Engineering a Verified Functional Language Compiler

Adam Chlipala
Harvard University
WMM 2009

The POPLmark ADT

- Induction
- Inversion
- Substitution

Compiling a Functional Language

Conversion to Continuation-Passing Style

f, x

```
λf. λx. f x
```

```
λp. let f = fst(p) in
    let k = snd(p) in
    let F = λp'. let x = fst(p') in
                 let k' = snd(p') in
                 let p'' = (x, k') in
                 f(p'') in
    k(F)
```

p, f, k, p', x, k'

Common Subexpression Elimination

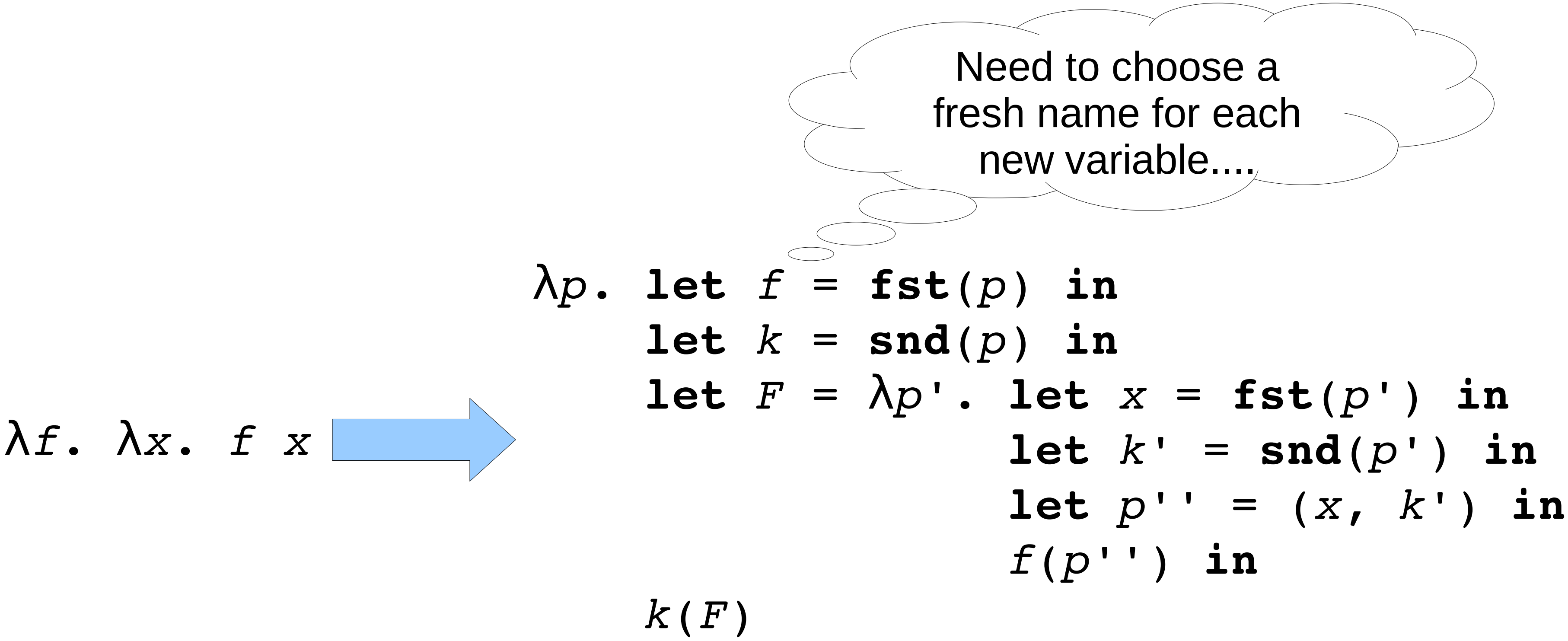
```
let x = fst(a) in
let y = snd(a) in
let z = (y, x) in
let u = fst(z) in
let p = (u, k') in
k(p)
```

a, k, k', x,
y, z, u, p

```

let y = snd(a) in



let p = (y, k') in
k(p)
```

a, k, k', y, p

Concrete Binding



Need to choose a fresh name for each new variable....

```
λf. λx. f x
```

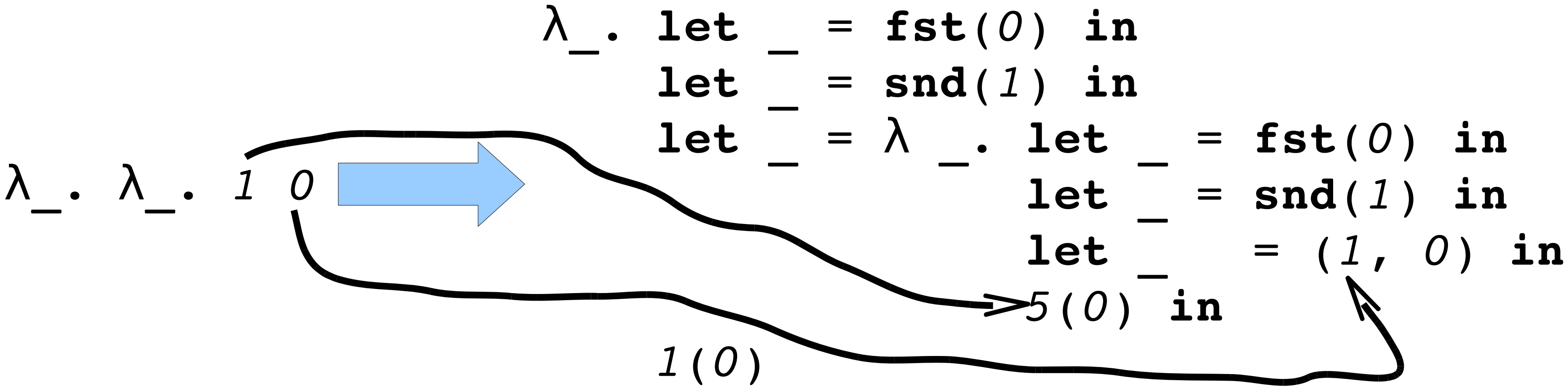
```
λp. let f = fst(p) in
    let k = snd(p) in
    let F = λp'. let x = fst(p') in
                 let k' = snd(p') in
                 let p'' = (x, k') in
                 f(p'') in
    k(F)
```

Every theorem must take **premises characterizing which variables are free in which terms**.

Every translation must **compute with these free variable sets** to come up with “fresh” names.

Nominal logic doesn't seem to help much here.

De Bruijn Indices



```
λ_. λ_. 1 0
```

```
λ_. let _ = fst(0) in
    let _ = snd(1) in
    let _ = λ _. let _ = fst(0) in
                 let _ = snd(1) in
                 let _  = (1, 0) in
                 5(0) in
```

`1(0)`

Transplanting terms requires **recursive operations to adjust indices** for free variables.

The right way to translate a term is very **context-dependent**.

Locally-nameless representation doesn't fare any better; we still need to adjust indices eventually.

Higher-Order Binding (HOAS)

$$
\begin{array}{ll}
\lambda(\underline{\lambda}f.\ \lambda(\underline{\lambda}x.\ f\ x)) \Rightarrow & \lambda(\underline{\lambda}p.\ \textbf{let fst}(p)\ (\underline{\lambda}f. \\
& \quad \textbf{let snd}(p)\ (\underline{\lambda}k. \\
& \quad \textbf{let}\ \lambda(\underline{\lambda}p'.\ \textbf{let fst}(p')\ (\underline{\lambda}x. \\
& \qquad \textbf{let snd}(p')\ (\underline{\lambda}k'. \\
& \qquad \textbf{let}\ (x,\ k')\ (\underline{\lambda}p''. \\
& \qquad f(p'')))))\ (\underline{\lambda}F. \\
& \quad k(F)))))
\end{array}
$$

Use the meta language's binders to encode object language binders.

Writing useful recursive functions over syntax is hard.

Several recent languages mitigate this problem with type systems that track variable contexts explicitly, bringing back some of the pain of first-order systems.

Substitution Commutes, etc.

Source program evaluates:

$$\frac{\texttt{e1\{x} \to \texttt{e2\}} \Rightarrow \texttt{v}}{\texttt{(}\lambda\texttt{x. e1) e2} \Rightarrow \texttt{v}}$$

Output program evaluates:

$$\frac{\dfrac{\texttt{[e1\{x} \to \texttt{e2\}]} \Rightarrow \texttt{[v]}}{\texttt{[e1]\{x} \to \texttt{[e2]\}} \Rightarrow \texttt{[v]}}\ \textbf{IH}}{\texttt{(}\lambda\texttt{x. [e1]) [e2]} \Rightarrow \texttt{[v]}}$$

$$\Vert$$

$$\texttt{[(}\lambda\texttt{x. e1) e2]} \Rightarrow \texttt{[v]}$$

Theorem. For all `e`, `x`, `e'`:

$$\texttt{[e]\{x} \to \texttt{[e']\}} = \texttt{[e\{x} \to \texttt{e'\}]}$$

POPL-ish proof. By a routine induction on `e`. ☐

Mechanized proof. “As usual with mechanizations using de Bruijn indices, the definitions of substitution and lifting plus the proofs of their properties take up a large part of our development.”

-- Dargaye and Leroy, “Mechanized Verification of CPS Translations”, 2007

(1685 lines of proof about “substitutions and their properties”)

Big Headaches of Verifying FP Compilers

- Reasoning about generated binders
- Characterizing interactions of substitution and friends with the different code transformations
- Overall high level of mostly irrelevant detail

The Challenge: A Realistic Verified Compiler for a Functional Language

- Compile a Mini-ML to assembly
  - Side effects included (references, exceptions)
- Development methodology that plausibly scales to production compilers
  - Minimize explicit context manipulation
  - **Zero** theorems about substitution!
  - Every proof automated
    - Proved by an adaptive program, rather than a proof tree
    - Proofs can keep working when the language changes

The State of the Art

- Compilers for first-order languages
  - Piton project [Moore 1989], CompCert [Leroy 2006]
- One-pass compilers for functional langs.
  - Flatau 1992, Benton & Hur 2009, ...
- Binders-to-binders phases for functional langs.
  - Minamide & Okuma 2003: Isabelle/HOL, concrete
  - Tian 2006: Twelf, HOAS
  - Chlipala 2007/2008: Coq, dependent de Bruijn/PHOAS
  - Dargaye & Leroy 2007: Coq, de Bruijn

From Mini-ML to Assembly

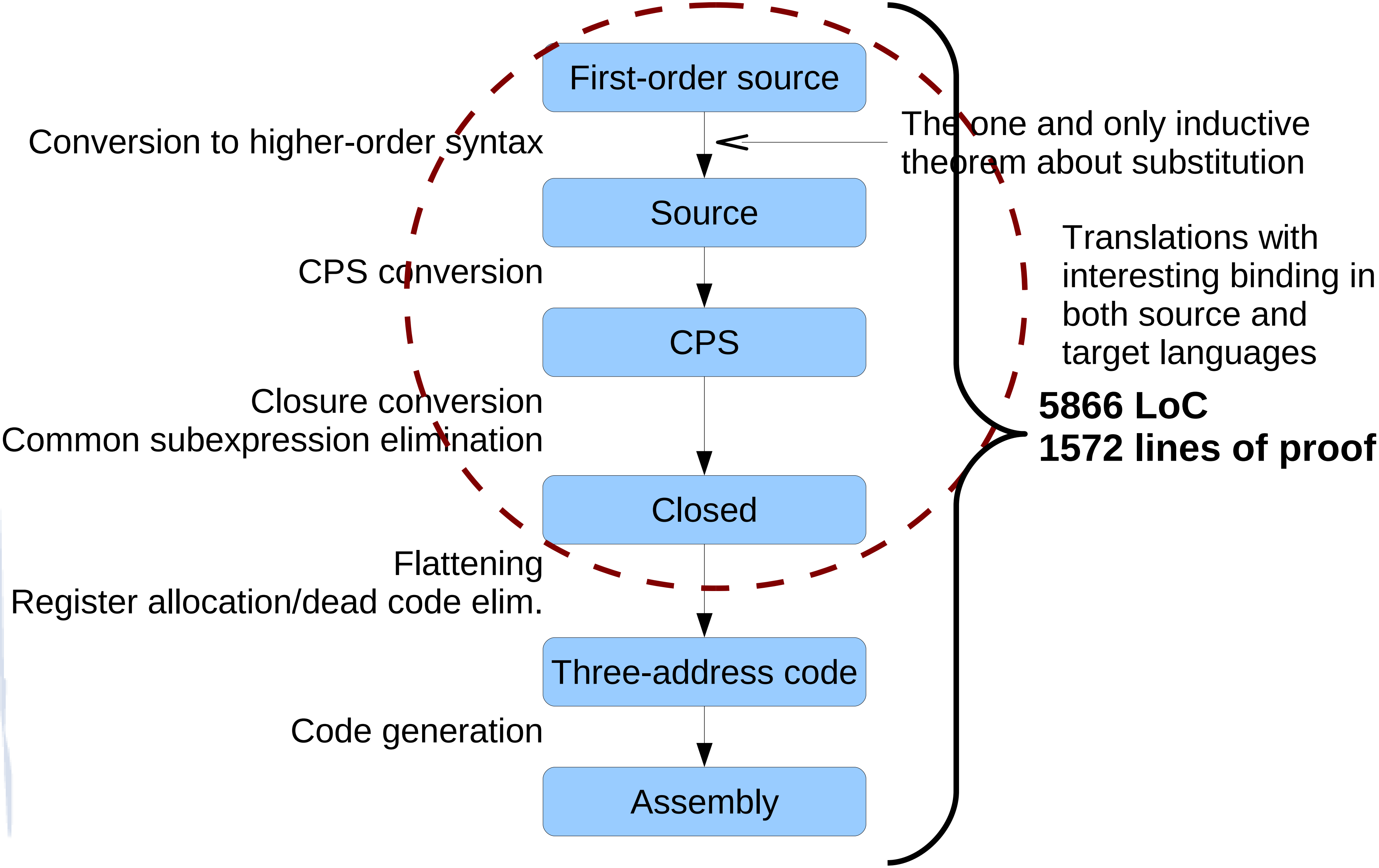
Source language

```
e ::= x | e e | λx. e | let x = e in e | ()
    | (e, e) | fst(e) | snd(e) | inl(e) | inr(e)
    | case e of inl(x) => e | inr(x) => e
    | ref(e) | !e | e := e
    | raise(e) | e handle x => e
```

Target language

```
     Lvalues L ::= r | [r + n] | [n]
     Rvalues R ::= n | r | [r + n] | [n]
Instructions I ::= L := R | r += n | jnz R, n
       Jumps J ::= halt | fail | jmp R
Basic blocks B ::= (I*, J)
    Programs P ::= (B*, B)
```

Phase Structure



First-order source

Conversion to higher-order syntax

Source

CPS conversion

CPS

Closure conversion
Common subexpression elimination

Closed

Flattening
Register allocation/dead code elim.

Three-address code

Code generation

Assembly

The one and only inductive theorem about substitution

Translations with interesting binding in both source and target languages

5866 LoC
1572 lines of proof

Overall Compiler Correctness

“If the input program terminates normally or with an uncaught exception, the output program terminates in the same way.”

Concrete Syntax in Coq

```
Inductive exp : Type :=
  | Var : string -> exp
  | Abs : string -> exp -> exp
  | App : exp -> exp -> exp.
```

HOAS ~~in Coq~~

```
Inductive exp : Type :=
  | Abs : (exp -> exp) -> exp
  | App : exp -> exp -> exp.

Definition uhoh (e : exp) : exp :=
  match e with
    | Abs f => f (Abs f)
    | _ => e
  end.
```

`uhoh (Abs uhoh)`

Parametric HOAS in Coq

[inspired by Washburn & Weirich, ICFP'03; adapted to Coq and semantics by Chlipala, ICFP'08]

```
Section var.
  Variable var : Type.

  Inductive exp : Type :=
    | Var : var -> exp
    | Abs : (var -> exp) -> exp
    | App : exp -> exp -> exp.
End var.

Definition Exp := forall var, exp var.

fun var => Abs var (fun x => Var var x)
```

Counting Tree Size

```
Fixpoint size (e : exp unit) : nat :=
  match e with
    | Var _  => 1
    | Abs f => size (f tt)
    | App e1 e2 => size e1 + size e2
  end.

Definition Size (E : Exp) : nat :=
  size (E unit).
```

CPS Conversion

```
cpsExp : forall var,
   Source.exp var              (* Source program *)
   -> (var -> CPS.exp var) (* Success continuation *)
   -> (var -> CPS.exp var) (* Exception handler *)
   -> CPS.exp var              (* Output program *)

Definition CpsExp (E : Source.Exp) : CPS.Exp :=
   fun var => cpsExp var (E var)
      (fun _ => Halt) (fun _ => Fail).
```

```
λf. λx. f x
```

```
λp. let f = fst(p) in
    let k = snd(p) in
    let F = λp'. let x = fst(p') in
                 let k' = snd(p') in
                 let p'' = (x, k') in
                 f(p'') in
    k(F)
```

Closure Conversion/Hoisting

- Choose `var` = `nat` to convert to first-order form locally.
- Main translation consumes a **proof** that the input expression is well-formed.

```
let F = λx.
    let G = λy.
        ...x...y... in
    ... in
F(z)
```

```
let G' = λp.
    let x = fst(p) in
    let y = snd(p) in
    ...x...y... in
let F' = λp.
    let x = snd(p) in
    let G = (G', x) in
    ... in
let F = (F', ()) in
let f = fst(F) in
let env = snd(F) in
let p = (env, z) in
f(p)
```

Common Subexpression Elimination

```
Inductive sval : Type := .... (* Symbolic values *)

cseExp : forall var,
    exp (var * sval)      (* Source program *)
    -> list (var * sval) (* Available expressions *)
    -> exp var            (* Output program *)

cseExp (Let e1 e2) xs =
    let sv = eval xs e1 in
    match lookup xs sv with
        | None => Let e1 (fun x =>
            cseExp (e2 (x, sv)) ((x, sv) :: xs))
        | Some x => cseExp (e2 (x, sv)) xs
    end
```

```
let x = fst(a) in
let y = snd(a) in
let z = (y, x) in
let u = fst(z) in
let p = (u, k') in
k(p)
```

```
let y = snd(a) in



let p = (y, k') in
k(p)
```

Flattening

- As with closure conversion, choose `var` = `nat` to convert to first-order form locally.

```
let F = λx.
    let y = ...x... in
    ...y...x... in
let G = λx. .... in
...F...G...
```

```
F:
[1] := ...[0]...;
...[1]...[0]...
G:
....
main:
...0...1...
```

Design Point #1:

Higher-order syntax avoids the need to reason about fresh name generation or index shuffling.

The Right Semantics for the Job?

Which encoding of dynamic semantics makes correctness easiest to prove?

Standard Big-Step Semantics

$$\frac{}{\lambda x.\ e \downarrow \lambda x.\ e}$$

$$\frac{e1 \downarrow \lambda x.\ e \qquad e2 \downarrow u \qquad e\{x \rightarrow u\} \downarrow v}{e1\ e2 \downarrow v}$$

Substitution Commutes, etc.

Source program evaluates:

$$\frac{\mathtt{e1\{x \rightarrow e2\}} \Downarrow \mathtt{v}}{\mathtt{(\lambda x.\ e1)\ e2} \Downarrow \mathtt{v}}$$

Output program evaluates:

$$\frac{\dfrac{\mathtt{[e1\{x \rightarrow e2\}]} \Downarrow \mathtt{[v]}}{\mathtt{[e1]\{x \rightarrow [e2]\}} \Downarrow \mathtt{[v]}}\ \textbf{IH}}{\mathtt{(\lambda x.\ [e1])\ [e2]} \Downarrow \mathtt{[v]}}$$

$$||$$

$$\mathtt{[(\lambda x.\ e1)\ e2]} \Downarrow \mathtt{[v]}$$

Theorem. For all `e`, `x`, `e'`:

`[e]{x → [e']} = [e{x → e'}]`

Environment Semantics

$$\frac{}{(E,\ x) \Downarrow E(x)}$$

$$\frac{}{(E,\ \lambda x.\ e) \Downarrow (E,\ \lambda x.\ e)}$$

$$\frac{\begin{array}{c}(E,\ e1) \Downarrow (E',\ \lambda x.\ e) \\ (E,\ e2) \Downarrow u \qquad (E'\{x \to u\},\ e) \Downarrow v\end{array}}{(E,\ e1\ e2) \Downarrow v}$$

Context Shuffling

f, x

λf. λx. f x

```
λp. let f = fst(p) in
    let k = snd(p) in
    let F = λp'. let x = fst(p') in
                 let k' = snd(p') in
                 let p'' = (x, k') in
                 f(p'') in
    k(F)
```

p, f, k, p', x, k'

Theorem. For all `E`, `e`, `v`, and `E'`,
If `(E, e) ⬇ v`,
And `E'` extends `E`,
Then `(E', e) ⬇ v`.

Freshness side conditions?
Index adjustment in `e`?

Another Tack?

Program transformations tend to manipulate *binding structure* more than they manipulate the *structure of values* that occur at runtime.

Can we design a dynamic semantics that takes advantage of this property to reduce bureaucracy?

A First Attempt

```
Section var.
  Variable var : Type.

  Inductive exp : Type :=
    | Var : var -> exp
    | Abs : (var -> exp) -> exp
    | App : exp -> exp -> exp.

  Inductive val : Type :=
    | VAbs : (~~val~~ -> exp) -> val.
End var.
```

$$\frac{\mathtt{e1} \downarrow \lambda \mathtt{f} \qquad \mathtt{e2} \downarrow \mathtt{u} \qquad \mathtt{f(u)} \downarrow \mathtt{v}}{\mathtt{e1\ e2} \downarrow \mathtt{v}}$$

Closure Semantics

$$\frac{}{(H,\ \lambda f) \Downarrow (f :: H,\ |H|)}$$

$$\frac{\begin{array}{c}(H1,\ e1) \Downarrow (H2,\ n) \\ (H2,\ e2) \Downarrow (H3,\ u) \qquad H3.n = f \\ (H3,\ f(u)) \Downarrow (H4,\ v)\end{array}}{(E,\ e1\ e2) \Downarrow (H4,\ v)}$$

```
Definition val := nat.
Definition closure := val -> exp val.
Definition closures := list closure.

Inductive eval : closures -> exp val
  -> closures -> val -> Prop := ....
```

For the Full Language

```
Inductive val : Type :=
| VFunc : nat -> val
| VUnit : val
| VPair : val -> val -> val
| VInl : val -> val
| VInr : val -> val
| VRef : nat -> val.
```

Design Point #2:

Closure semantics avoids the need to compensate semantically for changes in binding structure.

(Instead, we compensate for changes in closure heap structure, which are less common and simpler.)

And now, the whole CPS conversion correctness proof!

In 163 lines.

Relating Values, Part I

```
Section cr.
  Variable s1 : Core.closures.
  Variable s2 : CPS.closures.

  Import Core.

  Inductive cr : Core.val -> CPS.val -> Prop :=
  | EquivArrow : forall l1 l2 G
    (f1 : Core.val -> Core.exp Core.val)
    (f2 : CPS.val -> Core.exp CPS.val),
    (forall x1 x2, exp_equiv ((x1, x2) :: G) (f1 x1) (f2 x2))
    -> (forall x1 x2, In (x1, x2) G -> cr x1 x2)
    -> s1 # l1 = Some f1
    -> s2 # l2 = Some (cpsFunc f2)
    -> cr (Core.VFunc l1) (CPS.VCont l2)

  | EquivUnit : cr Core.VUnit CPS.VUnit

  | EquivPair : forall x1 x2 y1 y2,
    cr x1 x2
    -> cr y1 y2
    -> cr (Core.VPair x1 y1) (CPS.VPair x2 y2)
```

Relating Values, Part II

```
  | EquivInl : forall v1 v2,
    cr v1 v2
    -> cr (Core.VInl v1) (CPS.VInl v2)
  | EquivInr : forall v1 v2,
    cr v1 v2
    -> cr (Core.VInr v1) (CPS.VInr v2)

  | EquivRef : forall l,
    cr (Core.VRef l) (CPS.VRef l).

  Inductive crr : Core.result -> result' -> Prop :=
  | EquivAns : forall v1 v2,
    cr v1 v2
    -> crr (Core.Ans v1) (Ans' v2)
  | EquivEx : forall v1 v2,
    cr v1 v2
    -> crr (Core.Ex v1) (Ex' v2).
End cr.

Notation "s1 & s2 |-- v1 ~~ v2" := (cr s1 s2 v1 v2) (no associativity,
at level 70).
Notation "s1 & s2 |--- r1 ~~ r2" := (crr s1 s2 r1 r2) (no associativity,
at level 70).
```

The Relation is Monotone

```
Hint Constructors cr crr.

Section cr_extend'.
  Variables s1 s1' : Core.closures.
  Variables s2 s2' : CPS.closures.
  Hypothesis H1 : s1 ~> s1'.
  Hypothesis H2 : s2 ~> s2'.

  Lemma cr_extend' : forall v1 v2,
    s1 & s2 |-- v1 ~~ v2
    -> s1' & s2' |-- v1 ~~ v2.
    induction 1; eauto.
  Qed.
End cr_extend'.

Theorem cr_extend : forall s1 s2 s1' s2' v1 v2,
  s1 & s2 |-- v1 ~~ v2
  -> s1 ~> s1'
  -> s2 ~> s2'
  -> s1' & s2' |-- v1 ~~ v2.
  intros; eapply cr_extend'; eauto.
Qed.

Hint Resolve cr_extend.
```

Some More Lemmas

```
Lemma cr_push : forall v1 v2 v1' v2' G s1 s2,
  In (v1, v2) ((v1', v2') :: G)
  -> s1 & s2 |-- v1' ~~ v2'
  -> (forall v3 v4, In (v3, v4) G -> s1 & s2 |-- v3 ~~ v4)
  -> s1 & s2 |-- v1 ~~ v2.
  simpler.
Qed.

Hint Resolve cr_push.

Notation "s1 & s2 |-- h1 ~~~ h2" := (sall (cr s1 s2) h1 h2) (no
associativity, at level 70).

Lemma EquivRef' : forall s1 s2 h1 h2,
  s1 & s2 |-- h1 ~~~ h2
  -> s1 & s2 |-- Core.VRef (length h1) ~~ VRef (length h2).
  intros; match goal with
            | [ H : _ |- _ ] => rewrite <- (sall_length H)
          end; auto.
Qed.
```

Answers

```
Definition answer (r : result') (P1 : val -> Prop) (P2 : val -> Prop) :=
   match r with
     | Ans' v => P1 v
     | Ex' v => P2 v
   end.

Theorem answer_Ans : forall (P1 : _ -> Prop) P2 v,
  P1 v
  -> answer (Ans' v) P1 P2.
  tauto.
Qed.

Theorem answer_Ex : forall P1 (P2 : _ -> Prop) v,
  P2 v
  -> answer (Ex' v) P1 P2.
  tauto.
Qed.
```

The Main Lemma Statement

```
Lemma cpsExp_correct : forall s1 h1 (e1 : Core.expV) s1' h1' r1,
  Core.eval s1 h1 e1 s1' h1' r1
  -> forall G (e2 : Core.exp CPS.val),
    Core.exp_equiv G e1 e2
    -> forall s2 h2,
      (forall v1 v2, In (v1, v2) G -> s1 & s2 |-- v1 ~~ v2)
      -> s1 & s2 |-- h1 ~~~ h2
      -> forall k ke, exists s2', exists h2', exists r2,
        (forall r,
          answer r2
          (fun v2 => CPS.eval s2' h2' (k v2) r
            -> CPS.eval s2 h2 (cpsExp e2 k ke) r)
          (fun v2 => CPS.eval s2' h2' (ke v2) r
            -> CPS.eval s2 h2 (cpsExp e2 k ke) r))
        /\ s1' & s2' |--- r1 ~~ r2
        /\ s2 ~> s2'
        /\ s1' & s2' |-- h1' ~~~ h2'.
```

Hints for the Main Proof

```
Hint Constructors CPS.evalP.
Hint Resolve answer_Ans answer_Ex.
Hint Resolve CPS.EvalCaseL CPS.EvalCaseR EquivRef'.

Hint Extern 1 (CPS.eval _ _ (cpsFunc _ _) _) =>
  unfold cpsFunc, cpsFunc'.

Hint Extern 1 (CPS.eval ((fun x => ?ke x)
                      :: (fun x => ?k x) :: _) _ _ _) =>
  rewrite (eta_eq ke); rewrite (eta_eq k).
```

The Main Proof

```
induction 1; abstract (inversion 1; simpler;
  repeat (match goal with
            | [ H : _ & _ |-- _ ~~ _ |- _ ] => invert_1_2 H
            | [ H : _ & _ |--- _ ~~ _ |- _ ] => invert_1 H
            | [ H : forall G e2, Core.exp_equiv G ?E e2 -> _ |- _ ] =>
              match goal with
                | [ _ : Core.eval ?S _ E _ _ _,
                    _ : Core.eval _ _ ?E' ?S _ _,
                    _ : forall G e2, Core.exp_equiv G ?E' e2 -> _
                    |- _ ] => fail 1
                | _ => match goal with
                         | [ k : val -> expV, ke : val -> exp val,
                             _ : _ & ?s |-- _ ~~ _,
                             _ : context[VCont] |- _ ] =>
                           guessWith (ke :: k :: s) H
                         | _ => guess H
                       end
              end
          end; simpler);
  try (match goal with
         | [ H1 : _, H2 : _ |- _ ] => generalize (sall_grab H1 H2)
       end; simpler); splitter; eauto 9 with cps_eval; intros;
   try match goal with
          | [ H : _ & _ |--- _ ~~ ?r |- answer ?r _ _ ] =>
            inverter H; simpler; eauto 9 with cps_eval
        end).
```

The Final Theorem

```
Definition cpsResult (r : Core.result) :=
  match r with
    | Core.Ans _ => Ans
    | Core.Ex _ => Ex
  end.

Theorem CpsExp_correct : forall (E : Core.Exp) s h r,
  Core.Eval nil nil E s h r
  -> Core.Exp_wf E
  -> CPS.Eval nil nil (CpsExp E) (cpsResult r).
  Hint Constructors CPS.eval.

  unfold CpsExp, CPS.Eval; intros until r; intros H1 H2;
    generalize (cpsExp_correct H1 (H2 _ _)
      (s2 := nil) (fun _ _ pf => match pf with end)
      (sall_nil _) (fun _ => EHalt) (fun _ => EUncaught)); simpler;
    match goal with
      | [ H : _ & _ |--- _ ~~ _ |- _ ] => destruct H
    end; simpler.
Qed.
```

Implementation Stats

- 5866 lines in total for the whole compiler.
- 1572 lines are proof script, hints, or tactic definitions.
- The first version had no `let` at source level. Adding `let` required ~30 new lines.
  - Update language syntax and semantics
  - No old lines modified
  - No lines of proof added

Conclusion

- It's halfway believable that this methodology scales to real, evolving compilers.
- No lemmas about substitution or other purely syntactic administrivia!
- Programmatic proofs adapt to changing specifications.

Code available in the latest **Lambda Tamer** distribution:
`http://ltamer.sourceforge.net/`

Backup Slides

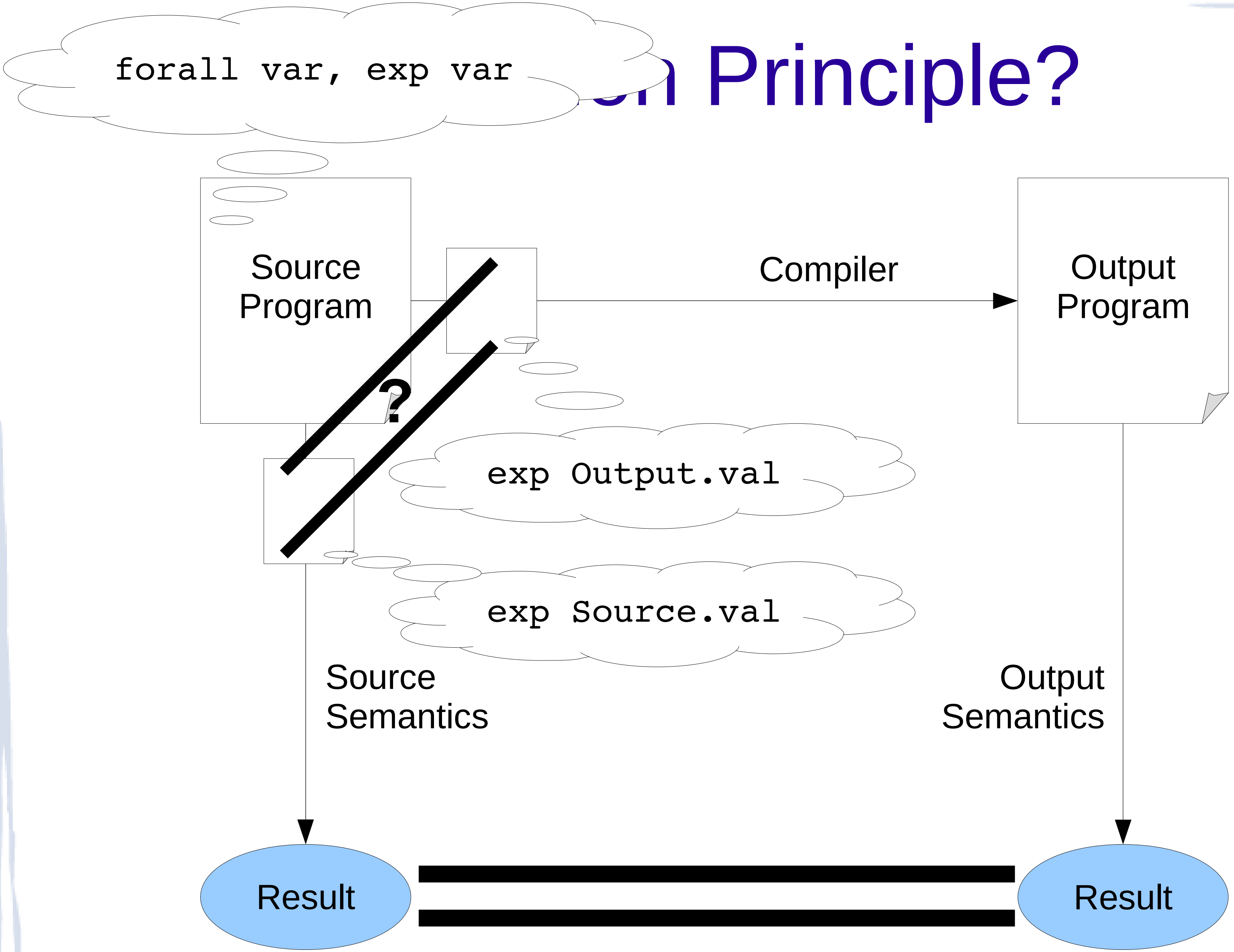
Counting Occurrences of a Variable

```
Fixpoint count (e : exp bool) : nat :=
  match e with
    | Var true => 1
    | Var false => 0
    | Abs f => count (f false)
    | App e1 e2 => count e1 + count e2
  end.

Definition Exp1 :=
  forall var, var -> exp var.

Definition Count (E : Exp1) : nat :=
  count (E bool true).
```

Building a New Term

```
Section var.
  Variable var : Type.

  Fixpoint swap (e : exp var) : exp var :=
     match e with
        | Var x => Var x
        | Abs f => Abs (fun x => swap (f x))
        | App e1 e2 => App (swap e2) (swap e1)
     end.
End var.

Definition Swap (E : Exp) : Exp :=
  fun var => swap (E var).
```

n Principle?


forall var, exp var
Source Program
Compiler
Output Program
?
exp Output.val
exp Source.val
Source Semantics
Output Semantics
Result
Result

Relating Instantiated Terms

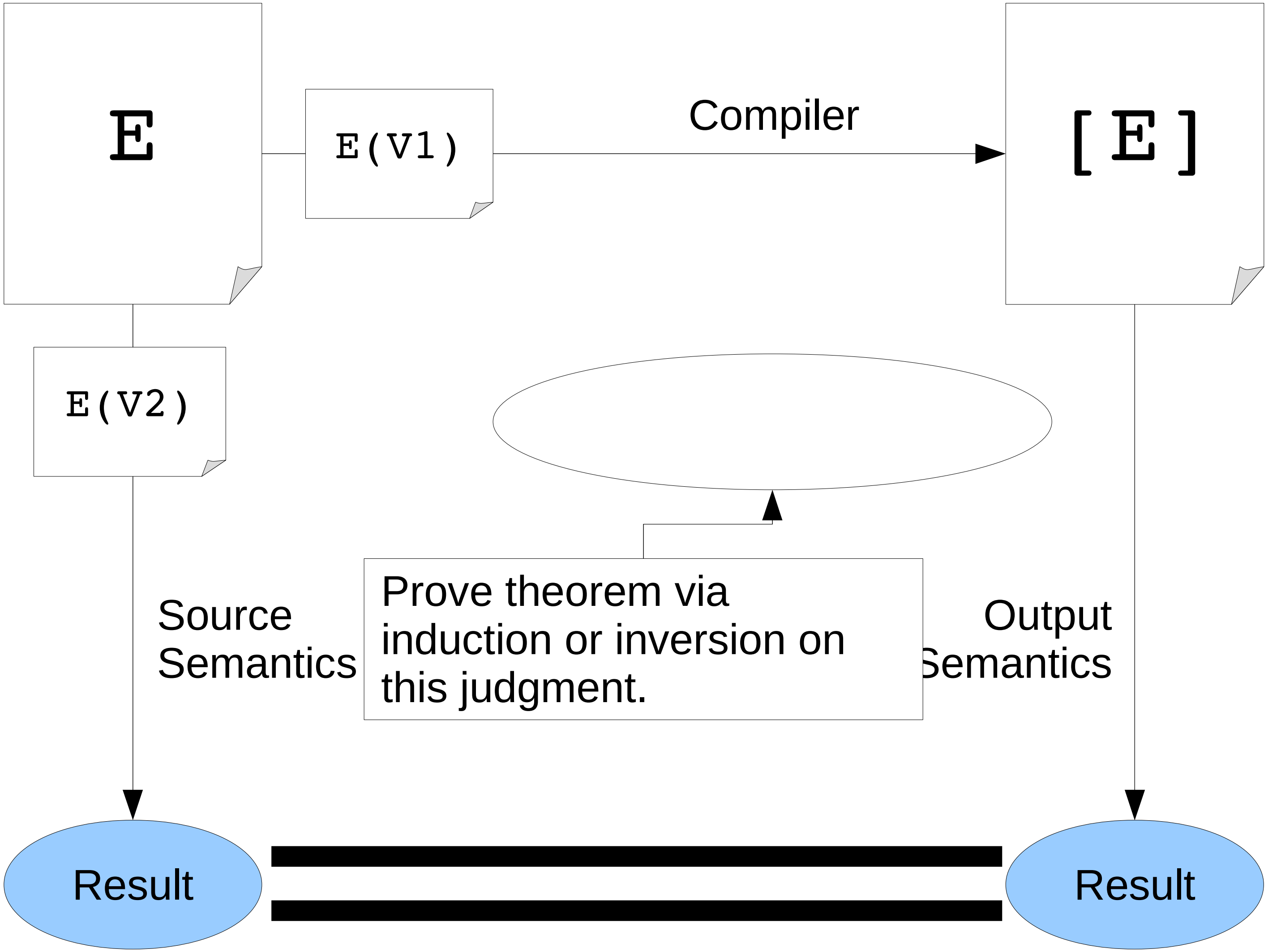
$$\frac{(\texttt{x, y}) \in \Gamma}{\Gamma \vdash \texttt{x} \sim \texttt{y}}$$

$$\frac{\Gamma \vdash \texttt{e1} \sim \texttt{e1'} \qquad \Gamma \vdash \texttt{e2} \sim \texttt{e2'}}{\Gamma \vdash \texttt{e1 e2} \sim \texttt{e1' e2'}}$$

$$\frac{\forall \texttt{x,y: } \Gamma\texttt{,(x, y)} \vdash \texttt{f1(x)} \sim \texttt{f2(y)}}{\Gamma \vdash \lambda\texttt{f1} \sim \lambda\texttt{f2}}$$

$$\frac{\forall \texttt{V1,V2:} \vdash \texttt{E(V1)} \sim \texttt{E(V2)}}{\texttt{E}\ \textbf{wf}}$$

Induction Principle?



E

E(V1)

Compiler

[E]

E(V2)

Source Semantics

Prove theorem via induction or inversion on this judgment.

Output Semantics

Result

Result