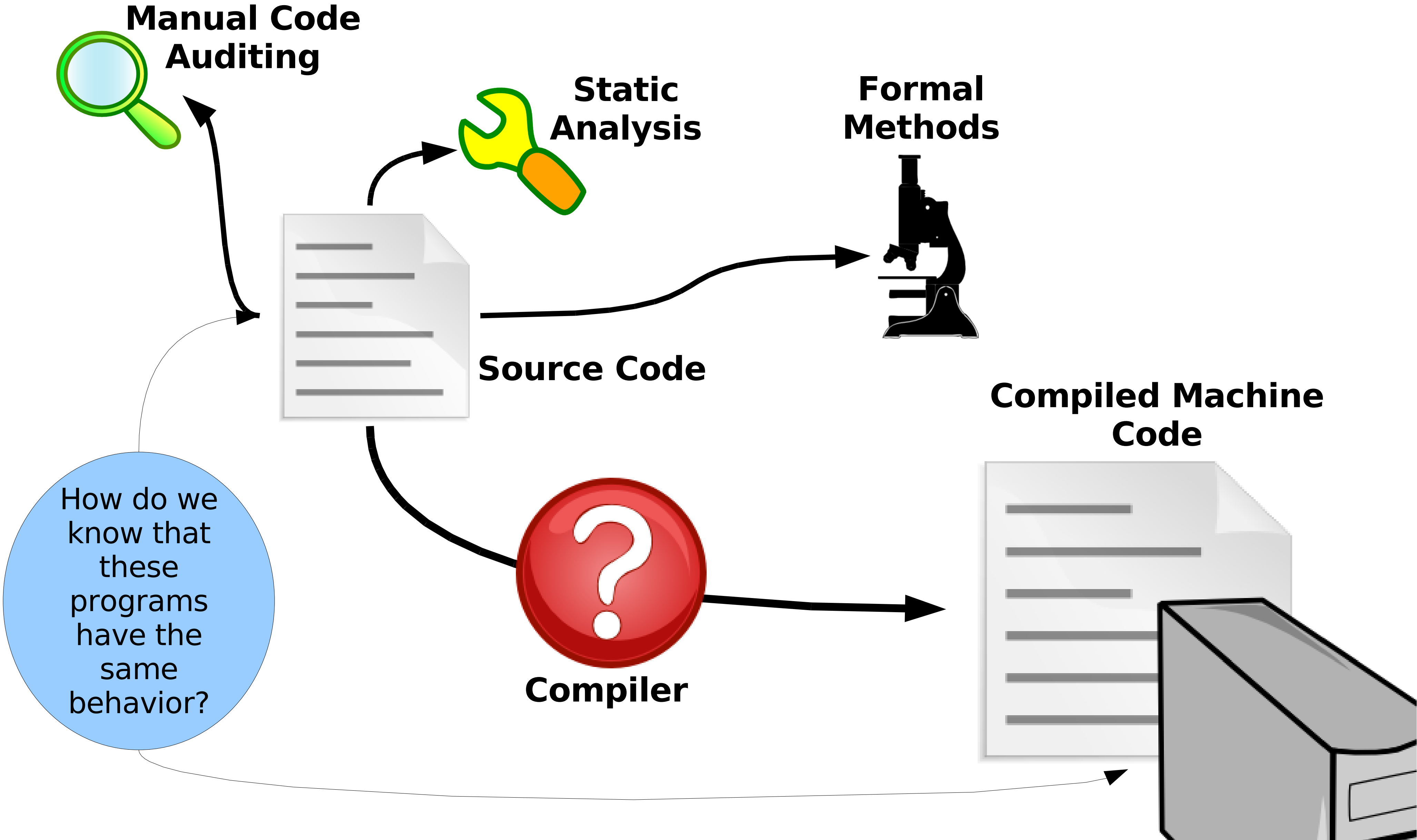
A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language

Adam Chlipala
University of California, Berkeley

PLDI 2007

Why Certified Compilers?



Manual Code Auditing

Static Analysis

Formal Methods

Source Code

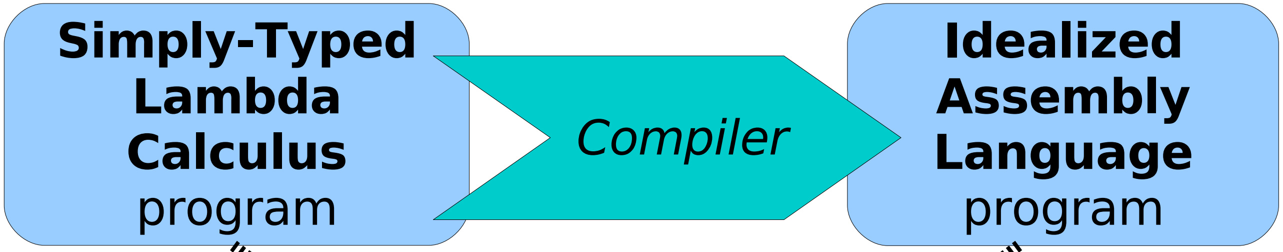
Compiled Machine Code

Compiler

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

End Product

```
$ 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], [])
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
```



Simply-Typed Lambda Calculus program

Compiler

Idealized Assembly Language program

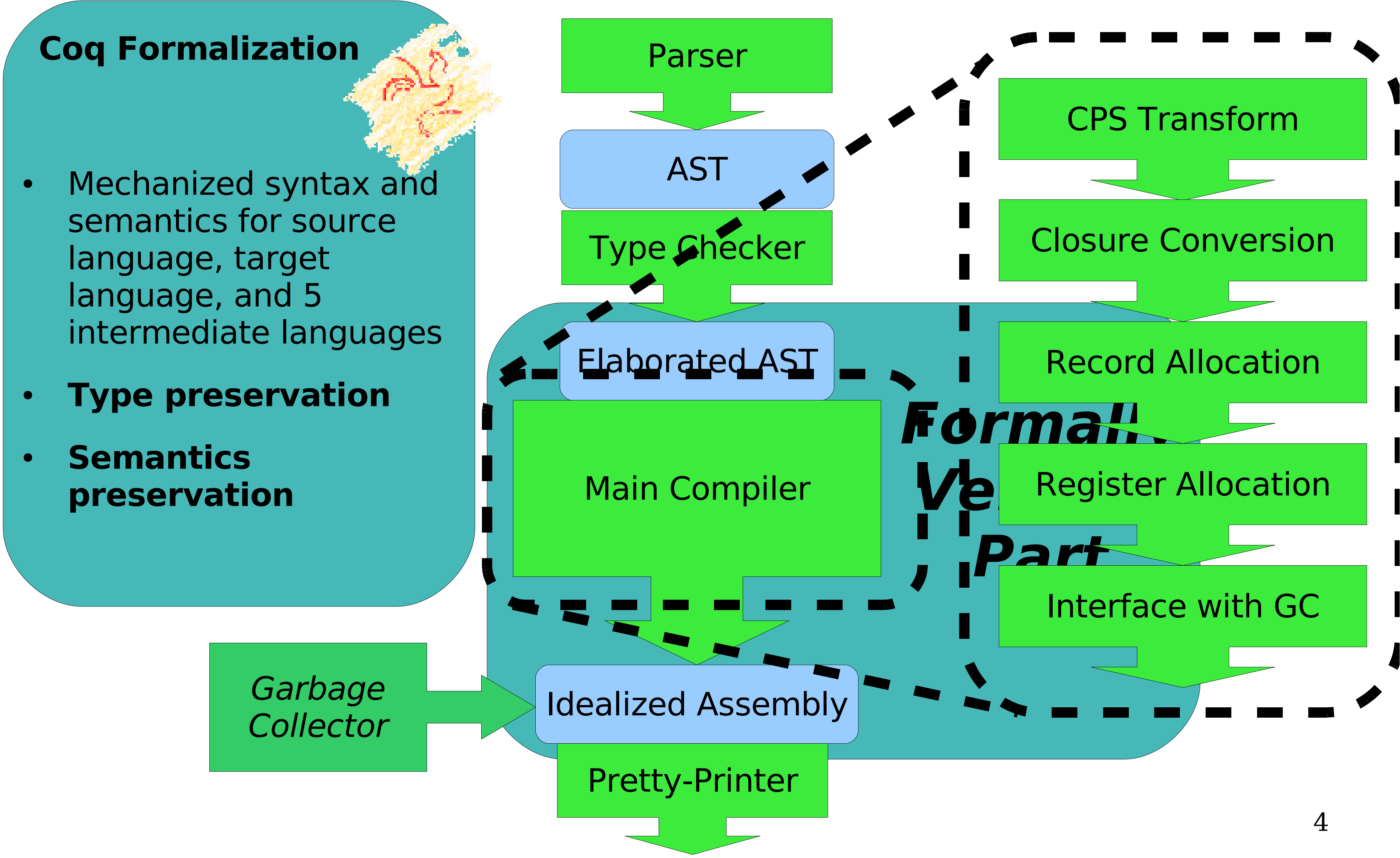
with a

Contribution #1:
Mechanizing proofs about type-preserving compilation

The Big Theorem:
(proved mechanically)

This commutative diagram holds for ***EVERY*** input program.

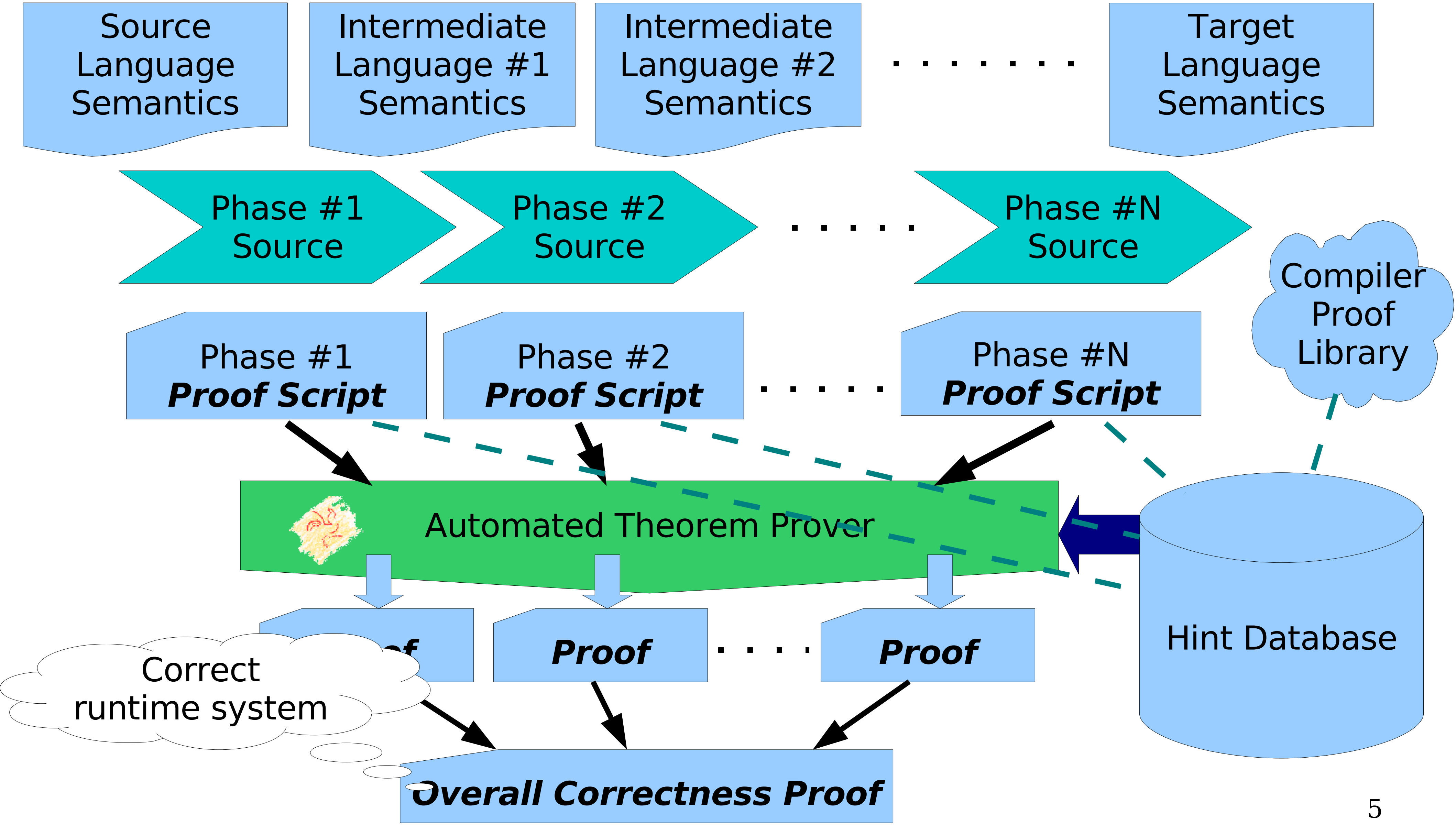
Architecture



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

Compiler Proof Library

Phase #1 ***Proof Script***

Phase #2 ***Proof Script***

. . . . . .

Phase #N ***Proof Script***

Automated Theorem Prover

Hint Database

of

Proof

. . . . .

Proof

Correct runtime system

Overall Correctness Proof

An Example

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

type exp =
     Const of int
   | Var of var
   | Lambda of var * exp
   | Apply of exp * exp
```

```
type lty =
     LInt
   | LArrow of lty * lty

type lexp =
     LConst of int
   | LVar of var
   | LLambda of var * lexp
   | LApply of exp * lexp
   | LLet of var * lexp * lexp
```

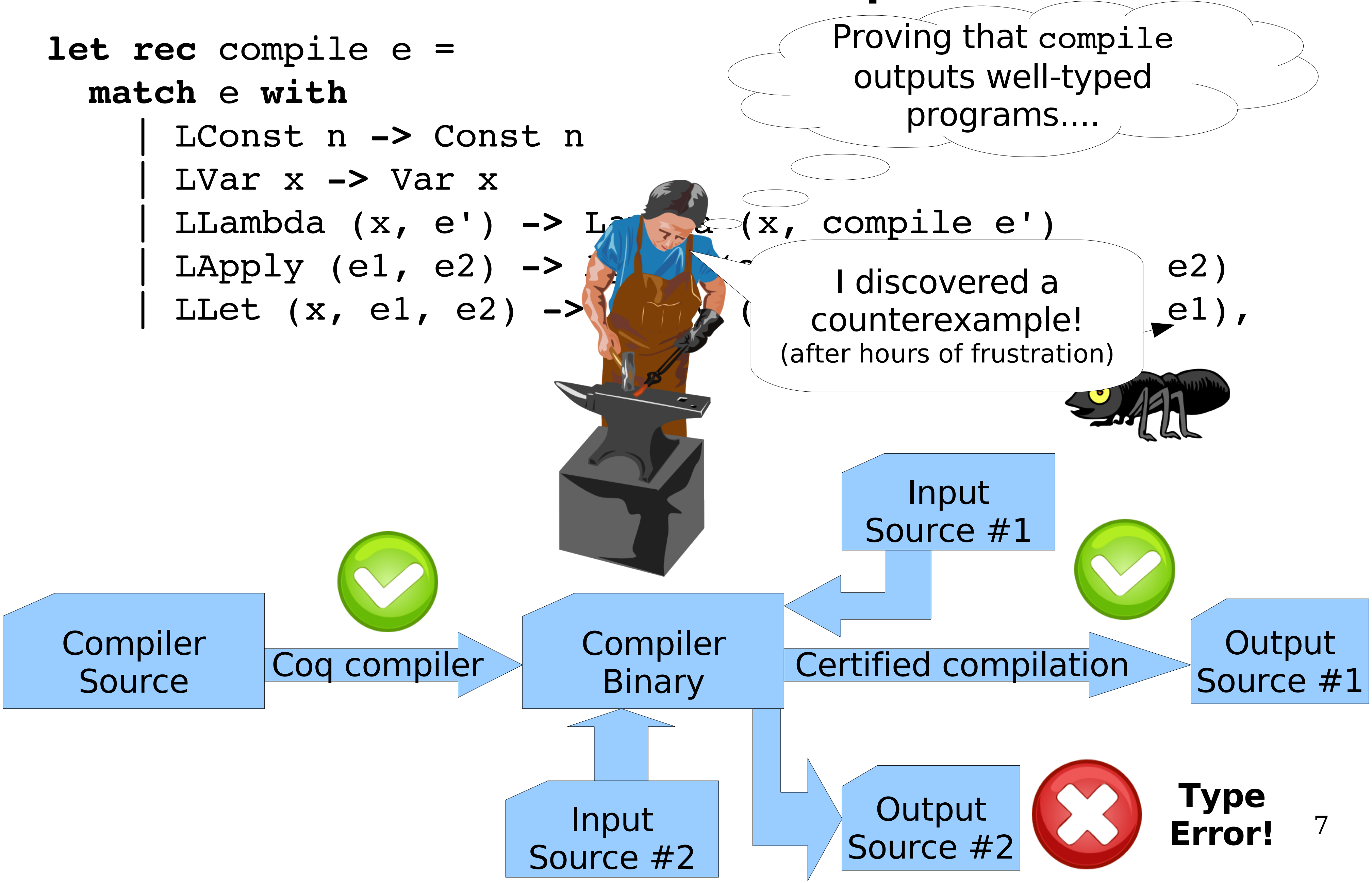
Objective:

Compile `lexps` into `exps` using this identity:

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \quad \simeq \quad (\lambda x.\ e_2)\ e_1$$

First Attempt

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



Compiler Source

Coq compiler

Compiler Binary

Input Source #1

Certified compilation

Output Source #1

Input Source #2

Output Source #2

Type Error!

Stating Our As

```
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
```

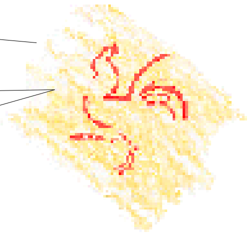
An expression *e* of type `(`Γ`,` τ`) exp` is isomorphic to a derivation of $\Gamma \vdash e : \tau$

Idea:

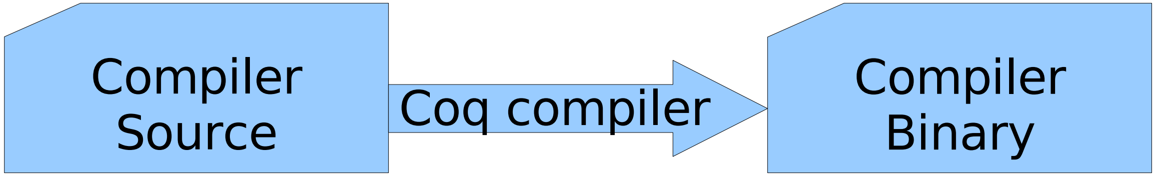
Represent expressions as their (strongly-typed) **typing derivations**

Second Attempt

```
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!



Compiler Source

Coq compiler

Compiler Binary

Type Error!

Dynamic Semantics

“Let” Language

```
LLet (x,
      L
```

Base Language

```
LApp ((LLambda (x,
               Var x),
```

D

Contribution #2:
Denotational semantics for proofs in type theory

(drawing on ideas related to GADTs and tagless interpreters)

let *x* = 1 **in** *x**

Equivalent!

(λ*x*. *x*) 1

Provable by the laws of the core language!

Common Core Language

Inside a Proof

Correctn

Prove by *ind*

Inductive ste

[`compile`

IH1: [`comp`
IH2: [`comp`

Contribution #3:

Aggressive automation of translation correctness proofs

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

$$[\texttt{compile}\ (\text{LLet}\ (x, e_1, e_2))] \simeq [\text{App}\ (\text{Lambda}\ (x, \texttt{compile}\ e_2), \texttt{compile}\ e_1)]$$

$$\simeq (\lambda x.\ [\texttt{compile}\ e_2])\ [\texttt{compile}\ e_1]$$

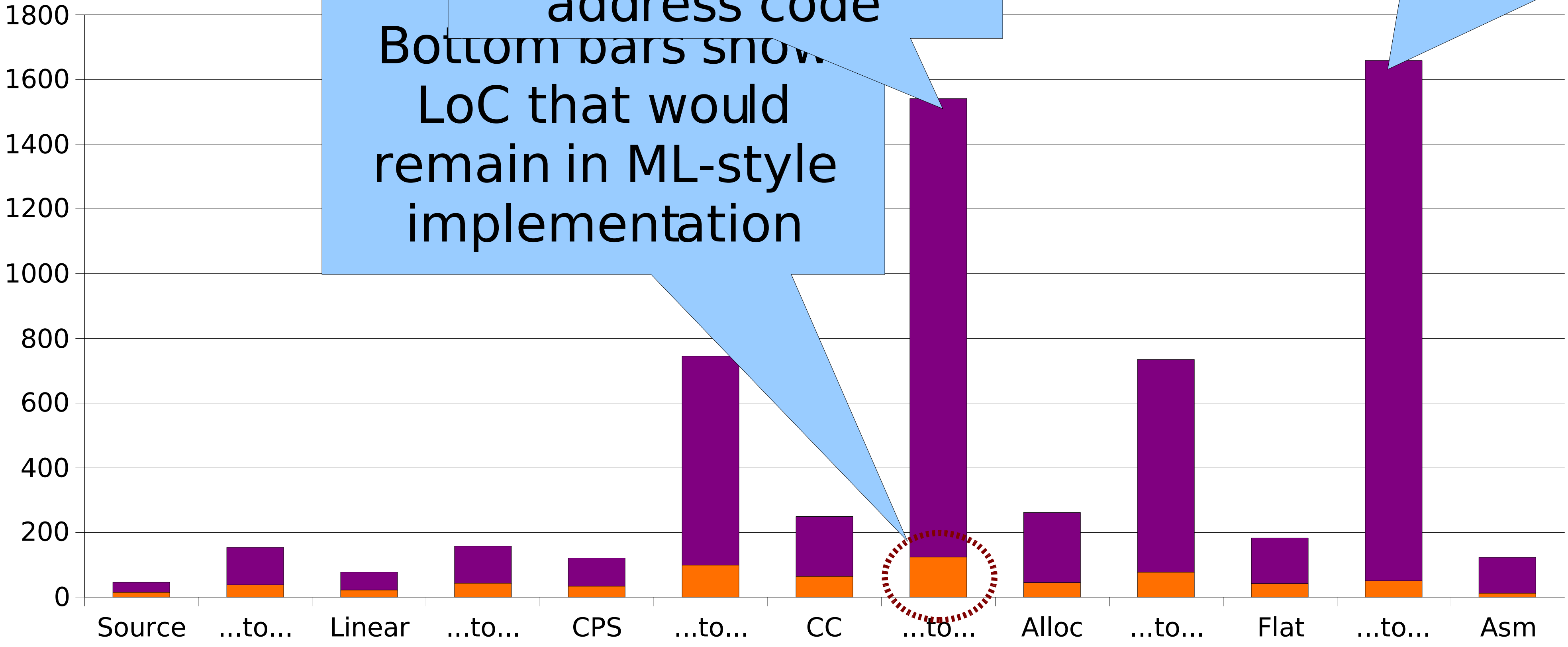
$$\simeq [\texttt{compile}\ e_2]\ \{x \Rightarrow [\texttt{compile}\ e_1]\}$$

$$\simeq [e_2]\ \{x \Rightarrow [e_1]\}$$

$$\simeq [\text{let}\ x = e_1\ \text{in}\ e_2]$$

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

Implementation Statistics



Shift from lambda calculus to "three-address code"

First reasoning about garbage collector interaction

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

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/`

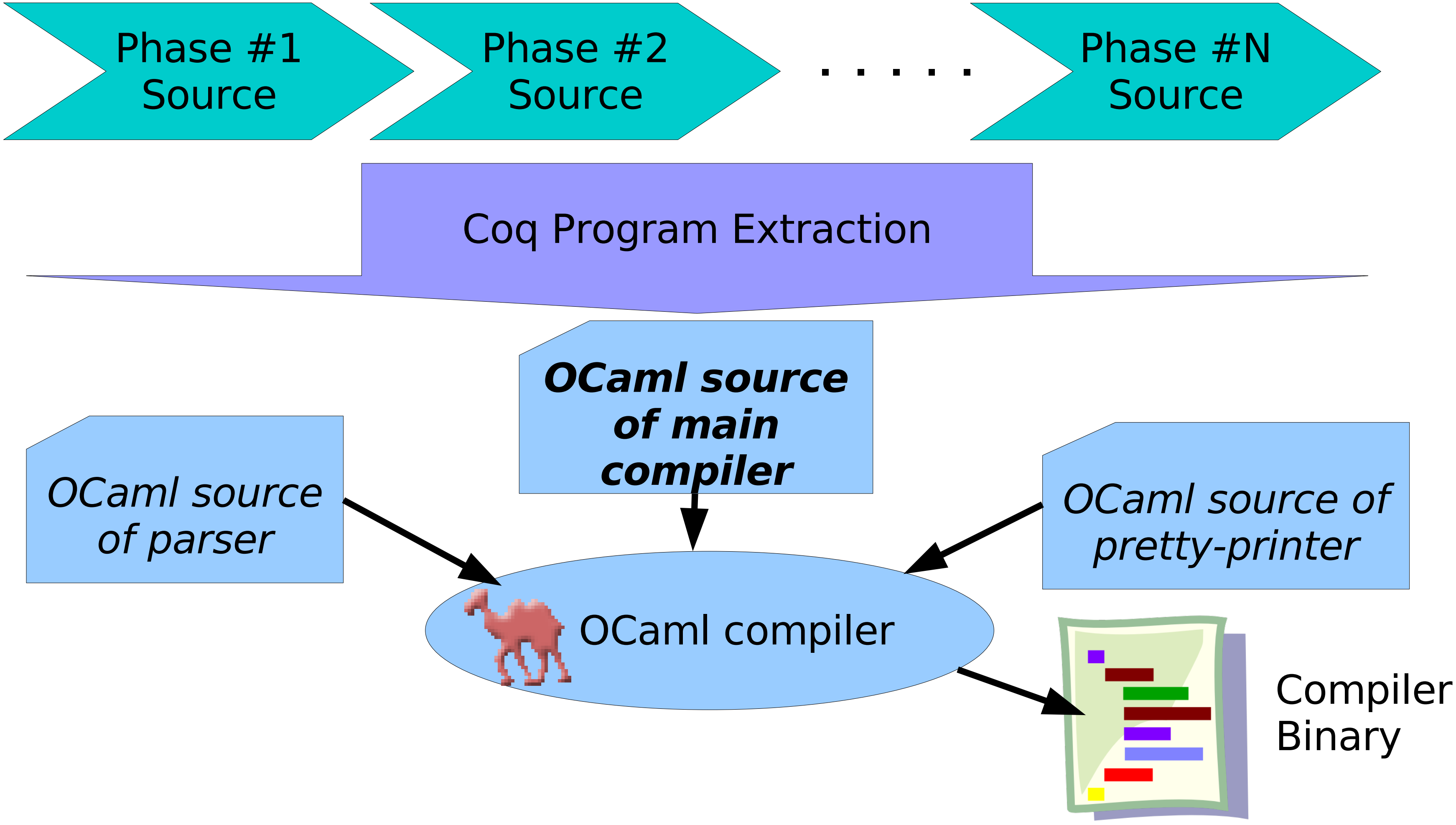
Key Innovations of This Work

- 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*

```
Require Import List.

Require Import LambdaTamer.LambdaTamer.
Require Import LambdaTamer.Tactics.

Set Implicit Arguments.

Inductive sty : Set :=
  | SBool : sty
  | SArrow : sty -> sty -> sty
  | SProd : sty -> sty -> sty.

Inductive sterm : list sty -> sty -> Set :=
  | SVar : forall G t,
    Var G t
    -> sterm G t
  | SConst : forall G,
    bool
    -> sterm G SBool
  | SLam : forall G dom ran,
    sterm (dom :: G) ran
    -> sterm G (SArrow dom ran)
  | SApp : forall G dom ran,
    sterm G (SArrow dom ran)
    -> sterm G dom
    -> sterm G ran
  | SPair : forall G t1 t2,
    sterm G t1
    -> sterm G t2
    -> sterm G (SProd t1 t2)
  | SFst : forall G t1 t2,
    sterm G (SProd t1 t2)
    -> sterm G t1
  | SSnd : forall G t1 t2,
    sterm G (SProd t1 t2)
    -> sterm G t2.

Fixpoint styDenote (t : sty) : Set :=
  match t with
    | SBool => bool
    | SArrow t1 t2 => styDenote t1 -> styDenote t2
    | SProd t1 t2 => (styDenote t1 * styDenote t2)%type
  end.

Fixpoint stermDenote (G : list sty) (t : sty) (e : sterm G t) {struct e}
  : Subst styDenote G -> styDenote t :=
    match e in (sterm G t) return (Subst styDenote G -> styDenote t) with
      | SConst _ b => fun _ => b
      | SVar _ _ v => VarDenote v
      | SLam _ _ _ e' => fun s x => stermDenote e' (SCons _ x s)
      | SApp _ _ _ e1 e2 => fun s => (stermDenote e1 s) (stermDenote e2 s)
      | SPair _ _ _ e1 e2 => fun s => (stermDenote e1 s, stermDenote e2 s)
      | SFst _ _ _ e' => fun s => fst (stermDenote e' s)
      | SSnd _ _ _ e' => fun s => snd (stermDenote e' s)
    end.

Inductive cty : Set :=
  | CBool : cty
  | CCont : cty -> cty
  | CProd : cty -> cty -> cty.

Inductive cprimexp : list cty -> cty -> Set :=
  | CVar : forall G t,
    Var G t
    -> cprimexp G t
  | CConst : forall G,
    bool
    -> cprimexp G CBool
  | CLam : forall G t,
    cexp (t :: G) tt
    -> cprimexp G (CCont t)
  | CPair : forall G t1 t2,
    Var G t1
    -> Var G t2
    -> cprimexp G (CProd t1 t2)
  | CFst : forall G t1 t2,
    Var G (CProd t1 t2)
    -> cprimexp G t1
  | CSnd : forall G t1 t2,
    Var G (CProd t1 t2)
    -> cprimexp G t2

    with cexp : list cty -> unit -> Set :=
  | CLet : forall G t,
    cprimexp G t
    -> cexp (t :: G) tt
    -> cexp G tt
  | CCall : forall G t,
    Var G (CCont t)
    -> Var G t
    -> cexp G tt.

Syntactify cprimexp.

Fixpoint ctyDenote (t : cty) : Set :=
  match t with
    | CBool => bool
    | CCont t1 => ctyDenote t1 -> bool
    | CProd t1 t2 => (ctyDenote t1 * ctyDenote t2)%type
  end.

Definition contTy (_ : unit) := bool.

Definition withCont_bind (A B : Set) (v : A) (f : A -> B) : B :=
  f v.

Fixpoint cprimexpDenote (G : list cty) (t : cty) (e : cprimexp G t) {struct e}
  : Subst ctyDenote G -> ctyDenote t :=
    match e in (cprimexp G t) return (Subst ctyDenote G -> ctyDenote t) with
      | CConst _ b => fun _ => b
      | CVar _ _ v => VarDenote v
      | CLam _ _ e' => fun s x => cexpDenote e' (SCons _ x s)
      | CPair _ _ _ e1 e2 => fun s => (VarDenote e1 s, VarDenote e2 s)
      | CFst _ _ _ e' => fun s => fst (VarDenote e' s)
      | CSnd _ _ _ e' => fun s => snd (VarDenote e' s)
    end
  with cexpDenote (G : list cty) (t : unit) (e : cexp G t) {struct e}
  : Subst ctyDenote G -> contTy t :=
  match e in (cexp G t) return (Subst ctyDenote G -> contTy t) with
    | CLet _ _ e1 e2 => fun s =>
                             withCont_bind
                             (cprimexpDenote e1 s)
                             (fun x => cexpDenote e2 (SCons _ x s))
    | CCall _ _ f x => fun s => (VarDenote f s) (VarDenote x s)
  end.

Syntactify cprimexpDenote.

Fixpoint cpsTy (t : sty) : cty :=
  match t with
    | SBool => CBool
    | SArrow t1 t2 => CCont (CProd (cpsTy t1) (CCont (cpsTy t2)))
    | SProd t1 t2 => CProd (cpsTy t1) (cpsTy t2)
  end.

Definition insert t G (e : cexp (t :: G) tt) t' : cexp (t :: t' :: G) tt :=
  Cexp.lift' (t :: nil) G e t'.

Definition insert2 t1 t2 G (e : cexp (t1 :: t2 :: G) tt) t' : cexp (t1 :: t2 :: t' :: G) tt :=
  Cexp.lift' (t1 :: t2 :: nil) G e t'.

Fixpoint cpsTerm (G : list sty) (t : sty) (e : sterm G t) {struct e}
  : cexp (CCont (cpsTy t) :: map cpsTy G) tt :=
  match e in (sterm G t) return (cexp (CCont (cpsTy t) :: map cpsTy G) tt) with
    | SVar _ _ v =>
      CCall (First _ _) (liftVar (VarConvert _ v) _)
    | SConst _ b =>
      CLet
      (CConst _ b)
      (CCall (Next _ (First _ _)) (First _ _))
    | SApp _ _ _ e1 e2 =>
      CLet (CLam
                               (CLet
                                 (CLam
                                   (CLet
                                     (CPair (First _ _) (Next _ (Next _ (First _ _))))
                                     (CCall (Next _ (Next _ (First _ _))) (First _ _))))
                                 (insert (insert (cpsTerm e2) _) _)))
      (insert (cpsTerm e1) _)
    | SLam _ _ _ e' =>
      CLet
      (CLam
                               (CLet
                                 (CFst (First _ _))
                                 (CLet
                                   (CSnd (Next _ (First _ _)))
                                   (insert2 (insert2 (cpsTerm e') _) _))))
      (CCall (Next _ (First _ _)) (First _ _))
    | SPair _ _ _ e1 e2 =>
      CLet (CLam
                               (CLet
                                 (CLam
                                   (CLet
                                     (CPair (Next _ (First _ _)) (First _ _))
                                     (CCall (Next _ (Next _ (Next _ (First _ _)))) (First _ _))))
                                 (insert (insert (cpsTerm e2) _) _)))
      (insert (cpsTerm e1) _)
    | SFst _ _ _ e' =>
      CLet (CLam
                               (CLet
                                 (CFst (First _ _))
                                 (CCall (Next _ (Next _ (First _ _))) (First _ _))))
      (insert (cpsTerm e') _)
    | SSnd _ _ _ e' =>
      CLet (CLam
                               (CLet
                                 (CSnd (First _ _))
                                 (CCall (Next _ (Next _ (First _ _))) (First _ _))))
      (insert (cpsTerm e') _)
  end.

Fixpoint val_lr (t : sty) : styDenote t -> ctyDenote (cpsTy t) -> Prop :=
  match t return (styDenote t -> ctyDenote (cpsTy t) -> Prop) with
    | SBool => fun b1 b2 => b1 = b2
    | SArrow dom ran => fun f1 f2 =>
      forall x1 x2, val_lr dom x1 x2
                                -> exists res, (forall k, f2 (x2, k) = k res)
                                  /\ val_lr ran (f1 x1) res
    | SProd t1 t2 => fun p1 p2 =>
      val_lr t1 (fst p1) (fst p2)
      /\ val_lr t2 (snd p1) (snd p2)
  end.

Definition Subst_lr := Subst_lr cpsTy val_lr.
Hint Unfold Subst_lr.

Definition term_lr (G : list sty) (t : sty) (e1 : sterm G t)
  (e2 : cexp (CCont (cpsTy t) :: map cpsTy G) tt) : Prop :=
  forall s1 s2, Subst_lr s1 s2
    -> exists res, (forall k, cexpDenote e2 (SCons _ k s2) = k res)
      /\ val_lr t (stermDenote e1 s1) res.

Lemma insert_sound : forall G t t' (e : cexp (t :: G) tt)
  (s : Subst ctyDenote G)
  (x : ctyDenote t') (x0 : ctyDenote t),
  cexpDenote (insert e t') (SCons t x0 (SCons t' x s)) =
  cexpDenote e (SCons t x0 s).
  intros.
  replace (SCons t x0 (SCons t' x s))
    with (liftSubst' (t :: nil) G (SCons _ x0 s) _ x); trivial.
  unfold insert.
  generalize (CexpDenote.lift'_sound (t :: nil) G e t'
    (SCons _ x0 s) x); trivial.
Qed.

Lemma insert2_sound : forall G t1 t2 t' (e : cexp (t1 :: t2 :: G) tt)
  (s : Subst ctyDenote G)
  (x : ctyDenote t') (x0 : ctyDenote t1) (x1 : ctyDenote t2),
  cexpDenote (insert2 e t') (SCons t1 x0 (SCons t2 x1 (SCons t' x s))) =
  cexpDenote e (SCons t1 x0 (SCons t2 x1 s)).
  intros.
  replace (SCons t1 x0 (SCons t2 x1 (SCons t' x s)))
    with (liftSubst' (t1 :: t2 :: nil) G (SCons _ x0 (SCons _ x1 s)) _ x); trivial.
  unfold insert2.
  generalize (CexpDenote.lift'_sound (t1 :: t2 :: nil) G e t'
    (SCons _ x0 (SCons _ x1 s)) x); trivial.
Qed.

Hint Rewrite insert_sound insert2_sound : CPS.

Ltac my_simpl := unfold term_lr, withCont_bind in *;
  fold styDenote in *; fold ctyDenote in *; fold cpsTy in *; idtac;
    autorewrite with CPS;
      try match goal with
                                  | [ |- ?F ?X1 = ?F ?X2 ] =>
                                    replace X2 with X1; trivial
                                end;
      try (apply ext_eq; intro).

Ltac my_inster T k :=
  match T with
    | (?T1 * ?T2)%type =>
      match goal with
                                | [ H1 : T1, H2 : T2 |- _ ] => k (H1, H2)
      end
    | ctyDenote _ * (ctyDenote _ -> bool) -> bool =>
      match goal with
                                | [ e : sterm _ _, s2 : Subst _ _ |- _ ] =>
                                  k (fun p => cexpDenote (cpsTerm e) (SCons _ (snd p) (SCons _ (fst p) s2)))
      end
    | _ => stricter_inster T k
  end.

Ltac my_lr_tac := lr_tacN 3 ltac:(var_adder val_lr) my_inster my_simpl.

Theorem cpsTerm_sound : forall G t (e : sterm G t),
  term_lr e (cpsTerm e).
  induction e; my_lr_tac.
Qed.

Hint Immediate Slr_Nil.

Theorem cpsTerm_sound_bool : forall (e : sterm nil SBool),
  stermDenote e (SNil _)
  = cexpDenote (cpsTerm e) (SCons (denote := ctyDenote) (CCont CBool) (fun x => x) (SNil _)).
  intros;
    generalize (cpsTerm_sound e);
      my_lr_tac.
Qed.

Recursive Extraction cpsTerm.
```

“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

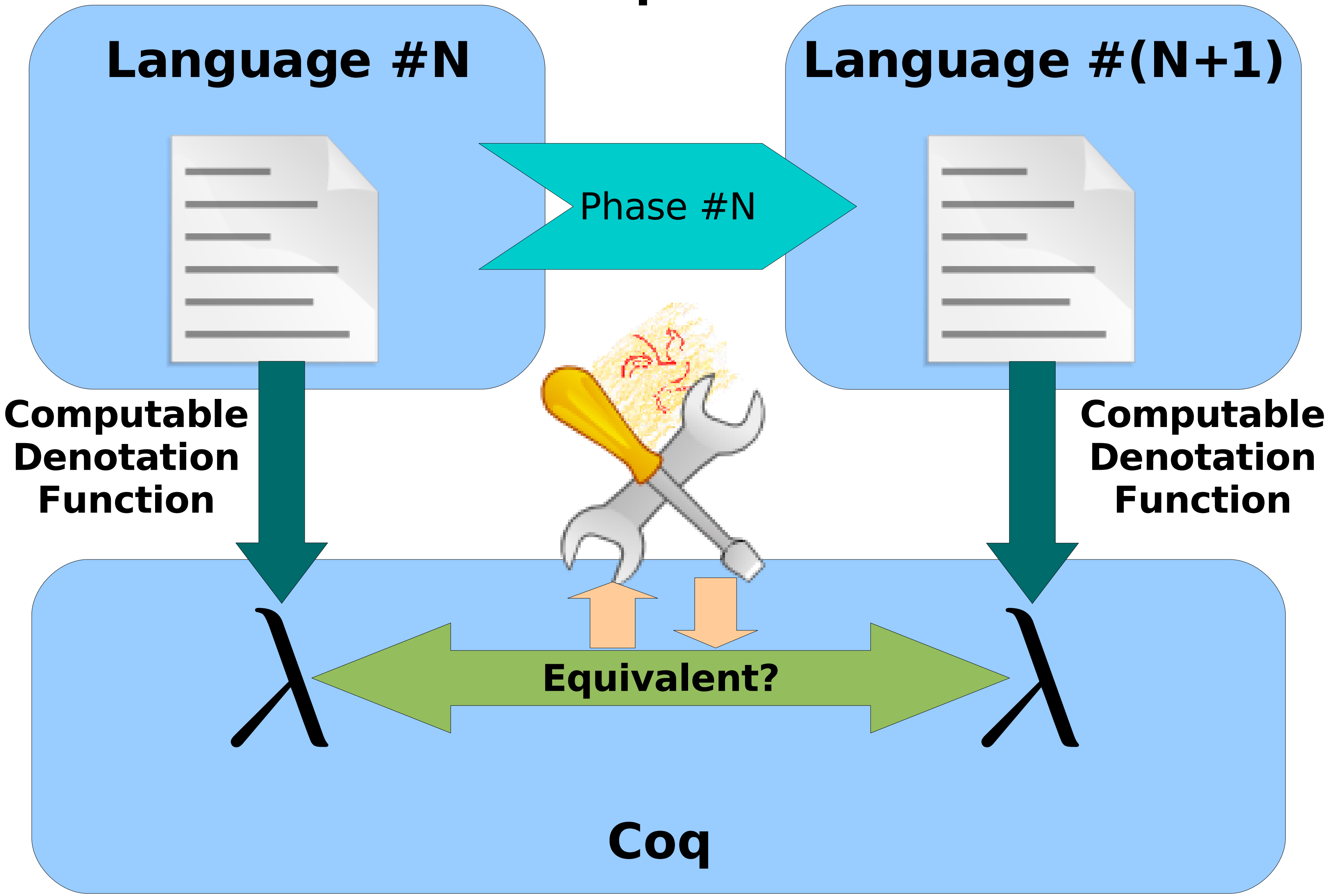
OCaml compiler

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

Computable Denotation Function

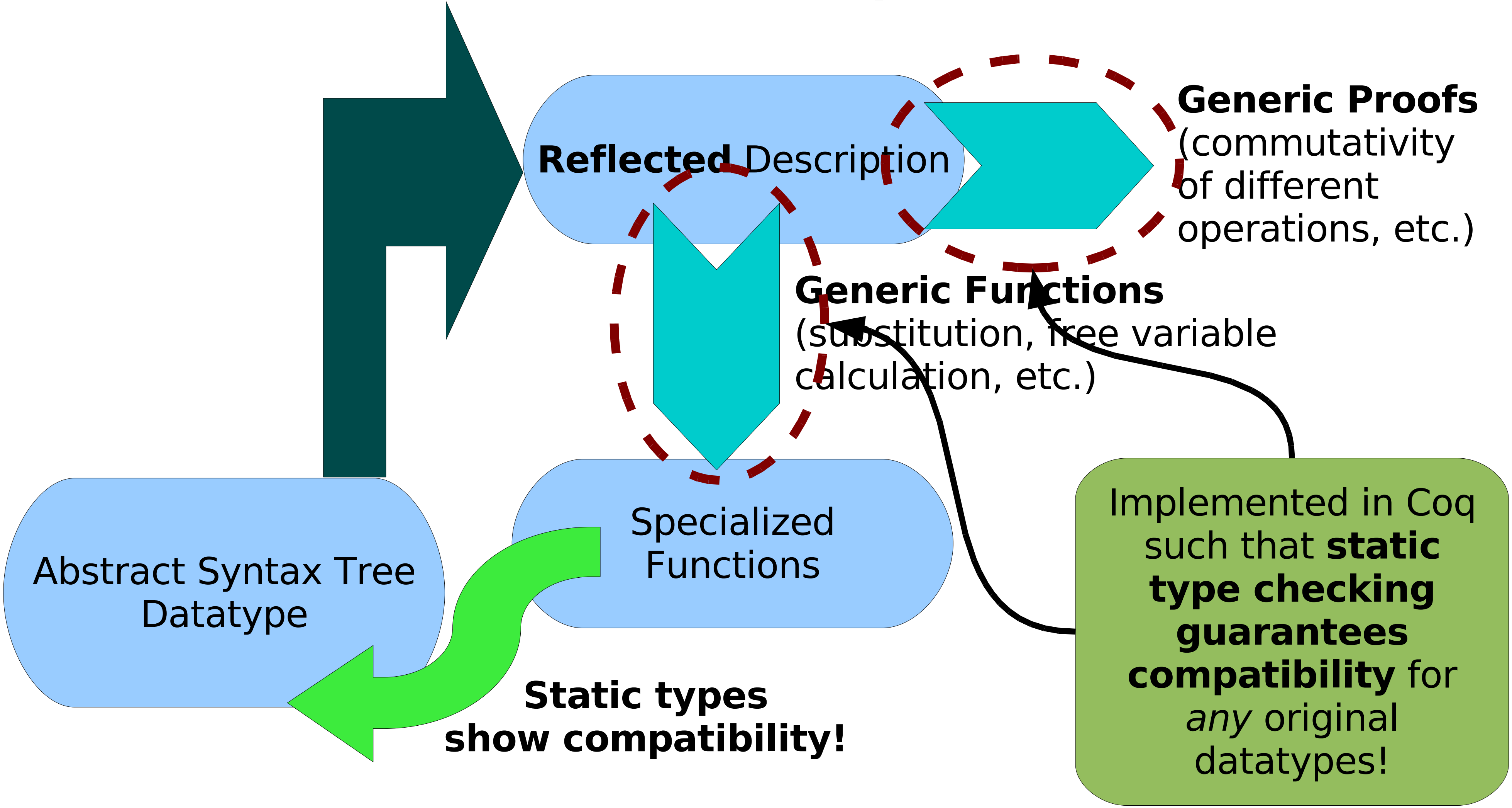
λ

Equivalent?

λ

Coq

Generic Programming of Variable Manipulation



Reflected Description

Generic Proofs
(commutativity of different operations, etc.)

Generic Functions
(substitution, free variable calculation, etc.)

Abstract Syntax Tree Datatype

Specialized Functions

Static types show compatibility!

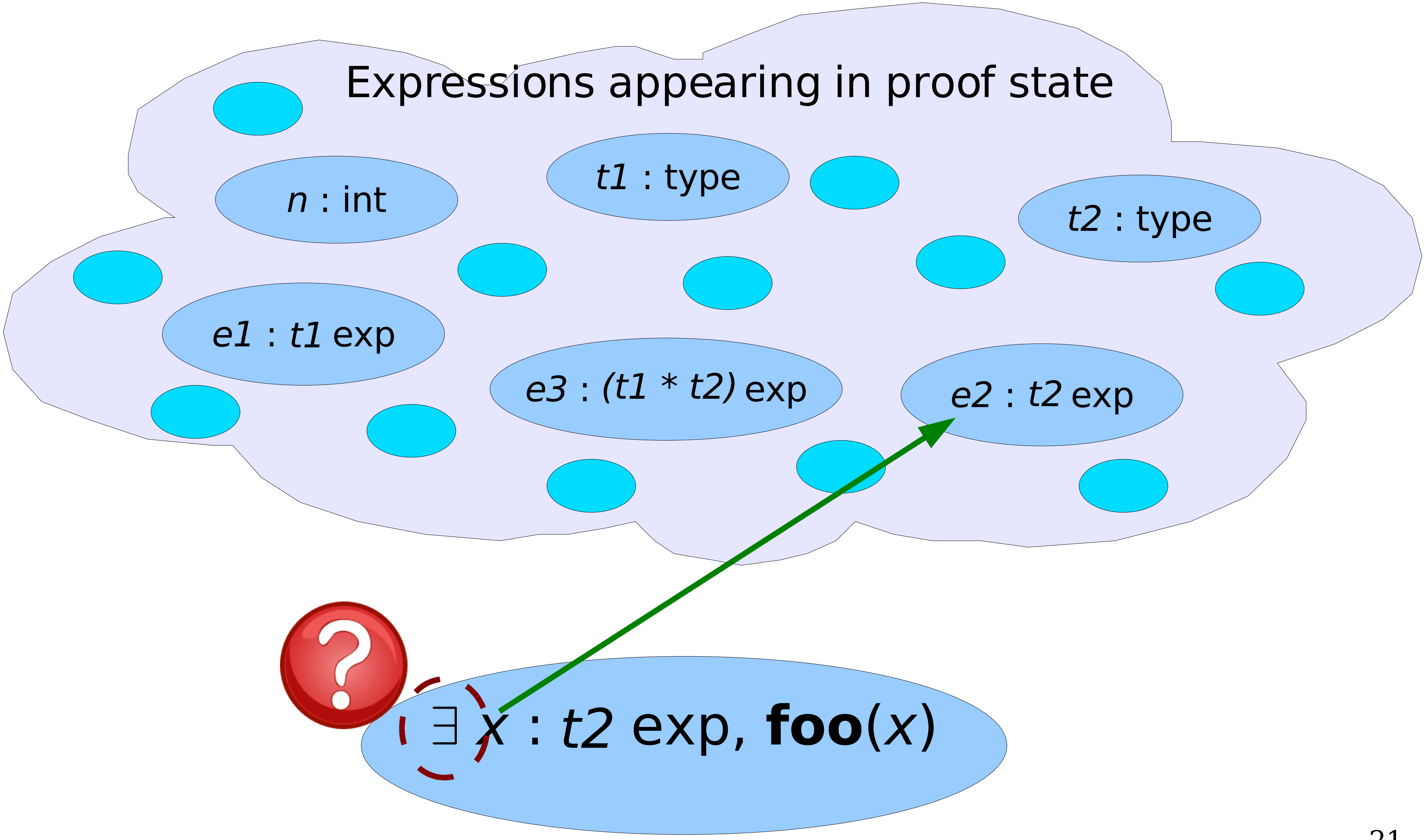
Implemented in Coq such that **static type checking guarantees compatibility** for *any* original datatypes!

Code/Proof Size Summary

PL Formalization Library:
3520 lines of Coq
2716 lines of OCaml

Component	**LoC**
Source	31
...to...	116
Linear	56
...to...	115
CPS	87
...to...	646
CC	185
...to...	1321
Alloc	217
...to...	658
Flat	141
...to...	868
Asm	111
Dictionaries	119
Traces	96
GC Safety	741
Glue code	119

Greedy Quantifier Instantiation



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

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

Partial Evaluation

Step 5: Apply IH2

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

Step 7: Apply IH1

Rich types make this relatively easy to automate!

Logic Programming

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)