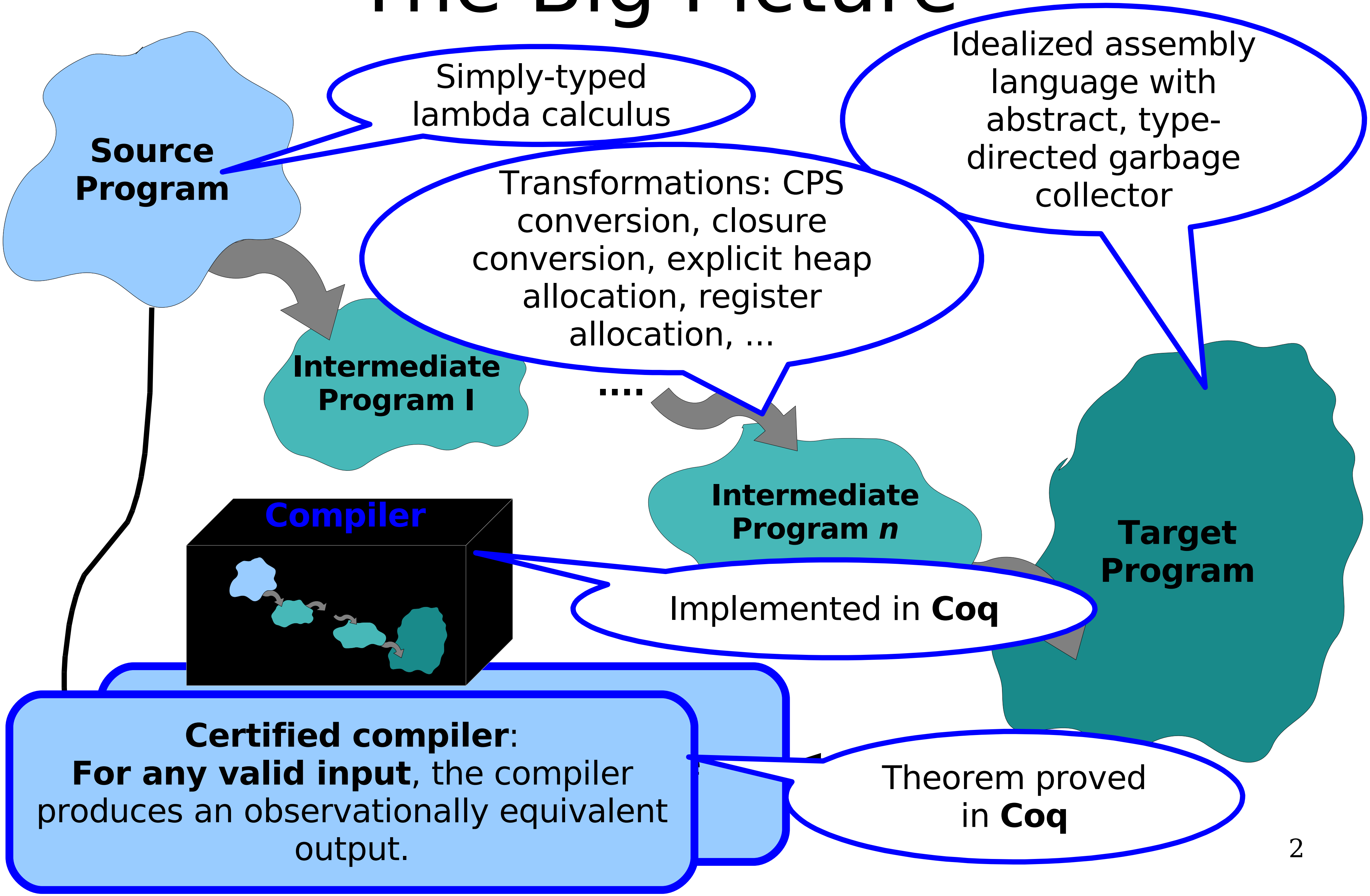
A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language

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

Adam Chlipala
University of California, Berkeley

The Big Picture



Type-Preserving Compilation

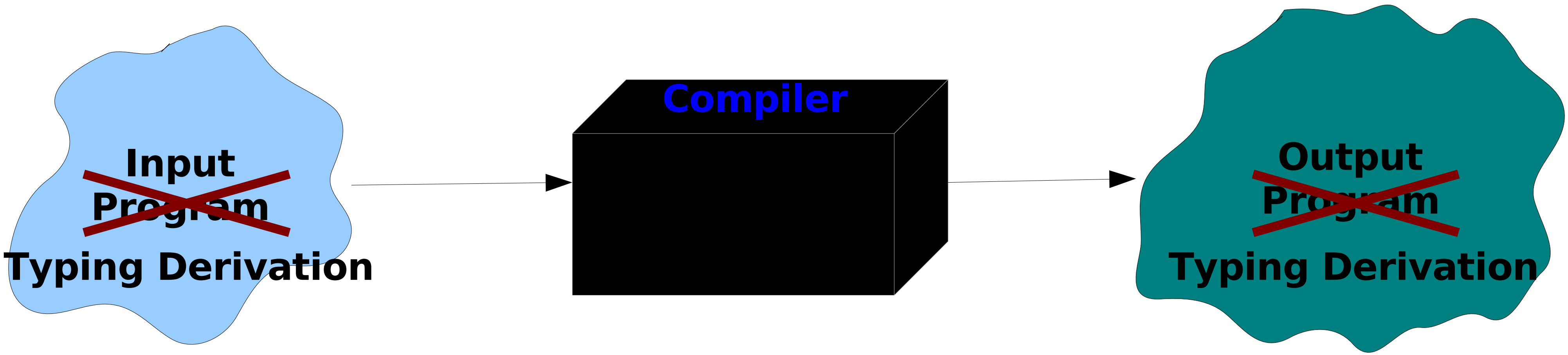
- Preserve **static type information** in some prefix of the compilation process.
- Taken all the way, you end up with **typed assembly language**, **proof-carrying code**, etc..
- More modestly, implement **nearly tag-free garbage collection.**
  - Replace tag bits, boxing, etc., with static tables mapping registers to types.
  - Used in the MLton SML compiler.

What's tricky?

- **Nested variable scopes**
- **Relational reas ning**
- **Proof manageme l automation**

This is what the **POPLmark Challenge** is all about!

Design Decision #1: Dependently-Typed ASTs

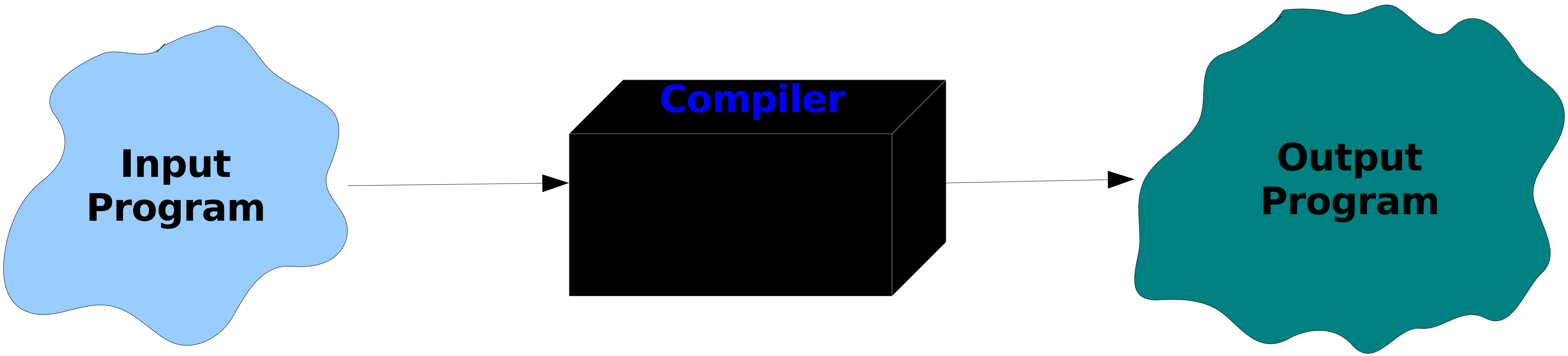


Use **dependent types** to make the compiler **type-preserving by construction**!

~~**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



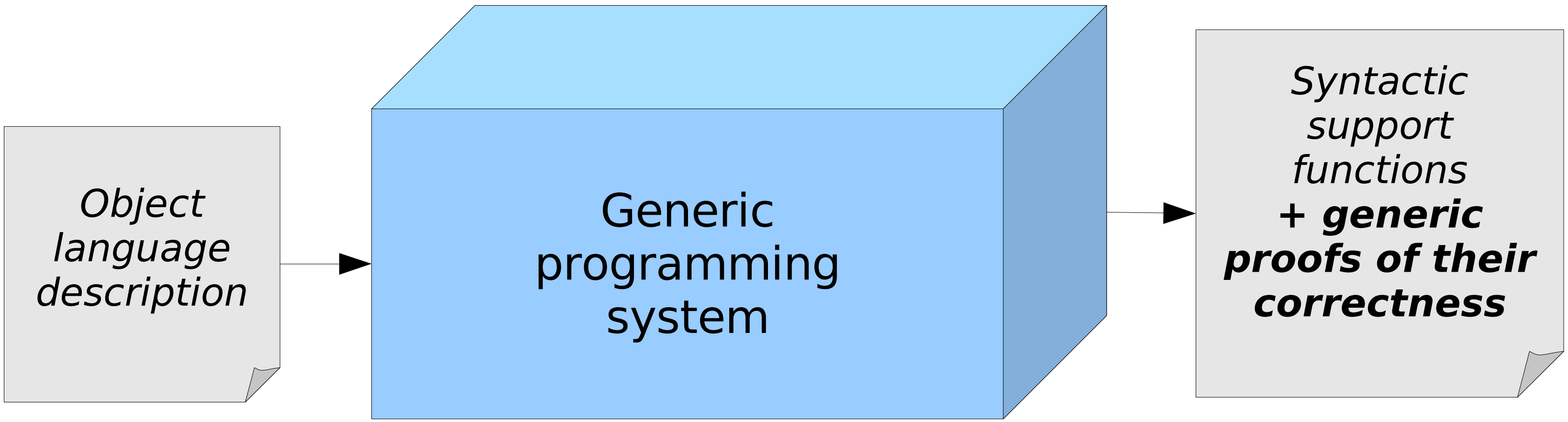
Denotational Semantics Version:

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 $\mathbf{C}(M)$.

Secret Weapons

Programming with dependent types is hard!



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**

$\tau ::= \mathbf{N} \mid \tau \to \tau$

$e ::= n \mid x \mid e\ e \mid \lambda x \mathbin{\mathbf{:}} \tau\mathbf{,}\ e$

Target language: **idealized assembly language**

$o ::= r \mid n \mid \mathbf{new(}R\mathbf{,}\ R\mathbf{)} \mid \mathbf{read(}r\mathbf{,}\ n\mathbf{)}$

$i\ ::= r \mathbin{\mathbf{:=}} o\mathbf{;}\ i \mid \mathbf{jump}\ r$

$p ::= \mathbf{(}I\mathbf{,}\ i\mathbf{)}$

Compiler Stages

λx, f x

CPS conversion

$\mathrm{k}_{top}(\lambda x,\ \lambda k,\ \mathrm{f}\ x\ k)$

Closure conversion

let F = λe, λx, λk, e.1 x k **in** $\mathrm{k}_{top}(\langle \mathrm{F}, [\mathrm{f}] \rangle)$

Explicit heap allocation

let F = λe, λx, λk, e.1.1 e.1.2 x k **in**
let r1 = [f] **in let** r2 = [F, r1] **in** $\mathrm{k}_{top}(\mathrm{r2})$

Flattening

F: r4 := r1.1; r1 := r4.2; r4 := r4.1; **jump** r4
main: r3 := r1.1; r1 := r1.2;
r2 := **new** [f]; r2 := new [*F*, r2]; **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 : τ, $\forall$P1, P2 : x = y, P1 = P2

- The compiler is *almost* runnable as part of a proof.

Denotational Semantics of the Source Language

$$\begin{aligned}
[\![\tau]\!] &: \text{types} \longrightarrow \text{sets} \\
[\![\mathsf{N}]\!] &= \mathbb{N} \\
[\![\tau_1 \to \tau_2]\!] &= [\![\tau_1]\!] \to [\![\tau_2]\!]
\end{aligned}$$

$$\begin{aligned}
[\![\Gamma]\!] &: \text{contexts} \longrightarrow \text{sets} \\
[\![\cdot]\!] &= \mathsf{unit} \\
[\![\Gamma, x : \tau]\!] &= [\![\Gamma]\!] \times [\![\tau]\!]
\end{aligned}$$

$$\begin{aligned}
[\![e]\!] &: [\Gamma \vdash e : \tau] \to [\![\Gamma]\!] \to [\![\tau]\!] \\
[\![n]\!]\sigma &= \overline{n} \\
[\![x]\!]\sigma &= \sigma(x) \\
[\![e_1\ e_2]\!]\sigma &= [\![e_1]\!]\sigma\ [\![e_2]\!]\sigma \\
[\![\lambda x : \tau, e]\!]\sigma &= \lambda x : [\![\tau]\!], [\![e]\!](\sigma, x)
\end{aligned}$$

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

```
Inductive term : Set :=
  | Const : nat -> term
  | Var : name -> term
  | Lam : name -> term -> term
  | App : term -> term -> term.
```

Representing Terms

De Bruijn syntax

```
Inductive term : Set :=
  | Const : nat -> term
  | Var : nat -> term
  | Lam :             term -> term
  | App : term -> term -> term.
```

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 : list ty) (t : ty) (e : term G t) {struct e}
  : subst tyDenote G -> tyDenote t :=
  match e in (term G t)
        return (subst tyDenote G -> tyDenote t) with
    | Const _ n => fun _ => n
    | Var _ _ x => fun s => varDenote x s
    | Lam _ _ _ e' => fun s =>
      fun x => termDenote e' (SCons x s)
    | App _ _ _ e1 e2 => fun s =>
      (termDenote e1 s) (termDenote e2 s)
  end.
```

Definition of “Values” for Free

Operational

$$\frac{}{n \ \textbf{value}}$$

$$\frac{}{\lambda x : \tau, e \ \textbf{value}}$$

Syntactic characterization used throughout definitions and proofs

Denotational

$$\begin{aligned} [\![\tau]\!] &: \text{types} \to \text{sets} \\ [\![\mathsf{N}]\!] &= \mathbb{N} \\ [\![\tau_1 \to \tau_2]\!] &= [\![\tau_1]\!] \to [\![\tau_2]\!] \end{aligned}$$

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!

No Substitution Function!

Operational

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

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

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

$$
\begin{aligned}
[\![(\lambda x : \mathsf{N}, x)\ 1]\!]() &= [\![\lambda x : \mathsf{N}, x]\!]()\ [\![1]\!]() \\
&= (\lambda x : \mathbb{N}, x)\ 1 \\
&= 1
\end{aligned}
$$

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

Free Metatheorems

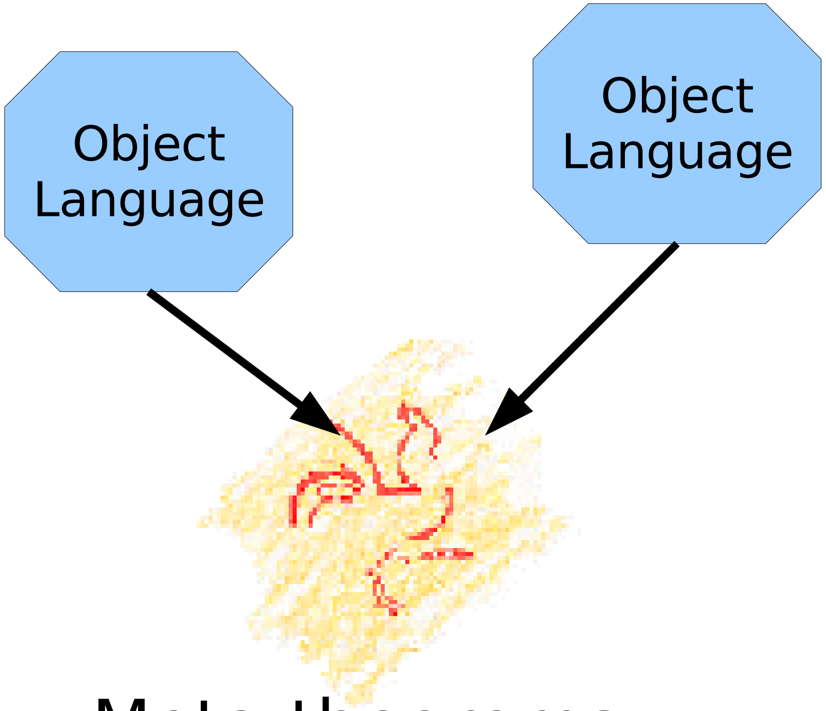
Operational

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

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

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

Denotational



Meta-theorems proved once and for all about CIC

Free Theorems

Theorem 1 *For any* n, $[\![(\lambda x : \mathsf{N}, x)\ n]\!]() = n$.

Proof. By ***reflexivity of equality***.

Coq's proof checker identifies as **equivalent** terms that reduce to each other!
This means that both **compilation of terms into CIC** and **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...

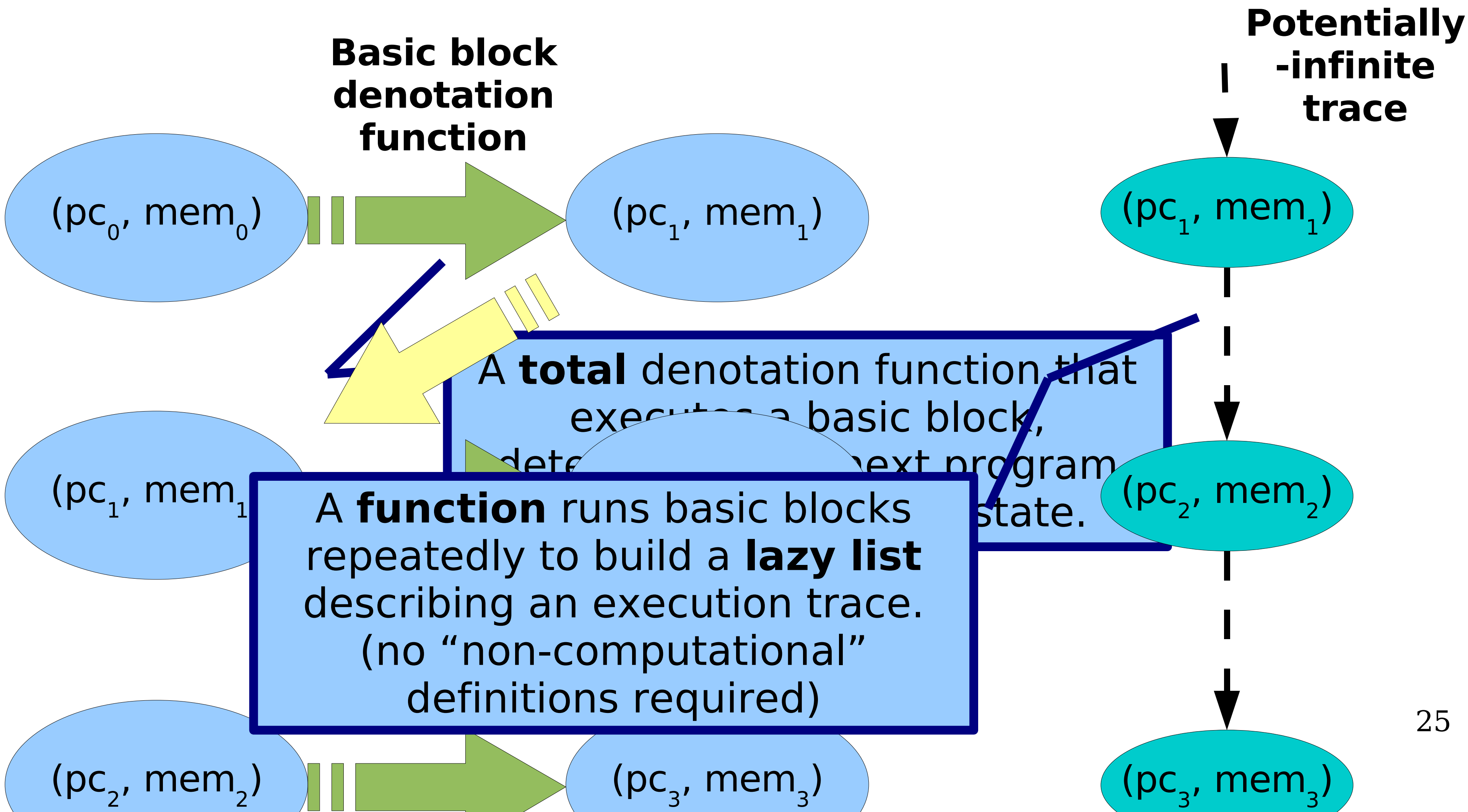
(In other words, a lot like Coq)

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)



Co-inductive Traces

$$T ::= n \mid \bot \mid \star, T$$

Take one more step of computation.

Run-time failure

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

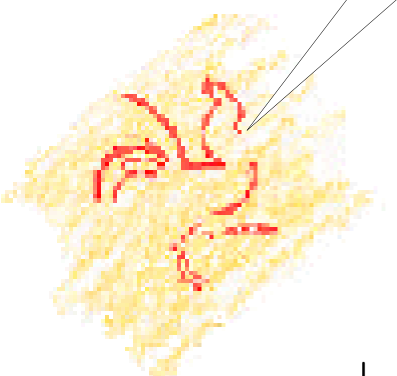
$(\text{let } y = o \text{ in } e$ $\quad u$ $\quad \text{let } y = o \text{ in } (e_1 \overset{u}{\bullet} e_2)$

$(\text{throw}$ $\quad \text{let } u = o \text{ in } e$

$(x\ y$ $\bullet\ e =$ $\quad \text{let } f = (\lambda v, \text{let } g$

$\quad \text{in } x\ y\ f$

Type error!



Translation works in some context Γ... but **used** in context Γ, *u* : $\tau_1 \rightarrow \tau_2$!

$$\lfloor n \rfloor = \text{throw } \langle n \rangle$$

$$\lfloor x \rfloor = \text{throw } \langle x \rangle$$

$$\lfloor e_1\ e_2 \rfloor = \lfloor e_1 \rfloor \overset{u}{\bullet} (\lfloor e_2 \rfloor \overset{v}{\bullet} (\text{let } f = (\lambda x, \text{throw } \langle x \rangle) \text{ in } u\ v\ f)))$$

$$\lfloor \lambda x : \tau, e \rfloor = \text{throw } \langle \lambda x : \tau, \lfloor e \rfloor \rangle$$

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

$$\text{weaken} : \textbf{forall}\ (G : \text{list ty})\ (t : \text{ty}),\ \text{term}\ G\ t$$

$$\textbf{->}\ \textbf{forall}\ (t' : \text{ty}),\ \text{term}\ (t' :: G)\ t$$

2. Lemmas about the function

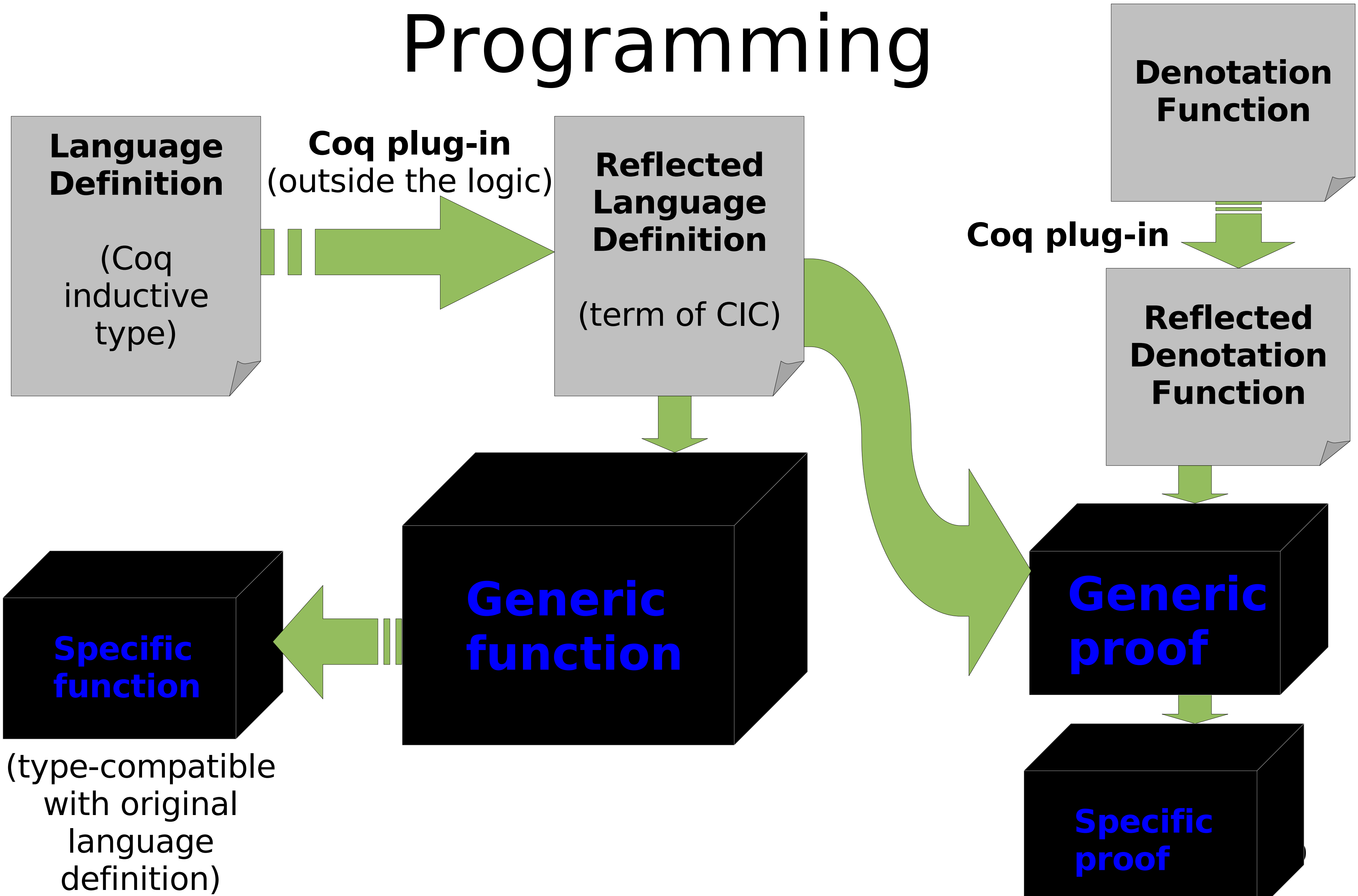
For any term *e*, properly-typed substitution σ, and properly-typed value *v*:

$$[\![weaken(e)]\!](\sigma, v) = [\![e]\!]\sigma$$

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

$$\begin{aligned} [\![x]\!]\sigma &= f_{var}(\sigma(x)) \\ [\![e_1\ e_2]\!]\sigma &= f_{app}([\![e_1]\!]\sigma, [\![e_2]\!]\sigma) \\ [\![\lambda x : \tau, e]\!]\sigma &= f_{lam}(\lambda x : [\![\tau]\!], [\![e]\!](\sigma, x)) \end{aligned}$$

Reflection-Based Generic Programming



What to Prove?

Overall correctness theorem: The compilation of a program of type **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:

$$
\begin{aligned}
n_1 \simeq_{\mathsf{N}} n_2 &= n_1 = n_2 \\
f_1 \simeq_{\tau_1 \to \tau_2} f_2 &= \forall x_1 : [\![\tau_1]\!]^S, \forall x_2 : [\![\tau_1]\!]^L, x_1 \simeq_{\tau_1} x_2 \\
&\quad \to \exists v : [\![\tau_2]\!]^L, \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, \\
&\quad\quad f_1\ x_1 \simeq_{\tau_2} v
\end{aligned}
$$

This function space contains many functions not representable in our object languages!

In the Trenches

$$
\begin{array}{ll}
H_1 : & \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \rightarrow \exists v : [\![\tau_1 \rightarrow \tau_2]\!]^L, \\
& \forall k : [\![\tau_1 \rightarrow \tau_2]\!]^L \rightarrow \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \rightarrow \tau_2} v \\
H_2 : & \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \rightarrow \exists v : [\![\tau_1]\!]^L, \\
& \forall k : [\![\tau_1]\!]^L \rightarrow \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v \\
\hline
& \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \rightarrow \exists v : [\![\tau_2]\!]^L, \\
& \forall k : [\![\tau_2]\!]^L \rightarrow \mathbb{N}, [\![\lfloor e_1\ e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1\ e_2]\!]^S \sigma^S \simeq_{\tau_2} v
\end{array}
$$

Easy first step: Use introduction rules for forall's and implications at the start of the goal.

In the Trenches

$$
\begin{aligned}
H_1 &: \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1 \to \tau_2]\!]^L, \\
&\quad \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v \\
H_2 &: \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq \\
&\quad \forall k : [\![\tau_1]\!]^L \to \mathbb{N} \\
\sigma^S &: \ldots \\
\sigma^L &: \ldots \\
H_3 &: \sigma^S \simeq_\Gamma \sigma^L \\
\hline
&\exists v : [\![\tau_2]\!]^L, \\
&\quad \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1\ e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1\ e_2]\!]^S \sigma^S \simeq_{\tau_2} v
\end{aligned}
$$

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

$$H_1 : \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1 \to \tau_2]\!]^L,$$
$$\forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v$$
$$H_2 : \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1]\!]^L,$$
$$\forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v$$
$$\sigma^S : \ldots$$
$$\sigma^L : \ldots$$
$$H_3 : \sigma^S \simeq_\Gamma \sigma^L$$

$$\exists v : [\![\tau_2]\!]^L,$$
$$\forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1\ e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1\ e_2]\!]^S \sigma^S \simeq_{\tau_2} v$$

In the Trenches

$$
\begin{array}{rl}
H_1 : & \exists v : [\![\tau_1 \to \tau_2]\!]^L, \\
& \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v \\
H_2 : & \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1]\!]^L, \\
& \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v \\
\sigma^S : & \ldots \\
\sigma^L : & \ldots \\
H_3 : & \sigma^S \simeq_\Gamma \sigma^L \\
\hline
& \exists v : [\![\tau_2]\!]^L, \\
& \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1\ e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1\ e_2]\!]^S \sigma^S \simeq_{\tau_2} v
\end{array}
$$

Existential hypotheses are easy to eliminate.

In the Trenches

$$
\begin{aligned}
v_1 &: [\![\tau_1 \to \tau_2]\!]^L \\
H_1 &: \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L\ k = k\ v_1 \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v_1 \\
H_2 &: \forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_\Gamma \sigma^L \to \exists v : [\![\tau_1]\!]^L, \\
&\quad\ \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v \\
\sigma^S &: \ldots \\
\sigma^L &: \ldots \\
H_3 &: \sigma^S \simeq_\Gamma \sigma^L
\end{aligned}
$$

$$
\begin{aligned}
&\exists v : [\![\tau_2]\!]^L, \\
&\forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1\ e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1\ e_2]\!]^S \sigma^S \simeq_{\tau_2} v
\end{aligned}
$$

We can't make further progress with this hypothesis, since no term of the type given for *k* exists in the proof state.

In the Trenches

$$
\begin{array}{rl}
v_1 & : [\![\tau_1 \to \tau_2]\!]^L \\
H_1 & : \forall k : [\![\tau_1 \to \tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \sigma^L\ k = k\ v_1 \wedge [\![e_1]\!]^S \sigma^S \simeq_{\tau_1 \to \tau_2} v_1 \\
v_2 & : [\![\tau_1]\!]^L \\
H_2 & : \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L\ k = k\ v_2 \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v_2 \\
\sigma^S & : \ldots \\
\sigma^L & : \ldots \\
H_3 & : \sigma^S \simeq_\Gamma \sigma^L \\
\hline
 & \exists v : [\![\tau_2]\!]^L, \\
 & \forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1\ e_2 \rfloor]\!]^L \sigma^L\ k = k\ v \wedge [\![e_1\ e_2]\!]^S \sigma^S \simeq_{\tau_2} v
\end{array}
$$

We can simplify the conclusion by applying rewrite rules (like those we generated automatically) until no more apply.

In the Trenches

$$
\begin{array}{rl}
v_1 & : \llbracket \tau_1 \to \tau_2 \rrbracket^L \\
H_1 & : \forall k : \llbracket \tau_1 \to \tau_2 \rrbracket^L \to \mathbb{N}, \llbracket \lfloor e_1 \rfloor \rrbracket^L \sigma^L \; k = k \; v_1 \wedge \llbracket e_1 \rrbracket^S \sigma^S \simeq_{\tau_1 \to \tau_2} v_1 \\
v_2 & : \llbracket \tau_1 \rrbracket^L \\
H_2 & : \forall k : \llbracket \tau_1 \rrbracket^L \to \mathbb{N}, \llbracket \lfloor e_2 \rfloor \rrbracket^L \sigma^L \; k = k \; v_2 \wedge \llbracket e_2 \rrbracket^S \sigma^S \simeq_{\tau_1} v_2 \\
\sigma^S & : \ldots \\
\sigma^L & : \ldots \\
H_3 & : \sigma^S \simeq_\Gamma \sigma^L \\
\hline
 & \exists v : \llbracket \tau_2 \rrbracket^L, \\
 & \forall k : \llbracket \tau_2 \rrbracket^L \to \mathbb{N}, \llbracket \lfloor e_1 \rfloor \rrbracket^L \; \sigma^L (\lambda x : \llbracket \tau_1 \to \tau_2 \rrbracket^L, \ldots) = \ldots \wedge \ldots
\end{array}
$$

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

In the Trenches

$$
\begin{aligned}
v_1 &: [\![\tau_1 \to \tau_2]\!]^L \\
H_1 &: [\![\lfloor e_1 \rfloor]\!]^L \sigma^L \, (\lambda x : [\![\tau_1 \to \tau_2]\!]^L, \ldots) = (\lambda x : [\![\tau_1 \to \tau_2]\!]^L, \ldots) \; v_1 \wedge \ldots \\
v_2 &: [\![\tau_1]\!]^L \\
H_2 &: \forall k : [\![\tau_1]\!]^L \to \mathbb{N}, [\![\lfloor e_2 \rfloor]\!]^L \sigma^L \; k = k \; v_2 \wedge [\![e_2]\!]^S \sigma^S \simeq_{\tau_1} v_2 \\
\sigma^S &: \ldots \\
\sigma^L &: \ldots \\
H_3 &: \sigma^S \simeq_\Gamma \sigma^L
\end{aligned}
$$

$$
\begin{aligned}
&\exists v : [\![\tau_2]\!]^L, \\
&\forall k : [\![\tau_2]\!]^L \to \mathbb{N}, [\![\lfloor e_1 \rfloor]\!]^L \, \sigma^L (\lambda x : [\![\tau_1 \to \tau_2]\!]^L, \ldots) \Rightarrow \ldots \wedge \ldots
\end{aligned}
$$

We can use H_1 to rewrite the goal.

In the Trenches

$$
\begin{array}{rl}
v_1 & : \llbracket \tau_1 \to \tau_2 \rrbracket^L \\
H_1 & : \llbracket \lfloor e_1 \rfloor \rrbracket^L \sigma^L \; (\lambda x : \llbracket \tau_1 \to \tau_2 \rrbracket^L, ...) = (\lambda x : \llbracket \tau_1 \to \tau_2 \rrbracket^L, ...) \; v_1 \wedge \ldots \\
v_2 & : \llbracket \tau_1 \rrbracket^L \\
H_2 & : \forall k : \llbracket \tau_1 \rrbracket^L \to \mathbb{N}, \llbracket \lfloor e_2 \rfloor \rrbracket^L \sigma^L \; k = k \; v_2 \wedge \llbracket e_2 \rrbracket^S \sigma^S \simeq_{\tau_1} v_2 \\
\sigma^S & : \ldots \\
\sigma^L & : \ldots \\
H_3 & : \sigma^S \simeq_\Gamma \sigma^L \\
\hline
& \exists v : \llbracket \tau_2 \rrbracket^L, \\
& \forall k : \llbracket \tau_2 \rrbracket^L \to \mathbb{N}, (\lambda x : \llbracket \tau_1 \to \tau_2 \rrbracket^L, ...) \; v_1 = \ldots \wedge \ldots
\end{array}
$$

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.