Parametric Higher-Order Abstract Syntax for Mechanized Semantics

Adam Chlipala
Harvard University
ICFP 2008

The Big Picture

- We want to write programs that manipulate syntax including *variable binders*...
- ...in the language of a proof assistant, which enforces termination of all functions...
- ...and then we want to build short, mechanized proofs that we did it right.

Running Example: CPS Translation

Source Language:

$\tau ::= \textbf{bool} \mid \tau \to \tau$

$e ::= x \mid \textbf{true} \mid \textbf{false} \mid e\ e \mid \lambda x : \tau.\ e$

Target Language:

$\tau ::= \textbf{bool} \mid \tau \to 0 \mid \tau \times \tau$

$p ::= \textbf{true} \mid \textbf{false} \mid \lambda x : \tau.\ e \mid (x, x) \mid x.1 \mid x.2$

$e ::= x\ x \mid \textbf{let}\ x = p\ \textbf{in}\ e$

$(\lambda x : \textbf{bool}.\ x)\ \textbf{true}$

$\textbf{let}\ f = \lambda p.$
$\quad \textbf{let}\ x = p.1\ \textbf{in}$
$\quad \textbf{let}\ k = p.2\ \textbf{in}$
$\quad k\ x\ \textbf{in}$
$\textbf{let}\ t = \textbf{true}\ \textbf{in}$
$\textbf{let}\ p = (t, k_{\textbf{top}})\ \textbf{in}$
$f\ p$

An Embarrassment of Riches

Choices for representing binders:

- Concrete syntax
- De Bruijn indices/levels
- Locally nameless
- Nominal logic
- Higher-order abstract syntax
- ...

- Why not add one more? :-)

Concrete Syntax

[Church 1936?]

```
type var = string

type typ =
    | Bool
    | Arrow of typ * typ

type exp =
    | Var of var
    | True
    | False
    | App of exp * exp
    | Abs of var * exp
```

Concrete CPS Translation

```
let rec cpsExp (e : exp) (k : var -> cexp) : cexp =
    match e with
        | Var x -> k x
        | True -> withFresh (fun x -> Let x CTrue (k x))
        | False -> withFresh (fun x -> Let x CFalse (k x))
        | App e1 e2 -> cpsExp e1 (fun  f : var ->
            cpsExp e2 (fun x ->
                withFresh (fun k’ ->
                    Let k’ (withFresh (fun a -> CAbs a (k a)))
                    (withFresh (fun p ->
                        Let p (Pair (x, k’)) (Call f p)))))
        | Abs x e’ -> withFresh (fun f ->
            Let f (withFresh (fun p -> CAbs p (
                Let x (Fst p)
                    (withFresh (fun k’ -> Let k’ (Snd p)
                        cpsExp e’ (fun r -> Call k’ r)))))
            (k f))
```

De Bruijn CPS Translation

```
let rec cpsExp (e : exp) (k : int -> var -> cexp) : cexp =
    match e with
        | Var x -> k 0 x
        | True -> Let CTrue (k 1 0)
        | False -> Let CFalse (k 1 0)
        | App e1 e2 -> cpsExp e1 (fun (nf : int) (f : var) ->
              cpsExp e2 (fun nx x ->
                  Let (CAbs (k (nf + nx) 0)))
                      (Let (Pair (x + 1, 0))
                          (Call (f + nx + 2, 0)))))
        | Abs e' -> Let (CAbs (
                  Let (Snd 0)
                      (Let (Fst 1)
                          cpsExp e' (fun nr r ->
                              Call (nr + 1) r))))
              (k 1 0)
```

Higher-Order Abstract Syntax

[Church 1940]

```
type exp =
    | True
    | False
    | App of exp * exp
    | Abs of exp -> exp
```

$\lambda x, \lambda y, x$ Abs (fun *x* -> Abs (fun _ -> *x*))

```
let maybeBeta e =
    match e with
        | App (Abs f) e’ -> f e’
        | _ -> e
```

Pandora’s Box

```
let f e =
    match e with
        | Abs e’ -> e’ e
        | _ -> e
```

```
type exp =
    | True
    | False
    | App of exp * exp
    | Abs of exp -> exp
```

f (Abs f)

- ➔ **match** Abs f **with** Abs e’ -> e’ (Abs f) | _ -> Abs f
- ➔ f (Abs f)
- ➔ **match** Abs f **with** Abs e’ -> e’ (Abs f) | _ -> Abs f
- ➔ f (Abs f)
- ➔ ....

Weak HOAS

[Despeyroux et al. 1995, Honsell et al. 2001]

```
type var      (* Abstract type! *)

type exp =
    | Var of var
    | True
    | False
    | App of exp * exp
    | Abs of var -> exp
```

$\lambda x, \lambda y, x$ ⇒ Abs (fun *x* -> Abs (fun _ -> Var *x*))

Getting Dependent

```
type 't var

type 't exp =
    | Var : 't var -> 't exp
    | True : bool exp
    | False : bool exp
    | App : ('d -> 'r) exp * 'd exp -> 'r exp
    | Abs : ('d var -> 'r exp) -> ('d -> 'r) exp
```

Getting Dependent

~~**type** '*t* var~~

```
type ('t, 'V) exp =
    | Var : 't 'V -> ('t, 'V) exp
    | True : (bool, 'V) exp
    | False : (bool, 'V) exp
    | App : ('d -> 'r, 'V) exp * ('d, 'V) exp -> ('r, 'V) exp
    | Abs : ('d 'V -> ('r, 'V) exp, 'V) -> ('d -> 'r, 'V) exp
```

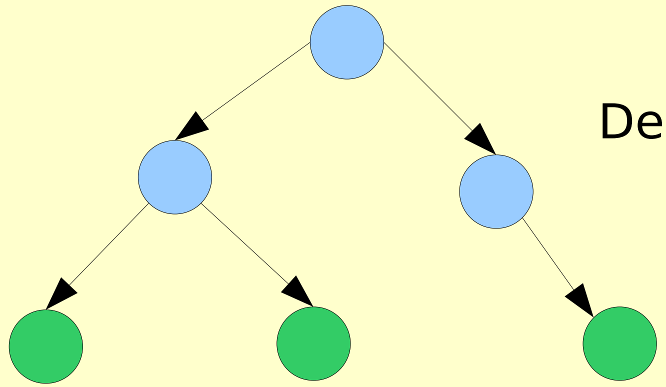
```
type 't Exp = forall 'V. 't exp('V)
(* [Boxes Go Bananas, Washburn and Weirich 2003] *)

(* Will call this representation scheme “Parametric HOAS,” or “PHOAS”…. *)
```

PHOAS CPS Translation

```
let rec cpsExp (e : ('t, 'V o cpsTyp) exp)
      (k : (cpsTyp 't) 'V -> 'V cexp) : 'V cexp =
   match e with
    | Var x -> k x
    | True -> Let CTrue k
    | F
    | A                                              ->
               Let (Pair (x, k')) (fun p ->
                  Call f p))))
    | Abs e' -> Let (CAbs (fun p ->
         Let (Fst p) (fun x ->
            Let (Snd p) (fun k' ->
               cpsExp (e' x) (fun r -> Call k' r))))) k
```

```
(Var x) : ('t, 'V o cpsTyp) exp
x : 't ('V o cpsTyp)
x : (cpsTyp 't) 'V
(k x) : 'V cexp
```



Concrete: `x y z`

De Bruijn: 0 1 2

Denotations: λ λ λ

```
Fixpoint
    matc
      | Bo
      | A                                        te t2
    end.
```

```
Fixpoint expDenote (t : typ) (e : exp typDenote t)
      : typDenote t =
    match e with
      | Var x => x
      | True => true
      | False => false
      | App (e1, e2) => (expDenote e1) (expDenote e2)
      | Abs e' => fun x => expDenote (e' x)
    end.
```

Total Correctness Theorem

Theorem: For any source term *E* of type bool,
CexpDenote (CpsExp *E* (**fun** *x* -> Halt *x*)) = ExpDenote *E*

Axiom: For any source term *E*
and variable types *U* and *V*,
(*E U*) and (*E V*) are syntactically equivalent,
up to replacement of *U*'s with *V*'s.

Informally, an easy consequence of a **parametricity** theorem....

A Proof

```
Scheme pterm_mut := Induction for pterm Sort Prop
with pprimop_mut := Induction for pprimop Sort Prop.

Section splice_correct.
  Variables result1 result2 : ptype.

  Variable e2 : ptypeDenote result1
    -> pterm ptypeDenote result2.

  Theorem splice_correct : forall e1 k,
    ptermDenote (splice e1 e2) k
    = ptermDenote e1 (fun r => ptermDenote (e2 r) k).
    apply (pterm_mut
      (fun e1 => forall k,
        ptermDenote (splice e1 e2) k
        = ptermDenote e1 (fun r => ptermDenote (e2 r) k))
      (fun t p => forall k,
        pprimopDenote (splicePrim p e2) k
        = pprimopDenote p (fun r => ptermDenote (e2 r) k)));
    equation.
  Qed.
End splice_correct.

Fixpoint lr (t : type) : typeDenote t -> ptypeDenote (cpsType t)
     -> Prop :=
  match t
    return (typeDenote t -> ptypeDenote (cpsType t) -> Prop) with
    | TBool => fun n1 n2 => n1 = n2
    | TArrow t1 t2 => fun f1 f2 =>
      forall x1 x2, lr _ x1 x2
        -> forall k, exists r,
          f2 (x2, k) = k r
          /\ lr _ (f1 x1) r
  end.

Lemma cpsTerm_correct : forall G t (e1 : term _ t) (e2 : term _ t),
  term_equiv G e1 e2
  -> (forall t v1 v2, In (vars (v1, v2)) G -> lr t v1 v2)
  -> forall k, exists r,
    ptermDenote (cpsTerm e2) k = k r
    /\ lr t (termDenote e1) r.
  Hint Rewrite splice_correct : ltamer.

  Ltac my_changer :=
    match goal with
      [ H : (forall (t : _) (v1 : _) (v2 : _),
        vars _ = vars _ \/ In _ _ -> _) -> _ |- _ ] =>
      match typeof H with
        | ?P -> _ =>
          assert P; [intros; push_vars; intuition; fail 2
            | idtac]
      end
    end.

  Ltac my_simpler := repeat progress (equation;
      fold ptypeDenote in *;
    fold cpsType in *; try my_changer).

  Ltac my_chooser T k :=
    match T with
      | bool => fail 1
      | type => fail 1
      | ctxt _ _ => fail 1
      | _ => default_chooser T k
    end.

  induction 1; matching my_simpler my_chooser; eauto.
Qed.

Theorem CpsTerm_correct : forall t (E : Term t),
  forall k, exists r,
    PtermDenote (CpsTerm E) k = k r
    /\ lr t (TermDenote E) r.
  unfold PtermDenote, TermDenote, CpsTerm; simpl; intros.
  eapply (cpsTerm_correct (G := nil)); simpl; intuition.
  apply Term_equiv.
Qed.

Theorem CpsTerm_correct_bool : forall (E : Term TBool),
  forall k, PtermDenote (CpsTerm E) k = k (TermDenote E).
  intros.
  generalize (CpsTerm_correct E k); firstorder congruence.
Qed.
```

Proof Size Comparison

Minamide & Okuma 2003	Isabelle/HOL	Concrete	~600
Tian 2006	Twelf	HOAS	~50
Dargaye & Leroy 2007	Coq	De Bruijn	~1700
This case study	Coq	PHOAS	~30
Expanded case study	Coq	PHOAS	~150

More PHOAS + Definitional Compilers

- Case studies in this paper:
  - Closure conversion for STLC
  - Compilation of ML-style pattern matching
  - CPS translation for System F
- In other ongoing work:
  - CPS translation for idealized Core ML

Conclusion

Lambda Tamer library available on the Web at:
`http://ltamer.sourceforge.net/`

PHOAS and definitional compilers combine to enable mechanized correctness proofs that are even easier to construct and maintain than those in ICFP papers.