for automated provers: proving existential facts and applying universal facts.
In the Trenches

\[ H_1 : \forall \sigma^S, \forall \sigma^L, \sigma^S \preceq_\Gamma \sigma^L \rightarrow \exists v : [\tau_1 \rightarrow \tau_2]^L, \]

\[ \forall k : [\tau_1 \rightarrow \tau_2]^L \rightarrow \mathbb{N}, [[e_1]]^L \sigma^L k = k v \land [e_1]^S \sigma^S \simeq_{\tau_1 \rightarrow \tau_2} v \]

\[ H_2 : \forall \sigma^S, \forall \sigma^L, \sigma^S \preceq_\Gamma \sigma^L \rightarrow \exists v : [\tau_1]^L, \]

\[ \forall k : [\tau_1]^L \rightarrow \mathbb{N}, [[e_2]]^L \sigma^L k = k v \land [e_2]^S \sigma^S \simeq_{\tau_1} v \]

\[ \sigma^S : \ldots \]

\[ \sigma^L : \ldots \]

\[ H_3 : \sigma^S \preceq_\Gamma \sigma^L \]

\[ \exists v : [\tau_2]^L, \]

\[ \forall k : [\tau_2]^L \rightarrow \mathbb{N}, [[e_1 \ e_2]]^L \sigma^L k = k v \land [e_1 \ e_2]^S \sigma^S \simeq_{\tau_2} v \]
In the Trenches

\[ H_1 : \exists v : \left[ \tau_1 \rightarrow \tau_2 \right]^L, \]
\[ \forall k : \left[ \tau_1 \rightarrow \tau_2 \right]^L \rightarrow \mathbb{N}, \left[ \left[ e_1 \right] \right]^L \sigma^L k = k v \land \left[ e_1 \right]^S \sigma^S \simeq_{\tau_1 \rightarrow \tau_2} v \]
\[ H_2 : \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_{\Gamma} \sigma^L \rightarrow \exists v : \left[ \tau_1 \right]^L, \]
\[ \forall k : \left[ \tau_1 \right]^L \rightarrow \mathbb{N}, \left[ \left[ e_2 \right] \right]^L \sigma^L k = k v \land \left[ e_2 \right]^S \sigma^S \simeq_{\tau_1} v \]
\[ \sigma^S : \ldots \]
\[ \sigma^L : \ldots \]
\[ H_3 : \sigma^S \simeq_{\Gamma} \sigma^L \]
\[ \exists v : \left[ \tau_2 \right]^L, \]
\[ \forall k : \left[ \tau_2 \right]^L \rightarrow \mathbb{N}, \left[ \left[ e_1 \ e_2 \right] \right]^L \sigma^L k = k v \land \left[ e_1 \ e_2 \right]^S \sigma^S \simeq_{\tau_2} v \]

Existential hypotheses are easy to eliminate.
In the Trenches

We can't make further progress with this hypothesis, since no term of the type given for $k$ exists in the proof state.
We can simplify the conclusion by applying rewrite rules (like those we generated automatically) until no more apply.
In the Trenches

$$v_1 : [\tau_1 \rightarrow \tau_2]^L$$

$$H_1 : \forall k : [\tau_1 \rightarrow \tau_2]^L \rightarrow \mathbb{N}, \llbracket e_1 \rrbracket^L \sigma^L k = k v_1 \land \llbracket e_1 \rrbracket^S \sigma^S \simeq_{\tau_1 \rightarrow \tau_2} v_1$$

$$v_2 : [\tau_1]^L$$

$$H_2 : \forall k : [\tau_1]^L \rightarrow \mathbb{N}, \llbracket e_2 \rrbracket^L \sigma^L k = k v_2 \land \llbracket e_2 \rrbracket^S \sigma^S \simeq_{\tau_1} v_2$$

$$\sigma^S : \ldots$$

$$\sigma^L : \ldots$$

$$H_3 : \sigma^S \simeq_\Gamma \sigma^L$$

$$\exists v : [\tau_2]^L,$$

$$\forall k : [\tau_2]^L \rightarrow \mathbb{N}, \llbracket e_1 \rrbracket^L \sigma^L(\lambda x : [\tau_1 \rightarrow \tau_2]^L, \ldots) \geq \ldots \land \ldots$$

Now the conclusion has a subterm with the right type to instantiate a hypothesis!
In the Trenches

We can use $H_1$ to rewrite the goal.
In the Trenches

\[ v_1 : [\tau_1 \rightarrow \tau_2]^L \]
\[ H_1 : [[e_1]]^L \sigma^L (\lambda x : [\tau_1 \rightarrow \tau_2]^L, ...) = (\lambda x : [\tau_1 \rightarrow \tau_2]^L, ...) v_1 \wedge \ldots \]
\[ v_2 : [\tau_1]^L \]
\[ H_2 : \forall k : [\tau_1]^L \rightarrow \mathbb{N}, [[e_2]]^L \sigma^L k = k \cdot v_2 \wedge [e_2]^S \sigma^S \simeq_{\tau_1} v_2 \]
\[ \sigma^S : \ldots \]
\[ \sigma^L : \ldots \]
\[ H_3 : \sigma^S \simeq_{\Gamma} \sigma^L \]
\[ \exists v : [\tau_2]^L, \forall k : [\tau_2]^L \rightarrow \mathbb{N}, (\lambda x : [\tau_1 \rightarrow \tau_2]^L, ...) v_1 = \ldots \wedge \ldots \]
And That's That!

- This strategy does almost all of the proving for the CPS transformation correctness proof!
  - About 20 lines of proof script total.

- Basic approach:
  - Figure out the right syntactic rewrite lemmas, prove them, and add them as hints.
  - State the induction principle to use.
  - Call a generic tactic from a library.
A Recipe for Certified Compilers

1. Define object languages with dependently-typed ASTs.
2. Give object languages denotational semantics.
3. Use generic programming to build basic support functions and lemmas.
4. Write compiler phases as dependently-typed Coq functions.
5. Express phase correctness with logical relations.
6. Prove correctness theorems using a generic decision procedure relying heavily on greedy quantifier instantiation.
Design Decisions

• Why dependently-typed ASTs?
  – Avoid well-formedness side conditions
  – Easy to construct denotational semantics defined only over well-typed terms
  – Makes greedy quantifier instantiation realistic

• Why denotational semantics?
  – Concise to define
  – Known to work well with code transformation
  – Many reasoning steps come for free via Coq's definitional equality
Conclusion

- Yet another bag of suggestions on how to formalize programming languages and their metatheories and tools!
- Would be interesting to see other approaches to formalizing this kind of compilation.

Acknowledgements

Thanks to my advisor George Necula.

This work was funded by a US National Defense fellowship and the US National Science Foundation.