Scrap your nameplate

James Cheney

University of Edinburgh

ICFP 2005

September 27, 2005

What is "nameplate"?

- Nameplate is boilerplate to do with names, binding, etc.
- A few examples (one from my own code, one from TAPL):

```
let rec apply_s s t =
  let h = apply_s s in
  match t with
    Name a -> Name a
  | Abs (a,e) -> Abs(a, h e)
  | App(c,es) -> App(c, List.map h es)
  | Susp(p,vs,x) -> (match lookup s x with
        Some tm -> apply_p p tm
      | None -> Susp(p,vs,x))
;;

let rec apply_s_g s g =
  let h1 = apply_s_g s in
  let h2 = apply_s_p s in
  match g with
```

```
  Gtrue -> Gtrue
| Gatomic(t) -> Gatomic(apply_s s t)
| Gand(g1,g2) -> Gand(h1 g1, h1 g2)
| Gor(g1,g2) -> Gor(h1 g1, h1 g2)
| Gforall(x,g) ->
    let x' = Var.rename x in
    Gforall(x', apply_s_g (join x (Susp(Perm.id,Univ,x'))) 
| Gnew(x,g) ->
    let x' = Var.rename x in
    Gnew(x,  apply_p_g (Perm.trans x x') g)
| Gexists(x,g) ->
    let x' = Var.rename x in
    Gexists(x', apply_s_g (join x (Susp(Perm.id,Univ,x')))
| Gimplies(d,g) -> Gimplies(h2 d, h1 g)
| Gfresh(t1,t2) -> Gfresh(apply_s s t1, apply_s s t2)
```

```
    | Gequals(t1,t2) -> Gequals(apply_s s t1, apply_s s t2)
    | Geunify(t1,t2) -> Geunify(apply_s s t1, apply_s s t2)
    | Gis(t1,t2) -> Gis(apply_s s t1, apply_s s t2)
    | Gcut -> Gcut
    | Guard (g1,g2,g3) -> Guard(h1 g1, h1 g2, h1 g3)
    | Gnot(g) -> Gnot(h1 g)

and apply_s_p s p =
   let h1 = apply_s_g s in
   let h2 = apply_s_p s in
   match p with
     Dtrue -> Dtrue
   | Datomic(t) -> Datomic(apply_s s t)
   | Dimplies(g,t) -> Dimplies(h1 g, h2 t)
   | Dforall (x,p) ->
```

```
      let x’ = Var.rename x in
      Dforall (x’, apply_s_p (join x (Susp(Perm.id,Univ,x’)))
  | Dand(p1,p2) -> Dand(h2 p1,h2 p2)
  | Dnew(a,p) ->
      let a’ = Var.rename a in
      Dnew(a,  apply_p_p (Perm.trans a a’) p)
;;
```

```
let tymap onvar c tyT =
  let rec walk c tyT = match tyT with
    TyId(b) as tyT -> tyT
  | TyVar(x,n) -> onvar c x n
  | TyArr(tyT1,tyT2) -> TyArr(walk c tyT1,walk c tyT2)
  | TyBool -> TyBool
  | TyTop -> TyTop
  | TyBot -> TyBot
  | TyRecord(fieldtys) -> TyRecord(List.map (fun (li,tyTi) -
  | TyVariant(fieldtys) -> TyVariant(List.map (fun (li,tyTi)
  | TyFloat -> TyFloat
  | TyString -> TyString
  | TyUnit -> TyUnit
  | TyAll(tyX,tyT1,tyT2) -> TyAll(tyX,walk c tyT1,walk (c+1)
  | TyNat -> TyNat
```

```
  | TySome(tyX,tyT1,tyT2) -> TySome(tyX,walk c tyT1,walk (c+
  | TyAbs(tyX,knK1,tyT2) -> TyAbs(tyX,knK1,walk (c+1) tyT2)
  | TyApp(tyT1,tyT2) -> TyApp(walk c tyT1,walk c tyT2)
  | TyRef(tyT1) -> TyRef(walk c tyT1)
  | TySource(tyT1) -> TySource(walk c tyT1)
  | TySink(tyT1) -> TySink(walk c tyT1)
  in walk c tyT

let tmmap onvar ontype c t =
  let rec walk c t = match t with
    TmVar(fi,x,n) -> onvar fi c x n
  | TmAbs(fi,x,tyT1,t2) -> TmAbs(fi,x,ontype c tyT1,walk (c+
  | TmApp(fi,t1,t2) -> TmApp(fi,walk c t1,walk c t2)
  | TmTrue(fi) as t -> t
  | TmFalse(fi) as t -> t
```

```
| TmIf(fi,t1,t2,t3) -> TmIf(fi,walk c t1,walk c t2,walk c
| TmProj(fi,t1,l) -> TmProj(fi,walk c t1,l)
| TmRecord(fi,fields) -> TmRecord(fi,List.map (fun (li,ti)
                                              (li,walk c ti
                                  fields)
| TmLet(fi,x,t1,t2) -> TmLet(fi,x,walk c t1,walk (c+1) t2)
| TmFloat _ as t -> t
| TmTimesfloat(fi,t1,t2) -> TmTimesfloat(fi, walk c t1, wa
| TmAscribe(fi,t1,tyT1) -> TmAscribe(fi,walk c t1,ontype c
| TmInert(fi,tyT) -> TmInert(fi,ontype c tyT)
| TmFix(fi,t1) -> TmFix(fi,walk c t1)
| TmTag(fi,l,t1,tyT) -> TmTag(fi, l, walk c t1, ontype c t
| TmCase(fi,t,cases) ->
    TmCase(fi, walk c t,
           List.map (fun (li,(xi,ti)) -> (li, (xi,walk (c+
```

```
              cases)
  | TmString _ as t -> t
  | TmUnit(fi) as t -> t
  | TmLoc(fi,l) as t -> t
  | TmRef(fi,t1) -> TmRef(fi,walk c t1)
  | TmDeref(fi,t1) -> TmDeref(fi,walk c t1)
  | TmAssign(fi,t1,t2) -> TmAssign(fi,walk c t1,walk c t2)
  | TmError(_) as t -> t
  | TmTry(fi,t1,t2) -> TmTry(fi,walk c t1,walk c t2)
  | TmTAbs(fi,tyX,tyT1,t2) ->
      TmTAbs(fi,tyX,ontype c tyT1,walk (c+1) t2)
  | TmTApp(fi,t1,tyT2) -> TmTApp(fi,walk c t1,ontype c tyT2)
  | TmZero(fi)      -> TmZero(fi)
  | TmSucc(fi,t1)   -> TmSucc(fi, walk c t1)
  | TmPred(fi,t1)   -> TmPred(fi, walk c t1)
```

```
  | TmIsZero(fi,t1) -> TmIsZero(fi, walk c t1)
  | TmPack(fi,tyT1,t2,tyT3) ->
      TmPack(fi,ontype c tyT1,walk c t2,ontype c tyT3)
  | TmUnpack(fi,tyX,x,t1,t2) ->
      TmUnpack(fi,tyX,x,walk c t1,walk (c+2) t2)
  in walk c t

let typeShiftAbove d c tyT =
  tymap
    (fun c x n -> if x>=c then TyVar(x+d,n+d) else TyVar(x,n
    c tyT

let termShiftAbove d c t =
  tmmap
    (fun fi c x n -> if x>=c then TmVar(fi,x+d,n+d)
```

```
                     else TmVar(fi,x,n+d))
       (typeShiftAbove d)
       c t

let termShift d t = termShiftAbove d 0 t

let typeShift d tyT = typeShiftAbove d 0 tyT

let bindingshift d bind =
  match bind with
    NameBind -> NameBind
  | TyVarBind(tyS) -> TyVarBind(typeShift d tyS)
  | VarBind(tyT) -> VarBind(typeShift d tyT)
  | TyAbbBind(tyT,opt) -> TyAbbBind(typeShift d tyT,opt)
  | TmAbbBind(t,tyT_opt) ->
```

```
    let tyT_opt' = match tyT_opt with
                     None->None
                   | Some(tyT) -> Some(typeShift d tyT) in
    TmAbbBind(termShift d t, tyT_opt')

(* -----------------------------------------------
(* Substitution *)

let termSubst j s t =
  tmmap
    (fun fi j x n -> if x=j then termShift j s else TmVar(fi
    (fun j tyT -> tyT)
    j t

let termSubstTop s t =
```

```
  termShift (-1) (termSubst 0 (termShift 1 s) t)

let typeSubst tyS j tyT =
  tymap
    (fun j x n -> if x=j then (typeShift j tyS) else (TyVar(
    j tyT

let typeSubstTop tyS tyT =
  typeShift (-1) (typeSubst (typeShift 1 tyS) 0 tyT)

let rec tytermSubst tyS j t =
  tmmap (fun fi c x n -> TmVar(fi,x,n))
        (fun j tyT -> typeSubst tyS j tyT) j t

let tytermSubstTop tyS t =
```

```
termShift (-1) (tytermSubst (typeShift 1 tyS) 0 t)
```

Why scrap it?

- I'm tired of writing nameplate such as α-equivalence, capture-avoiding substitution and free variables functions.
- Aren't you?
- It's boring! I have better uses for my time!
- There's nothing hard about these tasks, but need to redo for each new datatype
- de Bruijn encodings: require changing/translating from "natural" abstract syntax
- HOAS: Provides CAS for free, but hard to integrate with functional programming (active research topic)
- FreshML: Supports α-equivalence, but CAS has to be written explicitly.

Is there another way?

- Using the Gabbay-Pitts/FreshML approach (which I refer to as *nominal abstract syntax*), substitution and FVs are much better behaved.
- Starting point: much of the functionality of FreshML can be provided within Haskell using a class library (folklore)
- Use Lämmel-Peyton Jones “scrap your boilerplate” style of generic programming to provide instances automatically (including substitution, FVs)
- Claim: Users can use it without having to understand how it works.

The real problem

- For syntax trees without binding, substitution and FVs are essentially "fold", most of whose cases are boring.

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Exp} & = & \mathit{Var}\ \mathit{Name} \mid \mathit{Plus}\ \mathit{Exp}\ \mathit{Exp} \mid \ldots \\
\mathit{subst}\ a\ t\ (\mathit{Var}\ b) \mid a \equiv b & = & t \\
\mathit{subst}\ a\ t\ (\mathit{Var}\ b) \mid \mathit{otherwise} & = & \mathit{Var}\ b \\
\mathit{subst}\ a\ t\ (\mathit{Plus}\ \mathit{e1}\ \mathit{e2}) & = & \mathit{Plus}\ (\mathit{subst}\ a\ t\ \mathit{e1}) \\
 & & \qquad (\mathit{subst}\ a\ t\ \mathit{e2})
\end{array}
$$

- These functions are prime examples of "generic traversals" and "generic queries" of the *scrap your boilerplate* generic programming [Peyton Jones and Lämmel 2003,2004,2005]

- Thus, prime candidates for boilerplate-scrapping

The real problem

- As soon as we add binding syntax, this nice structure disappears!

$$
\begin{array}{ll}
\mathbf{data}\ \mathit{Exp} & = \mathit{Var}\ \mathit{Name} \mid \mathit{Lam}\ \mathit{Name}\ \mathit{Exp} \mid \ldots \\
\multicolumn{2}{l}{\mathbf{instance}\ \mathit{Monad}\ M\ \mathbf{where}\ \ldots} \\
\multicolumn{2}{l}{\mathit{fresh} :: M\ \mathit{Name}} \\
\mathit{rename} & :: \mathit{Name} \to \mathit{Name} \to \mathit{Exp} \to M\ \mathit{Exp} \\
\mathit{subst} & :: \mathit{Name} \to \mathit{Exp} \to \mathit{Exp} \to M\ \mathit{Exp} \\
\mathit{subst}\ a\ t\ (\mathit{Var}\ b) \mid a \equiv b & = \mathit{return}\ t \\
\mathit{subst}\ a\ t\ (\mathit{Var}\ b) & = \mathit{return}\ (\mathit{Var}\ b) \\
\mathit{subst}\ a\ t\ (\mathit{Lam}\ b\ e) & = \mathbf{do}\ b' \leftarrow \mathit{fresh} \\
& \qquad e' \leftarrow \mathit{rename}\ b\ b'\ e \\
& \qquad e'' \leftarrow \mathit{subst}\ a\ t\ e' \\
& \qquad \mathit{return}\ (\mathit{Lam}\ b'\ e'')
\end{array}
$$

The real problem

- As soon as we add binding syntax, this nice structure disappears!
- Because
  - We need to know how to safely rename bound names to fresh ones
  - That means we need side-effects to generate fresh names
  - and need to know which names are bound
- This makes CAS much trickier to implement generically.
- And things get even worse when there are multiple datatypes involved, each with variables (e.g., types, terms, kinds).

Our approach

- First, observe that we can factor the code as follows:

$$
\begin{array}{ll}
\textbf{data}\ a \setminus\!\!\setminus\!\!\setminus t & = a \setminus\!\!\setminus\!\!\setminus t \\
\textbf{data}\ \mathit{Exp} & = \mathit{Var}\ \mathit{Name} \mid \mathit{Lam}\ (\mathit{Name} \setminus\!\!\setminus\!\!\setminus \mathit{Exp}) \mid \ldots \\
\mathit{subst\_abs}\ a\ t\ (b \setminus\!\!\setminus\!\!\setminus e) & = \textbf{do}\ b' \leftarrow \mathit{fresh} \\
 & \qquad\ \ e' \leftarrow \mathit{rename}\ b\ b'\ e \\
 & \qquad\ \ e'' \leftarrow \mathit{subst}\ a\ t\ e' \\
 & \qquad\ \ \mathit{return}\ (b' \setminus\!\!\setminus\!\!\setminus e'') \\
\mathit{subst}\ a\ t\ (\mathit{Lam}\ b\ e) & = \textbf{do}\ e' \leftarrow \mathit{subst\_abs}\ a\ t\ e \\
 & \qquad\ \mathit{return}\ (\mathit{Lam}\ e')
\end{array}
$$

- Note: we do the *same work* as the naive version, but the cases involving name-binding are handled by an "abstraction" type constructor and written *once and for all*.

Our approach (2)

- Next, let's use a pure function $swap$ instead of $rename$.

$$
\begin{aligned}
&\mathbf{data}\ a \setminus\!\!\setminus\!\!\setminus t && = a \setminus\!\!\setminus\!\!\setminus t \\
&\mathbf{data}\ Exp && = Var\ name \mid Lam\ (Name \setminus\!\!\setminus\!\!\setminus Exp) \mid \ldots \\
&swap && :: Name \rightarrow Name \rightarrow Exp \rightarrow Exp \\
&subst\_abs\ a\ t\ (b \setminus\!\!\setminus\!\!\setminus e) && = \mathbf{do}\ b' \leftarrow fresh \\
& && \phantom{= \mathbf{do}\ } e' \leftarrow subst\ a\ t\ (swap\ b\ b'\ e) \\
& && \phantom{= \mathbf{do}\ } return\ (b' \setminus\!\!\setminus\!\!\setminus e') \\
&subst\ a\ t\ (Lam\ b\ e) && = \mathbf{do}\ e' \leftarrow subst\_abs\ a\ t\ e \\
& && \phantom{= \mathbf{do}\ } return\ (Lam\ e')
\end{aligned}
$$

- We'll see why this is important later.

- (Basically, it's because $swap$ is pure, easy to define and "naturally" capture avoiding.)

Our approach (3)

- Next, note that we can parameterize the substitution functions by an monad m that provides a fresh name generator:

$$\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{FreshM}\ m\ \mathbf{where}$$
$$\quad \mathit{fresh} :: m\ \mathit{Name}$$
$$\mathit{subst\_abs} :: \mathit{FreshM}\ m \Rightarrow$$
$$\quad \mathit{Name} \to \mathit{Exp} \to \mathit{Name} \lll \mathit{Exp} \to m\ (\mathit{Name} \lll \mathit{Exp})$$
$$\mathit{subst} :: \mathit{FreshM}\ m \Rightarrow \mathit{Name} \to \mathit{Exp} \to \mathit{Exp} \to m\ \mathit{Exp}$$

Our approach (4)

- Next, observe that we can make both substitution functions instances of a type class:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Subst}\ t\ u\ \mathbf{where} \\
\quad \mathit{subst} :: \mathit{FreshM}\ m \Rightarrow \mathit{Name} \to t \to u \to m\ u \\
\mathbf{instance}\ \mathit{Subst}\ \mathit{Exp}\ (\mathit{Name} \mathbin{\backslash\!\!\backslash\!\!\backslash} \mathit{Exp})\ \mathbf{where} \\
\quad \mathit{subst}\ a\ t\ (b \mathbin{\backslash\!\!\backslash\!\!\backslash} e) \qquad = \mathbf{do}\ b' \leftarrow \mathit{fresh} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad e' \leftarrow \mathit{subst}\ a\ t\ (\mathit{swap}\ b\ b'\ e) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (b' \mathbin{\backslash\!\!\backslash\!\!\backslash} e') \\
\mathbf{instance}\ \mathit{Subst}\ \mathit{Exp}\ \mathit{Exp} \\
\quad \mathit{subst}\ a\ t\ (\mathit{Lam}\ b\ e) = \mathbf{do}\ e' \leftarrow \mathit{subst}\ a\ t\ e \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{Lam}\ e') \\
\quad \ldots
\end{array}
$$

Story so far

- So far, I've suggested how nameplate can be *reorganized*, but not yet *scrapped*.
- E.g., using a type class for $Subst$ and a monad for name-generation.
- Next step: provide a *library* with appropriate type classes and instances for common situations
- Key issue: defining *renaming* at all types.
- We use a FreshML-like approach based on swapping as the primitive renaming operation.
- I'll describe $FreshLib$: a library that provides much of the functionality of FreshML as a Haskell class library

FreshLib

- Types $Name$, $Name \mathbin{\backslash\!\backslash\!\backslash} a$: represent names, name-abstractions.
- Class Nom: provides swapping ($swap$), freshness ($fresh$), α-equivalence (aeq)
- Class $Subst$, $FreeVars$: provide substitution $subst$ and free variable fvs functions for types that "have variables"
- Class $HasVar$: says what case of user-defined type acts as variable of that type.
- Class $BType$: provides enough information to use a type as a binder

Getting started

- To *use* $\mathit{FreshLib}$, you just write **data** declarations, empty Nom instances, and HasVar declarations.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Lam} = \mathit{Var}\ \mathit{Name} \\
&\qquad\qquad\ \mid\ \mathit{App}\ \mathit{Lam}\ \mathit{Lam} \\
&\qquad\qquad\ \mid\ \mathit{Lam}\ (\mathit{Name} \mathbin{\backslash\!\!\backslash\!\!\backslash} \mathit{Lam}) \\
&\mathbf{instance}\ \mathit{Nom}\ \mathit{Lam}\ \mathbf{where} \\
&\qquad \text{-- empty} \\
&\mathbf{instance}\ \mathit{HasVar}\ \mathit{Lam}\ \mathbf{where} \\
&\quad \mathit{is\_var}\ (\mathit{Var}\ x) = \mathit{Just}\ x \\
&\quad \mathit{is\_var}\ \_ \qquad\quad\ \ = \mathit{Nothing}
\end{aligned}
$$

- swap, fresh, aeq, subst, fvs are derived automatically.

Nominal types

- Type class Nom

$$
\begin{aligned}
&\textbf{class } Nom\ a \textbf{ where}\\
&\quad swap :: Name \to Name \to a \to a\\
&\quad fresh :: Name \to a \to Bool\\
&\quad aeq :: a \to a \to Bool
\end{aligned}
$$

- $swap\ a\ b\ x$: exchanges (all occurrences of) two names a, b in x
- $fresh\ a\ x$: tests whether a is “fresh for” (not free in) x
- $aeq\ x\ y$: tests alpha-equivalence of x and y

Instances of Nom

- Int and other base types

$$
\begin{array}{l}
\textbf{instance } Nom\ Int \textbf{ where} \\
\quad swap\ a\ b\ x = x \\
\quad fresh\ a\ x \quad = True \\
\quad aeq\ x\ y \qquad = x \equiv y
\end{array}
$$

- Pairs

$$
\begin{array}{l}
\textbf{instance } (Nom\ a, Nom\ b) \Rightarrow Nom\ (a, b) \textbf{ where} \\
\quad swap\ a\ b\ (x, y) \quad = (swap\ a\ b\ x, swap\ a\ b\ y) \\
\quad fresh\ a\ (x, y) \qquad = fresh\ a\ x \wedge fresh\ a\ y \\
\quad aeq\ (x, y)\ (x', y') = aeq\ x\ x' \wedge aeq\ y\ y'
\end{array}
$$

Instances of Nom

- $Name$: where the rubber meets the road

$$
\begin{array}{l}
\textbf{instance } Nom\ Name \textbf{ where} \\
\quad swap\ a\ b\ c = \textbf{if } a \equiv c \textbf{ then } b \\
\quad\qquad\qquad\quad\ \ \textbf{else if } b \equiv c \textbf{ then } a \textbf{ else } c \\
\quad fresh\ a\ b \quad = a \not\equiv b \\
\quad aeq\ a\ b \qquad = a \equiv b
\end{array}
$$

- Abstractions

$$
\begin{array}{l}
\textbf{instance } (Nom\ a) \Rightarrow Nom\ (Name \setminus\!\!\setminus\!\!\setminus a) \textbf{ where} \\
\quad swap\ a\ b\ (c \setminus\!\!\setminus\!\!\setminus x) \quad = (swap\ a\ b\ c) \setminus\!\!\setminus\!\!\setminus (swap\ a\ b\ x) \\
\quad fresh\ a\ (b \setminus\!\!\setminus\!\!\setminus x) \qquad = a \equiv b \vee fresh\ a\ x \\
\quad aeq\ (a \setminus\!\!\setminus\!\!\setminus x)\ (b \setminus\!\!\setminus\!\!\setminus y) = (a \equiv b \wedge aeq\ x\ y) \\
\qquad\qquad\qquad\qquad\qquad\quad \vee\ (fresh\ a\ y \wedge aeq\ x\ (swap\ a\ b\ y))
\end{array}
$$

Class Subst

- For ordinary types, substitutions ignore structure.

$$\begin{array}{l}\textbf{instance}\ (\mathit{Subst}\ t\ a, \mathit{Subst}\ t\ b) \Rightarrow \mathit{Subst}\ t\ (a, b)\ \textbf{where} \\ \quad \mathit{subst}\ a\ t\ (x, y) = \textbf{do}\ \begin{array}[t]{l} x' \leftarrow \mathit{subst}\ a\ t\ x \\ y' \leftarrow \mathit{subst}\ a\ t\ y \\ \mathit{return}\ (x', y') \end{array}\end{array}$$

- For abstractions, substitutions rename bound names, then proceed

$$\begin{array}{l}\textbf{instance}\ \mathit{Subst}\ t\ a \Rightarrow \mathit{Subst}\ t\ (\mathit{Name} \mathbin{\backslash\!\!\backslash\!\!\backslash} a)\ \textbf{where} \\ \quad \mathit{subst}\ a\ t\ (b \mathbin{\backslash\!\!\backslash\!\!\backslash} x) = \textbf{do}\ \begin{array}[t]{l} b' \leftarrow \mathit{fresh} \\ x' \leftarrow \mathit{subst}\ a\ t\ (\mathit{swap}\ b\ b'\ x) \\ \mathit{return}\ (b' \mathbin{\backslash\!\!\backslash\!\!\backslash} x') \end{array}\end{array}$$

Class *FreeVars*

- For ordinary types, *fvs* is union of *fvs* of components.

$$\textbf{instance}\ (\mathit{FreeVars}\ t\ a, \mathit{FreeVars}\ t\ b) \Rightarrow \mathit{FreeVars}\ t\ (a, b)$$

$$\textbf{where}$$

$$\mathit{FreeVars}\ t\ (x, y) = \mathit{union}\ (\mathit{fvs}\ t\ x)\ (\mathit{fvs}\ t\ y)$$

- For abstractions, remove bound name from set

$$\textbf{instance}\ \mathit{FreeVars}\ t\ a \Rightarrow \mathit{FreeVars}\ t\ (\mathit{Name} \mathbin{\backslash\!\!\backslash\!\!\backslash} a)\ \textbf{where}$$

$$\mathit{fvs}\ t\ (b \mathbin{\backslash\!\!\backslash\!\!\backslash} x) = \mathit{fvs}\ t\ x \mathbin{\backslash\backslash} [b]$$

Making it generic

- Using generic programming extensions to GHC, the instances of Nom, $Subst$, and $FreeVars$ can be provided automatically
- for user-defined data types using $Name$ and $\cdot \mathbin{\backslash\!\backslash\!\backslash} \cdot$
- Key ingredient #1: derivable (or generic) type classes (Hinze & Peyton Jones 2000) used to implement default form of Nom
- Key ingredient #2: scrap your boilerplate with class (Lämmel & Peyton Jones 2005) used to implement default cases of $Subst$, $FreeVars$
- Crucial fact: behavior for $Name$, $Name \mathbin{\backslash\!\backslash\!\backslash} \cdot$, other special types can be provided using specialized type class instances.
- This makes $FreshLib$ much more extensible/customizable.

The tricky part

- The tricky part is that *subst* and *fvs* are almost but not quite structure-driven
- All cases except Var are structural recursion.
- Want to avoid having to write per-datatype instances:

$$
\begin{array}{l}
\textbf{instance } \mathit{Subst}\ \mathit{Lam}\ \mathit{Lam}\ \textbf{where} \\
\quad \mathit{subst}\ a\ t\ (\mathit{Var}\ b)\ |\ a \equiv b = \mathit{return}\ t \\
\quad \mathit{subst}\ a\ t\ (\mathit{Var}\ b) \qquad\qquad = \mathit{return}\ (\mathit{Var}\ b) \\
\quad \mathit{subst}\ a\ t\ (\mathit{App}\ t1\ t2) \qquad = \ldots \\
\quad \mathit{subst}\ a\ t\ (\mathit{Lam}\ abs) \qquad\quad = \ldots \\
\quad \ldots
\end{array}
$$

Solution

- Need a way to tell the library what constructors represent variables
- This is what $HasVar$ is for.
- Generic definition of $Subst$ looks like[a]

$$\begin{array}{l}\textbf{instance } Subst\ t\ a \textbf{ where}\\ \quad subst\ a\ t\ x = gmapM\ (subst\ a\ t)\ x\\ \textbf{instance } HasVar\ a \Rightarrow Subst\ a\ a \textbf{ where}\\ \quad subst\ a\ t\ x = \textbf{if } is\_var\ x \equiv Just\ a\\ \qquad\qquad\qquad\quad \textbf{then } return\ t\\ \qquad\qquad\qquad\quad \textbf{else } gmapM\ (subst\ a\ t)\ x\end{array}$$

- using SYB library's $gmapM$ combinator.

[a] The real code is a little scary.

Extensions/future work

- Multiple name types
  - With a single name type, bindings can “interfere”
- Alternative (efficient) implementations, such as
  - de Bruijn
  - HOAS
  - other efficient λ-term representations?
- Lightweight language extensions (e.g. Cαml): more flexible?

Conclusion

- We have shown how to provide most of the functionality of FreshML as a Haskell class library *FreshLib*
- We have also shown how to use generic programming techniques to provide additional capabilities
- such as capture-avoiding substitution and FVs "for free"
- No claim of efficiency, but should at least be useful for prototyping/teaching