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'))) g)
| 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'))) g)
| 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) -> (li,walk tyTi)) fieldtys)
        | TyVariant(fieldtys) -> TyVariant(List.map (fun (li,tyTi) -> (li,walk tyTi)) fieldtys)
        | TyFloat -> TyFloat
        | TyString -> TyString
        | TyUnit -> TyUnit
        | TyAll(tyX,tyT1,tyT2) -> TyAll(tyX,walk c tyT1,walk (c+1) tyT2)
        | TyNat -> TyNat
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+1)t2)
  | 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 t3)
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,walk c t2)
TmAscribe(fi,t1,tyT1) -> TmAscribe(fi,walk c t1,ontype c tyT1)
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 tyT)
TmCase(fi,t,cases) ->
    TmCase(fi, walk c t, List.map (fun (li,(xi,ti)) -> (li, (xi,walk (c+1) ti))) cases)
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+d))
c tyT

let termShiftAbove d c t =
  tmmmap
  (fun fi c x n -> if x>=c then TmVar(fi, x+d, n+d) else TmVar(fi, x, 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 n))
      (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(x,n))
    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 $\alpha$-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 $\alpha$-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.

\[
\text{data } \text{Exp} \quad = \quad \text{Var Name} \mid \text{Plus Exp Exp} \mid \ldots
\]

\[
\text{subst } a \ t \ (\text{Var } b) \mid a \equiv b \quad = \quad t
\]

\[
\text{subst } a \ t \ (\text{Var } b) \mid \text{otherwise} \quad = \quad \text{Var } b
\]

\[
\text{subst } a \ t \ (\text{Plus } e1 \ e2) \quad = \quad \text{Plus} \ (\text{subst } a \ t \ e1) \ (\text{subst } a \ t \ e2)
\]

- 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-scraping
The real problem

- As soon as we add binding syntax, this nice structure disappears!
data $Exp \quad = \quad Var \ Name \ | \ Lam \ Name \ Exp \ | \ ...

instance Monad $M$ where ...

fresh :: $M$ Name

rename :: Name $\rightarrow$ Name $\rightarrow$ Exp $\rightarrow$ $M$ Exp

subst :: Name $\rightarrow$ Exp $\rightarrow$ Exp $\rightarrow$ $M$ Exp

subst a t (Var b) $| a \equiv b = return \ t$

subst a t (Var b) $= return (Var b)$

subst a t (Lam b e) $= do \ b' \leftarrow$ fresh

\[ e' \leftarrow$ rename b b' e \]
\[ e'' \leftarrow subst a t \ e' \]
\[ return (Lam b' e'') \]
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:

\[
\text{data } a \parallel t = a \parallel t
\]

\[
\text{data } \text{Exp} = \text{Var Name} \mid \text{Lam} (\text{Name} \parallel \text{Exp}) \mid \ldots
\]

\[
\text{subst_abs } a \ t \ (b \parallel e) = \text{do } b' \leftarrow \text{fresh}
\]

\[
e' \leftarrow \text{rename } b \ b' \ e
\]

\[
e'' \leftarrow \text{subst } a \ t \ e'
\]

\[
\text{return } (b' \parallel e'')
\]

\[
\text{subst } a \ t \ (\text{Lam } b \ e) = \text{do } e' \leftarrow \text{subst_abs } a \ t \ e
\]

\[
\text{return } (\text{Lam } e')
\]

- 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`.
  
  \[
  \text{data} \ a \parallel t = a \parallel t
  \]
  
  \[
  \text{data} \ Exp = \text{Var name} | \text{Lam} \ (Name \parallel Exp) | ... \]
  
  \[
  \text{swap} :: Name \to Name \to Exp \to Exp
  \]
  
  \[
  \text{subst_abs} a t (b \parallel e) = \begin{cases} 
  b' \leftarrow \text{fresh} \\
  e' \leftarrow \text{subst} a t (\text{swap} b b' e) \\
  \text{return} (b' \parallel e')
  \end{cases}
  \]
  
  \[
  \text{subst} a t (\text{Lam} b e) = \begin{cases} 
  e' \leftarrow \text{subst_abs} a t e
  \text{return} (\text{Lam} e')
  \end{cases}
  \]
  
  • 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:

  ```haskell
  class Monad m => FreshM m where
    fresh :: m Name
    subst_abs :: FreshM m =>
      Name -> Exp -> Name \ Exp -> m (Name \ Exp)
    subst :: FreshM m => Name -> Exp -> Exp -> m Exp
  ```
Our approach (4)

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

```haskell
class Subst t u where
    subst :: FreshM m ⇒ Name → t → u → m u

instance Subst Exp (Name \| Exp) where
    subst a t (b \| e) = do b' ← fresh
        e' ← subst a t (swap b b' e)
    return (b' \| e')

instance Subst Exp Exp
    subst a t (Lam b e) = do e' ← subst a t e
        return (Lam e')
```

...
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 $\text{Name}$, $\text{Name} \downarrow a$: represent names, name-abstractions.
- Class $\text{Nom}$: provides swapping ($\text{swap}$), freshness ($\text{fresh}$), $\alpha$-equivalence ($\text{aeq}$)
- Class $\text{Subst}, \text{FreeVars}$: provide substitution $\text{sub}_st$ and free variable $\text{fvs}$ functions for types that “have variables”
- Class $\text{HasVar}$: says what case of user-defined type acts as variable of that type.
- Class $\text{BType}$: provides enough information to use a type as a binder
Getting started

- To use FreshLib, you just write **data** declarations, empty **Nom** instances, and **HasVar** declarations.

```haskell
data Lam = Var Name
          | App Lam Lam
          | Lam (Name \(\parallel\) Lam)

instance Nom Lam where
  -- empty

instance HasVar Lam where
  is_var (Var x) = Just x
  is_var _ = Nothing
```

- *swap, fresh, aeq, subst, fvs* are derived automatically.
Nominal types

- Type class *Nom*

  ```haskell
class Nom a where

  swap :: Name → Name → a → a
  fresh :: Name → a → Bool
  aeq :: a → a → Bool
  ```

- *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

  ```haskell
  instance Nom Int where
    swap a b x = x
    fresh a x = True
    aeq x y = x ≡ y
  ```

- Pairs

  ```haskell
  instance (Nom a, Nom b) ⇒ Nom (a, b) where
    swap a b (x, y) = (swap a b x, swap a b y)
    fresh a (x, y) = fresh a x ∧ fresh a y
    aeq (x, y) (x', y') = aeq x x' ∧ aeq y y'
  ```
Instances of \( Nom \)

- \( Name \): where the rubber meets the road

\[
\text{instance } \text{Nom } Name \text{ where} \\
\text{swap } a \ b \ c = \text{if } a \equiv c \text{ then } b \\
\quad \text{else if } b \equiv c \text{ then } a \text{ else } c \\
\text{fresh } a \ b = a \neq b \\
\text{aeq } a \ b = a \equiv b
\]

- Abstractions

\[
\text{instance } (\text{Nom } a) \Rightarrow \text{Nom } (\text{Name } \parallel a) \text{ where} \\
\text{swap } a \ b \ (c \parallel x) = (\text{swap } a \ b \ c) \parallel (\text{swap } a \ b \ x) \\
\text{fresh } a \ (b \parallel x) = a \equiv b \lor \text{fresh } a \ x \\
\text{aeq } (a \parallel x) \ (b \parallel y) = (a \equiv b \land \text{aeq } x \ y) \\
\quad \lor (\text{fresh } a \ y \land \text{aeq } x \ (\text{swap } a \ b \ y))
\]
Class Subst

- For ordinary types, substitutions ignore structure.

  \[
  \text{instance } (\text{Subst } t \ a, \text{Subst } t \ b) \Rightarrow \text{Subst } t \ (a, b) \ \text{where}
  \]

  \[
  \begin{align*}
  \text{subst } a \ t \ (x, y) &= \text{do } x' \leftarrow \text{subst } a \ t \ x \\
  y' &\leftarrow \text{subst } a \ t \ y \\
  \text{return } (x', y')
  \end{align*}
  \]

- For abstractions, substitutions rename bound names, then proceed

  \[
  \text{instance } \text{Subst } t \ a \Rightarrow \text{Subst } t \ (\text{Name} \parallel a) \ \text{where}
  \]

  \[
  \begin{align*}
  \text{subst } a \ t \ (b \parallel x) &= \text{do } b' \leftarrow \text{fresh} \\
  x' &\leftarrow \text{subst } a \ t \ (\text{swap } b \ b' \ x) \\
  \text{return } (b' \parallel x')
  \end{align*}
  \]
Class \textit{FreeVars}

- For ordinary types, \textit{fvs} is union of \textit{fvs} of components.

\begin{verbatim}
instance (FreeVars t a, FreeVars t b) ⇒ FreeVars t (a, b)
where
  FreeVars t (x, y) = union (fvs t x) (fvs t y)
\end{verbatim}

- For abstractions, remove bound name from set

\begin{verbatim}
instance FreeVars t a ⇒ FreeVars t (Name \parallel a) where
  fvs t (b \parallel x) = fvs t x \setminus \{b\}
\end{verbatim}
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 `\|\|`.
- 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`, `\|\|`, 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:

```haskell
instance Subst Lam Lam where
    subst a t (Var b) | a ≡ b = return t
    subst a t (Var b) = return (Var b)
    subst a t (App t1 t2) = ...
    subst a t (Lam abs) = ...
    ...
```
Solution

- Need a way to tell the library what constructors represent variables
- This is what *HasVar* is for.
- Generic definition of *Subst* looks like\(^a\)

\[
\text{instance } \text{Subst } t \ a \ \text{where}
\]
\[
\text{subst } a \ t \ x = \text{gmapM} \ (\text{subst } a \ t) \ x
\]

\[
\text{instance } \text{HasVar } a \Rightarrow \text{Subst } a \ a \ \text{where}
\]
\[
\text{subst } a \ t \ x = \text{if } \text{is_var } x \equiv \text{Just } a
\]
\[
\text{then return } t
\]
\[
\text{else } \text{gmapM} \ (\text{subst } a \ t) \ x
\]

- 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. Caml): 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