Generic Programming and Proving for Programming Language Metatheory

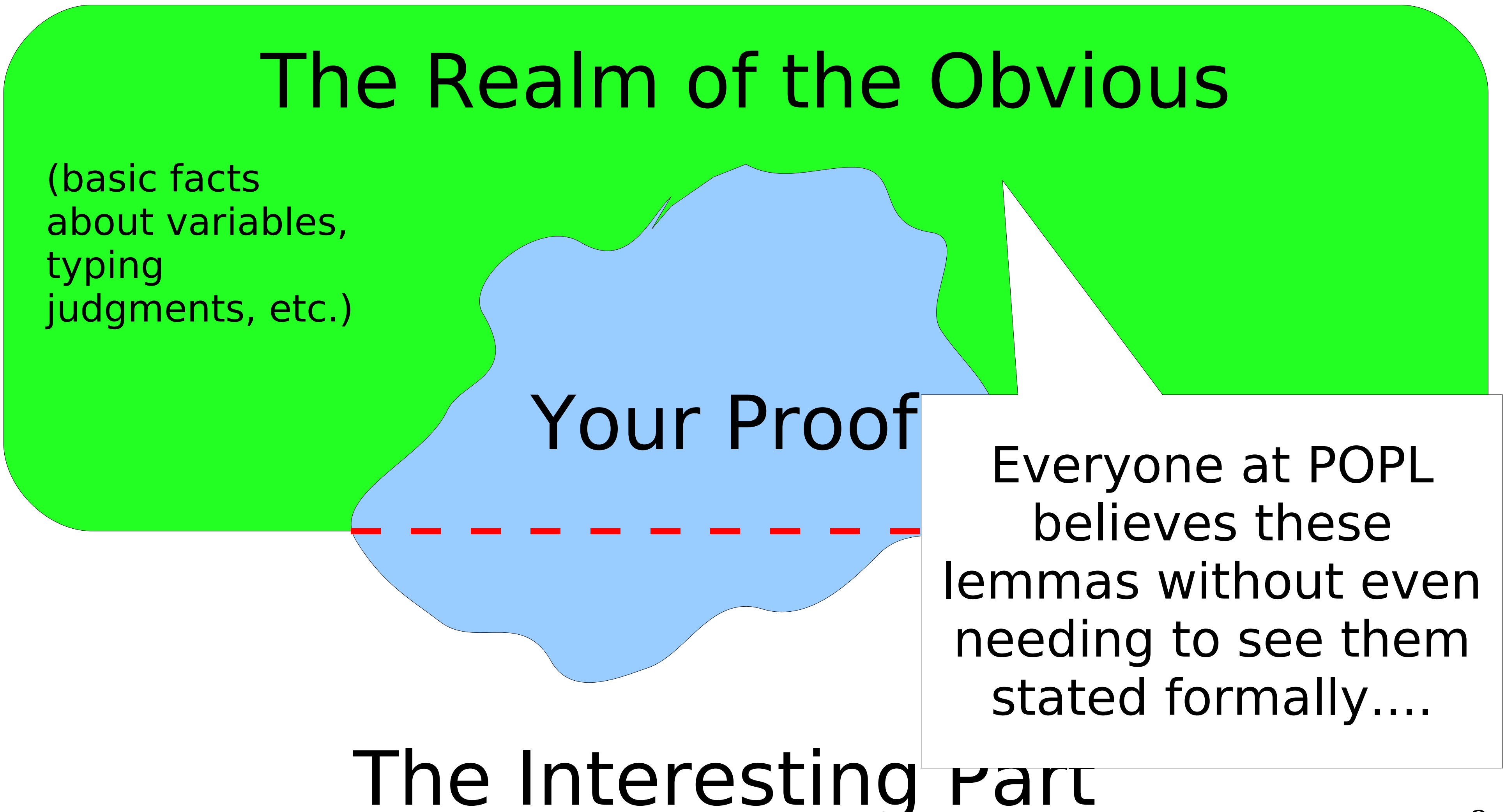
Adam Chlipala
WMM 2007

POPLmark 1A Solutions Using Coq

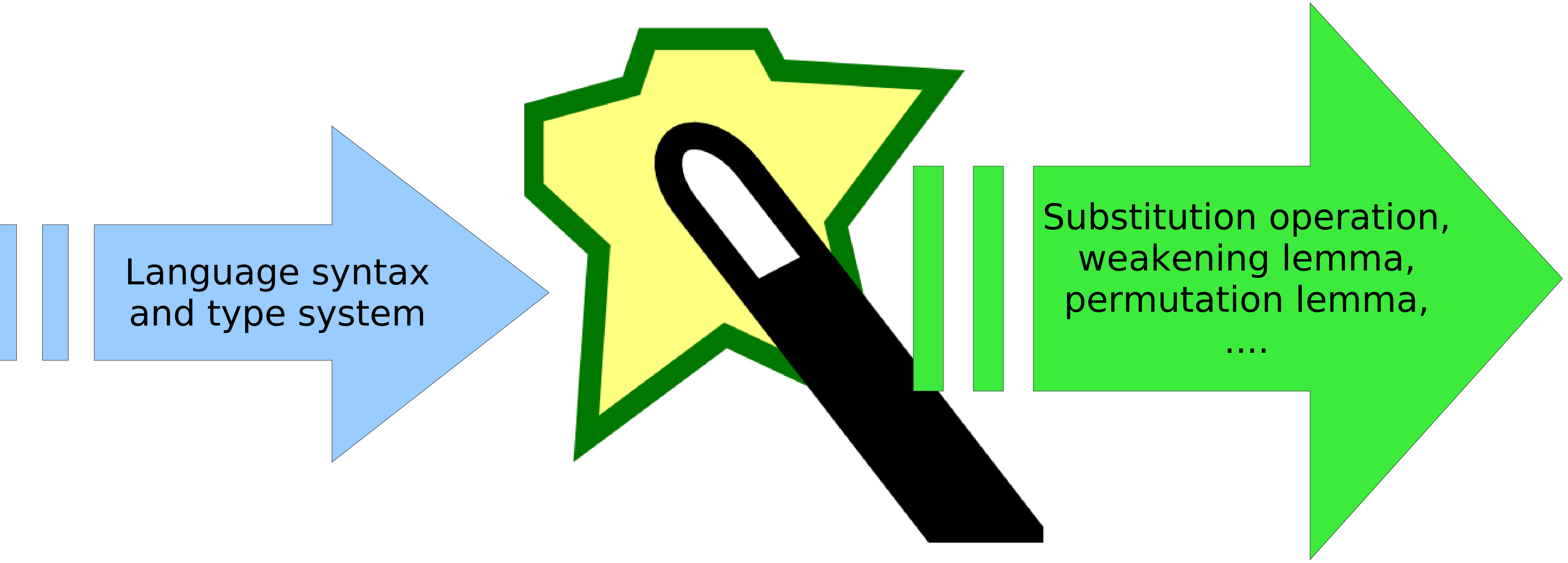
Representation	**Lemmas**	**Proof Steps**
de Bruijn	30	402
locally nameless	49	495
levels/names	56	938
de Bruijn (nested)	49	1574
locally nameless	23	75
locally nameless	22	101
on paper	***2***	***2 pages***

(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....
The Interesting Part

What We Really Want



Metatheory Wizard

But Isn't That Twelf?

Write It in OCaml?

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

Metatheory Wizard

Write It with Dependent Types?

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

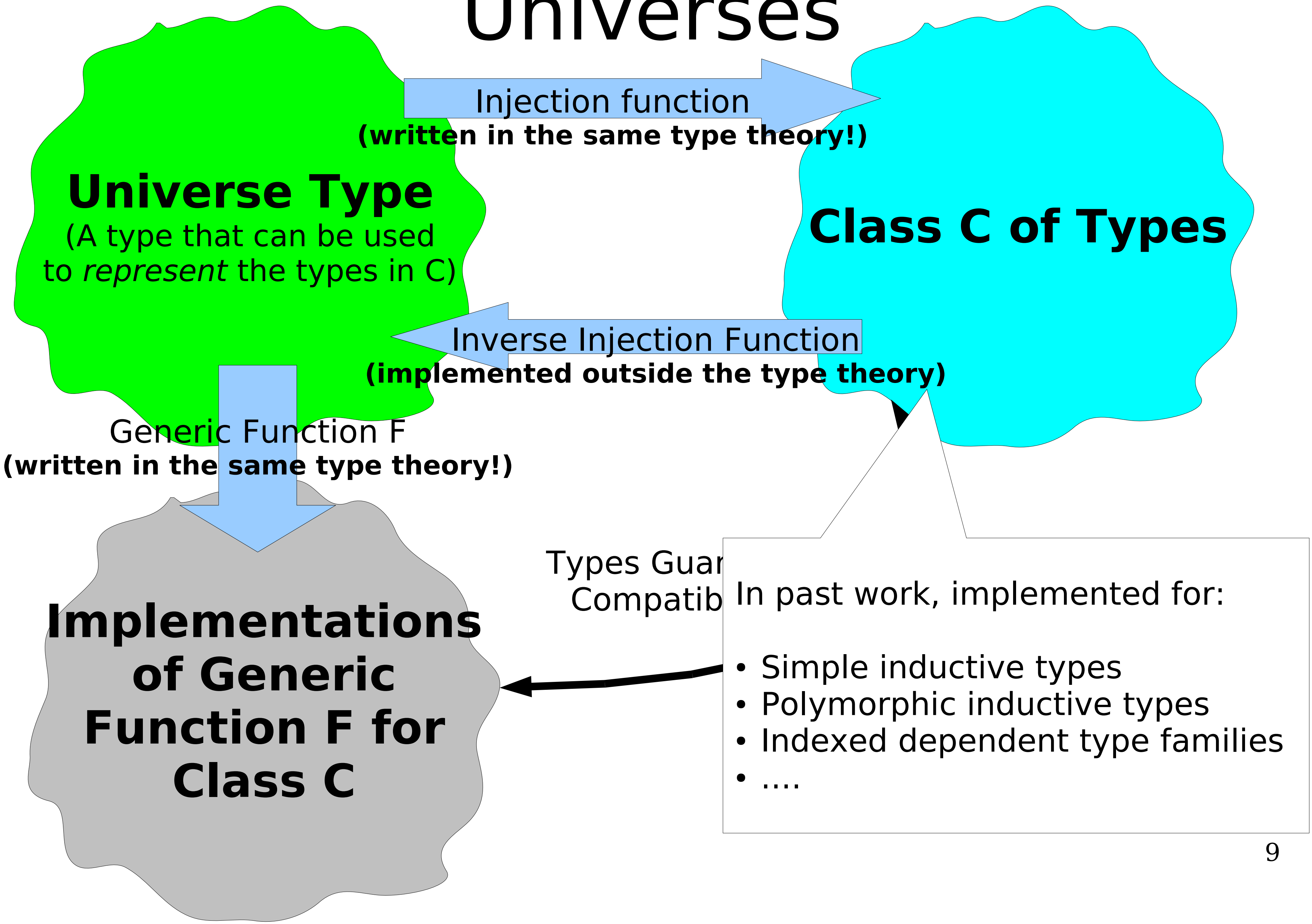
We can do this ***in Coq itself***!

Metatheory Wizard

Generic Programming with *Universes* in Type Theory

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

Universes


Injection function
(written in the same type theory!)
Universe Type
(A type that can be used
to *represent* the types in C)
Class C of Types
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 Guar
Compatib
In past work, implemented for:
• Simple inductive types
• Polymorphic inductive types
• Indexed dependent type families
• ....

Universes for ASTs

```
(* De Bruijn indices *)
type dbvar = int

(* Untyped lambda calculus terms *)
type term =                  → [const; var; app; lam]
    | Const of int           → const = {vars = 0; terms = [];     data = int  }
    | Var of dbvar           → var =   {vars = 1; terms = [];     data = unit}
    | App of term * term     → app =   {vars = 0; terms = [0; 0]; data = unit}
    | Lam of term            → lam =   {vars = 0; terms = [1];    data = unit}

(* Universe for AST constructors *)
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 *)
type language = constructor list
```

Evidence

```
type term =           → (const_in, (v
    | Const of int    → const_in
    | Var of dbvar    → var_in =
    | App of term * term → app_in =                          e1, e2)
    | Lam of term     → lam_in      fun _ (e, ()) _           e
```

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

```
(* What do we need to
let constructor_evi
    repeat dbvar con.vars
    -> type_map (fun _ -> term) con.terms
    -> con.data
    -> term

(* What do we n
let langua
    type_
```

term_rep : language_evidence lang term

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

Reflecting Recursion

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

```
let term_rec _ (const, (var, (app, (lam, ()))))) =
    let rec f : term -> result = 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 *)
    in f
```

```
    let rec f = 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), ()) ()
```

```
(* 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
```

The recursive value on each subterm…

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

Reflecting Recursion

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

```
term_rep : language_evidence lang term =
    (const_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 Gen ction

Loop over the constructors' evidence packages, using each to produce a pattern matching branch....

```
let lift'_var (min : int) (
    if x >= min then x

(* Helper function th  lifts within a r    of De Bruijn indices *)
let lift' (lang : langua  ) (term : Type)
    ((builders,   urse) : language_eviden
    : int -> term -> term =
    recurse term
        (type_remap (fun (con : cons
            (build : constructor_evid
            fu   ars terms data (
            b
```

Use each subterm's recursive function with min

by the

w

Keep the same uninterpreted data....

```
let rec type_remap (f : 'a ->    ) (   : '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)
```

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

Generic Proofs

We have generic *lift*.

lift e = e with every free variable's De Bruijn index incremented

Assume we also have generic *subst*:

subst x e_1 e_2 = e_2 with e_1 substituted for free variable x

Theorem subst_lift_commute :

$\forall$ e_1 e_2. *lift* (*subst* 0 e_1 e_2) = *subst* 1 (*lift* e_1) (*lift* e_2)

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

We can **prove this generically** if we force language evidence to include proofs of a theorem like this:

```
    recursor branches (c vars terms data)
=   branches.c
        vars
        (map (fun term -> (term, recursor branches term)) terms)
        data
```

Dependently-Typed ASTs

Prove theorems like this generically:

term_denote (lift e) (x, σ)
= term_denote e σ

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

type (Γ, τ) var = ...
(Type of a variable of type τ found within Γ *)*

type (Γ, τ) term =
  | Const : **forall** Γ. (Γ, Int) term
  | Var : **forall** Γ τ. (Γ, τ) var -> (Γ, τ) term
  | App : **forall** Γ τ_1 τ_2. (Γ, Arrow (τ_1, τ_2)) term -> (Γ, τ_1) term -> (Γ, τ_2) term
  | Lam : **forall** Γ τ_1 τ_2. (τ_1 :: Γ, τ_2) term -> (Γ, Arrow (τ_1, τ_2)) term

let ty_denote : ty -> **Type** = ...
(Denotational semantics of types *)*

let subst_denote : ty list -> **Type** = type_map ty_denote
(Denotational semantics of contexts *)*

let term_denote : **forall** Γ τ. term Γ τ -> subst_denote Γ -> ty_denote τ = ...
(Denotational semantics of terms *)*

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/`