Generic Programming and Proving for Programming Language Metatheory

Adam Chlipala
WMM 2007
### COQ Solutions on Paper

<table>
<thead>
<tr>
<th>Representation</th>
<th>Lemmas</th>
<th>Proof Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>de Bruijn</td>
<td>30</td>
<td>402</td>
</tr>
<tr>
<td>locally nameless</td>
<td>49</td>
<td>495</td>
</tr>
<tr>
<td>levels/names</td>
<td>56</td>
<td>938</td>
</tr>
<tr>
<td>de Bruijn (nested)</td>
<td>49</td>
<td>1574</td>
</tr>
<tr>
<td>locally nameless</td>
<td>23</td>
<td>75</td>
</tr>
<tr>
<td>locally nameless</td>
<td>22</td>
<td>101</td>
</tr>
</tbody>
</table>

*on paper* 2 2 pages

(from Aydemir/Chargueraud/Pierce/Pollack/Weirich 2007)
What's Wrong?

The Realm of the Obvious

(basic facts about variables, typing judgments, etc.)

Your Proof

Everyone at POPL believes these lemmas without even needing to see them stated formally....
What We Really Want

Language syntax and type system

Metatheory Wizard

Substitution operation, weakening lemma, permutation lemma, ....
But Isn't That Twelf?
Write It in OCaml?

You *really* don't want to write a code generator this complicated in ML.

Language syntax and type system

Substitution operation, weakening lemma, permutation lemma, ....

Metatheeory Wizard
Write It with Dependent Types?

We can do this in Coq itself!

Language syntax and type system

Metathteorpy Wizard

Once you satisfy the type checker, you know your “wizard” is sound!

Substitution operation, weakening lemma, permutation lemma, ....
Generic Programming with *Universes* in Type Theory

- Altenkirch and McBride with *Oleg*
- Pfeifer and Rueß with *Lego*
- Benke et al. with *Agda*
Universes

Universe Type
(A type that can be used to represent the types in C)

Class C of Types

Injection function
(written in the same type theory!)

Inverse Injection Function
(implemented outside the type theory)

Generic Function F
(written in the same type theory!)

Implementations of Generic Function F for Class C

Types Guarantee Compatibility!

In past work, implemented for:
- Simple inductive types
- Polymorphic inductive types
- Indexed dependent type families
- ....
Universes for ASTs

(* De Bruijn indices *)

```ocaml
type dbvar = int
```

(* Untyped lambda calculus terms *)

```ocaml
type term = [const; var; app; lam]
  | Const of int
  | Var of dbvar
  | App of term * term
  | Lam of term
```

const = {vars = 0; terms = []; data = int}
var = {vars = 1; terms = []; data = unit}
app = {vars = 0; terms = [0; 0]; data = unit}
lam = {vars = 0; terms = [1]; data = unit}

(* Universe for AST constructors *)

```ocaml
type constructor = {
  vars : int; (* How many variables? *)
  terms : int list; (* How many new binders around each subterm? *)
  data : Type; (* What other arguments? *)
}
```

(* Universe for AST languages *)

```ocaml
type language = constructor list
```
**Evidence**

What is the type of evidence that would convince us that con is really a constructor of term?

---

```ocaml
type term =
  | Const of int
  | Var of dbvar
  | App of term * term
  | Lam of term

let constructor_evidence (con : constructor) (term : Type) :
  Type =
repeat dbvar con.vars
  -> type_map (fun _ -> term) con.terms
  -> con.data
  -> term

let language_evidence (lang : language) (term : Type) :
  Type =
type_map (fun con -> constructor_evidence con term) lang

let rec repeat (t : Type) (n : int) :
  Type =
match n with
| 0 -> unit
| _ -> t * repeat t (n - 1)

let rec type_map (f : 'a -> Type) (ls : 'a list) :
  Type =
match ls with
| [] -> unit
| h :: t -> f h * type_map f t
```

---

What is the type of evidence that would convince us that con is really a constructor of term?
Reflecting Recursion

**Reflecting Recursion**

```ocaml
let term_rec _ (const, (var, (app, (lam, ())))) = function
  | Const n -> const () () n
  | Var x -> var (x, ()) () ()
  | App (e1, e2) -> app () ((e1, f e1), ((e2, f e2), ())) ()
  | Lam e -> app () ((e, f e), ()) ()

let rec f = function
  | Const n -> (* type = branch const term result *)
  | Var x -> (* type = branch var term result *)
  | App (e1, e2) -> (* type = branch app term result *)
  | Lam e -> (* type = branch lam term result *)

(* Build the type of one branch of a recursive definition. *)
let branch (con : constructor) (term : Type) (result : Type) : Type =
  repeat dbvar con.vars
  -> type_map (fun _ -> term * result) con.terms
  -> con.data
  -> result

(* Build the type of a recursor for an AST language. *)
let recursor_of_language (lang : language) (term : Type) : Type =
  forall result : Type.
  type_map (fun con -> branch con term result) lang
  -> (term -> result)
```

The recursive value on each subterm...
Reflecting Recursion

**type** term =
| Const of int
| Var of dbvar
| App of term * term
| Lam of term

**term_rep : language_evidence lang term =**
(cons_in, (var_in, (app_in, (lam_in, ()))),
term_rec)

(* What do we need to know about an AST language? *)
**let** language_evidence (lang : language) (term : **Type**) : **Type** =
type_map (**fun** con -> constructor_evidence con term) lang
* recursor_of_language lang term
A Generic Lift Function

let lift'_var (min : int) (x : dbvar) : dbvar =
  if x >= min then x + 1 else x

(* Helper function that lifts within a range of De Bruijn indices *)
let lift' (lang : language) (term : Type)
  ((builders, recurse) : language_evidence lang term)
: int -> term -> term =
  recurse term
    (type_remap (fun (con : constructor)
      (build : constructor_evidence con term) ->
        fun vars terms data (min : int) ->
          build (repeat_map (lift'_var min) vars)
            (type_remap (fun binders (_, call) ->
              call (min + binders)) terms)
              data)

(* Increment De Bruijn index of every free variable. *)
let lift (lang : language) (term : Type) (ev : language_evidence lang term)
: term -> term =
  lift' lang term ev 0

let rec repeat_map (t : Type) (n : int) (f : t -> 'a) (vs : repeat t n)
: repeat 'a n =
  match n with
  | 0 -> ()
  | _ -> let (x, rest) = vs in f x, repeat_map t (n-1) f rest

let rec type_remap (f : 'a -> Type) (f' : 'a -> Type)
  (trans : forall 'a. f 'a -> f' 'a)
  (vs : type_map f ls) : type_map f' ls =
  match ls with
  | [] -> ()
  | h :: t -> let (x, rest) = vs in trans x, type_remap f t f' trans rest
Generic Proofs

We have generic \textit{lift}.

\textit{lift} \( e = e \) with every free variable's De Bruijn index incremented

Assume we also have generic \textit{subst}:

\( \textit{subst} \ x \ e_1 \ e_2 = e_2 \) with \( e_1 \) substituted for free variable \( x \)

\textbf{Theorem} \textit{subst.lift_commute} :

\[ \forall e_1 \ e_2. \ \textit{lift} (\textit{subst} \ 0 \ e_1 \ e_2) = \textit{subst} \ 1 \ (\textit{lift} \ e_1) \ (\textit{lift} \ e_2) \]

Proof shouldn't depend in any deep way on specific language!

We can \textbf{prove this generically} if we force language evidence to include proofs of a theorem like this:

\[
\text{recursor branches (c vars terms data)} \ \\
= \ \text{branches.c} \\
\quad \text{vars} \\
\quad \text{(map (fun term -> (term, recursor branches term)) terms)} \\
\quad \text{data}
\]
Dependently-Typed ASTs

\begin{align*}
type \ ty &= \\
& | \ Int \\
& | \ Arrow \ of \ ty \times ty
\end{align*}

\begin{align*}
type \ (\Gamma, \tau) \ var &= … \\
(* \ Type \ of \ a \ variable \ of \ type \ \tau \ found \ within \ \Gamma \ *)
\end{align*}

\begin{align*}
type \ (\Gamma, \tau) \ term &= \\
& | \ Const : \textbf{forall} \ \Gamma. \ (\Gamma, \ Int) \ term \\
& | \ Var : \textbf{forall} \ \Gamma \tau. \ (\Gamma, \tau) \ var \rightarrow (\Gamma, \tau) \ term \\
& | \ App : \textbf{forall} \ \Gamma \tau_1 \tau_2. \ (\Gamma, \ Arrow \ (\tau_1, \tau_2)) \ term \rightarrow (\Gamma, \tau_1) \ term \rightarrow (\Gamma, \tau_2) \ term \\
& | \ Lam : \textbf{forall} \ \Gamma \tau_1 \tau_2. \ (\tau_1 :: \Gamma, \tau_2) \ term \rightarrow (\Gamma, \ Arrow \ (\tau_1, \tau_2)) \ term
\end{align*}

\begin{align*}
\textbf{let} \ ty\_denote : \ ty \rightarrow \textbf{Type} &= … \\
(* \ Denotational \ semantics \ of \ types \ *)
\end{align*}

\begin{align*}
\textbf{let} \ subst\_denote : \ ty \ list \rightarrow \textbf{Type} &= \text{type\_map} \ ty\_denote \\
(* \ Denotational \ semantics \ of \ contexts \ *)
\end{align*}

\begin{align*}
\textbf{let} \ term\_denote : \textbf{forall} \ \Gamma \tau. \ term \ \Gamma \tau \rightarrow \text{subst\_denote} \ \Gamma \rightarrow \ ty\_denote \ \tau &= … \\
(* \ Denotational \ semantics \ of \ terms \ *)
\end{align*}
Implemented in Lambda Tamer System

• Used in the construction of a certified type-preserving compiler from lambda calculus to assembly language [PLDI07]

• Flagship example: A certified CPS transformation for simply-typed lambda calculus in 250 LoC
Summary

1. Write **generic functions** that operate on **AST universes**. [entirely inside the type theory]

2. Write **generic proofs** about those functions. [entirely inside the type theory]

3. Then **reflect** individual language definitions into the AST universe type. [outside the type theory]

4. Construct **evidence** that your reflection is sound. [outside the type theory]

5. Start using the generic functions and theorems! [entirely inside the type theory]

Code and documentation on the web at:
http://ltamer.sourceforge.net/