Generic capture-avoiding substitution

James Cheney

Binding Challenges workshop

April 24, 2005

My wish list

- Support for situations with unbound names and name generation (e.g. let-bound polymorphism, record fields, memory references, state ids, nonces.)

- Support for logics with unusual contexts of arbitrary “shape”, e.g., BI, separation logic

- Support for logics with unusual forms of quantification, e.g. Hoare logic, dynamic logic, nominal logic itself

- Support for unusual forms of binding, e.g. pattern matching

More challenges

- Proof terms in a sensible (e.g., predicative) constructive logic or functional programming language
- Formalized proofs mirror paper inductive proofs/recursive definitions
- Explainable to/usable by a 1st year grad student
- Support for capture-avoiding substitution

The challenge of capture-avoiding substitution

- “This generic programming stuff is neat and all, but it will never be able to deal with something really useful like capture-avoiding substitution, will it?” (SPJ, 2003, paraphrase)

- “This nominal stuff is interesting, if weird, but without HOAS’s *implementation and theoretical* support for substitution, how can it ever get off the ground?” (FP, 2004, paraphrase)

- Advanced abstract syntax techniques **must** support substitution.

- Generic programming techniques can help

Motivation

- Higher-order abstract syntax: second-class variables, $\alpha\beta\eta$-equivalence (formalized classically) provided by metalanguage
  - CAS provided for free at all types, but encodings difficult to analyze, intractable semantic problems

- Nominal (Gabbay-Pitts) syntax: first-class names, α-equivalence via swapping, freshness.
  - Analysis/semantics more straightforward, but CAS apparently must be written by hand for new types

Goal

- Provide capture-avoiding substitution “for free” in a real language
- by combining generic programming (GP) and nominal (NAS) techniques
- in a library that programmers can use to write *real programs* (or at least PL homework exercises or prototypes)
- and *without needing expertise in GP or NAS!*

In other words, I want to never ever again have to write (or read, or explain to others how to write, or tolerate, in any form) code like

this.

```
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))
;;
```

or this.

```
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)
;;
```

or this.

```
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)
```

Never.

I mean it.

In an ideal world...

In the binding-free case

- In the case of no binding, substitution is entirely algebraic

- Think of groups/rings/fields/algebras $K[X_1, \ldots, X_n]$ over generators $X_1, \ldots, X_n$

- Suppose $h : \{X_1, \ldots, X_n\} \to K'$.

- There is a *homomorphic extension* $h^\circ : K[X_1, \ldots, X_n] \to K'$ satisfying $h(X_i) = h^\circ(x_i)$ for each X_i.

Focus on initial Σ-algebras

- Let's focus on initial Σ-algebras,
- that is, algebras over some uninterpreted signature Σ
- that is, *sets of terms*.
- Closed terms T_Σ, terms T_Σ^V over variables V
- Homomorphic extension unique.

Duh

- It's easy to write down the unique endomorphism generated by h in a term algebra over

$$\Sigma = (c, \ldots, f^n, \ldots)$$

- To wit:

$$\begin{array}{rcll} h : V \to T_\Sigma^V & \mapsto & h^\circ : T_\Sigma^V \to T_\Sigma^V & \\ h^\circ(c) & = & c & \\ h^\circ(f^n(t_1, \ldots, t_n)) & = & f^n(h^\circ(t_1), \ldots, h^\circ(t_n)) & \\ h^\circ(X) & = & h(X) & (X \in V) \end{array}$$

- This function is almost completely uninteresting.

Duh (II)

- Now what if we have a *sorted* Σ-algebra

$$\Sigma = (\{S_1, \ldots, S_n\}, c : S, \ldots, f : S_1 \times \cdots \times S_n \to S, \ldots)$$

- Then we have

$$\begin{array}{rcll} h : V(S) \to T_\Sigma^V(S) & \mapsto & h_S^\circ : T_\Sigma^V(S) \to T_\Sigma^V(S) & \\ h_S^\circ(c) & = & c & (c : S) \\ h_S^\circ(f(t_1, \ldots, t_n)) & = & f(h_{S_1}^\circ(t_1), \ldots, h_{S_n}^\circ(t_n)) & (f : S_1 \times \cdots \times S_n \to S \\ h_S^\circ(X) & = & h(X) & (X : S \in V) \end{array}$$

- Only interesting part: the types

Duh (III)

- For a particular Σ-algebra, we can easily code up substitution in, say, Haskell.

- In fact, for a given term structure, there is one interesting case, the rest are structural recursions:

$$\begin{aligned}
&subst && :: (V \rightarrow T) \rightarrow (T \rightarrow T) \\
&subst\ f\ (Var\ x) && = f\ x \\
&subst\ f\ C && = C \\
&subst\ f\ (F\ (t1, ..., tn)) && = F\ (subst\ f\ t1, ..., subst\ f\ tn) \\
&\ldots
\end{aligned}$$

Duh (IV)

- For a particular *sorted* Σ-algebra, we can less easily code up substitution in, say, Haskell.

$$
\begin{array}{ll}
subst\_S\_S & :: (V\_S \rightarrow T\_S) \rightarrow (T\_S \rightarrow T\_S) \\
subst\_S\_S\ f\ (SVar\ x) & = f\ x \\
subst\_S\_S\ f\ C & = C \\
subst\_S\_S\ f\ (F\ (t1, ..., tn)) & = F\ (subst\_S1\ f\ t1, ..., subst\_Sn\ f\ tn) \\
\ldots & \\
subst\_S\_T & :: (V\_S \rightarrow T\_S) \rightarrow (T\_T \rightarrow T\_T) \\
subst\_S\_T\ f\ D & = D \\
subst\_S\_T\ f\ (G\ (t1, ..., tn)) & = G\ (subst\_S1\ f\ t1, ..., subst\_Sn\ f\ tn) \\
\ldots &
\end{array}
$$

Snag

- Two problems: we need to write mn functions to substitute m substitutable types into n types in which variables can appear

- Most cases are "the same", just not in an easy to express way

- To add insult to injury, need to use a different function name for each pair of types involved.

- (For this reason, usually consider substitution for at most 2-3 kinds of things.)

Type classes to the rescue?

- Haskell's powerful *type class* feature at least lets us overload the name $subst$.

$$\textbf{class}\ Subst\ v\ t\ u\ \textbf{where}$$
$$\quad subst :: (v \to t) \to u \to u$$

- Intuitively, $Subst\ v\ t\ u$ = "t substitutable for v in u"

- But mn cases still need to be written.

Generic programming to the rescue

- *Generic programming* (in the context of typed functional languages) means *writing concise definitions of functions that work for any type*.
- Popular approaches based on generalizing maps, folds, etc.
- Most advanced GP features provided in/for Haskell
- Straightforward to implement algebraic substitution using existing GP techniques.

That’s all well and good, but...

A bigger snag

- **If you have name-binding, apparently this all breaks.**

$$\begin{aligned}
&\textbf{data } \mathit{Exp} = \mathit{Var}\ V \mid \mathit{App}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Lam}\ V\ \mathit{Exp} \\
&\mathit{subst}\ a\ t\ (\mathit{Var}\ v) \quad\ = \textbf{if } a \equiv b \textbf{ then } \mathit{return}\ t \textbf{ else } \mathit{return}\ (\mathit{Var}\ b) \\
&\mathit{subst}\ a\ t\ (\mathit{App}\ t1\ t2) = \textbf{do } t1' \leftarrow \mathit{subst}\ a\ t\ t1 \\
&\qquad\qquad\qquad\qquad\qquad\quad t2' \leftarrow \mathit{subst}\ a\ t\ t2 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{App}\ t1'\ t2') \\
&\mathit{subst}\ a\ t\ (\mathit{Lam}\ v\ t1) \ = \textbf{do } v' \ \leftarrow \mathit{gensym}\ v \\
&\qquad\qquad\qquad\qquad\qquad\quad t1' \leftarrow \mathit{subst}\ v\ (\mathit{Var}\ w)\ t1 \\
&\qquad\qquad\qquad\qquad\qquad\quad t1'' \leftarrow \mathit{subst}\ a\ t\ t1' \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{Lam}\ v'\ t1'')
\end{aligned}$$

Back to the drawing board!

What about HOAS?

- In a functional language, can encode languages with bound variables using function types.

- Then *capture-avoiding substitution becomes function application*

- The theory of HOAS + CAS is nonalgebraic; recursion/induction with HOAS is a very hard current (+ last 10-15 years) research area.

- Whatever its merits, HOAS not practical in typical current functional languages because functions can't be "decomposed"

Nominal abstract syntax to the rescue?

- Nominal abstract syntax (i.e. Gabbay-Pitts FM syntax of binding via swapping and freshness) purports to be compatible with inductive/algebraic reasoning

- Can it be incorporated into a “real” language? Yes—FreshML, αProlog

- Does *capture-avoiding* substitution fit into this framework? um possibly...

- Is it still algebraic enough to define generically? Claim yes.

Nominal algebra (TODO)

- We identify V_S with sets of names $\mathbb{A}_S$ in NAS, one per sort.

- Suppose we have a "nominal Σ-algebra" with function symbol sorts like

$$f : \langle V \rangle S \to S, g : S \times \langle V \rangle \langle V \rangle S \to T, \ldots$$

  where $\langle V \rangle S$ is the *sort of things* $\langle a \rangle x$ *consisting of a value* x *of type* S *with one bound name* a *of type* V (a.k.a. "abstraction")

- Suppose also: For some sorts S, know a "variable" function symbol $v_S : V \to S$ embedding names as things of type S.

Nominal homomorphism theorem

- A nominal homomorphism ought be a finitely supported function satisfying:

$$h(\langle a \rangle x) = \langle a \rangle h(x) \qquad a \# h$$

for any "fresh" a *not mentioned in h*

- A "homomorphism theorem" (hopefully true) for nominal algebras:

Pre-Theorem 1. *For any finitely supported* $h : V \to T^V \Sigma(S)$ *there exists a unique homomorphism* $(h'^{\circ}_S : T^V \Sigma(S') \to T^V \Sigma(S') | S' \in Sorts)$ *extending* h*.*

Nominal capture-avoiding substitution

- Let

$$h_{[x \mapsto t]}(y) = \begin{cases} t & (x = y) \\ y & \end{cases}$$

- Claim: For all "reasonable" encodings of languages with binding, $[x \mapsto t]$ defined as $[x \mapsto t]u = h^{\circ}(u)$ is capture-avoiding substitution.

- Why? Because for abstractions, we require

$$[a \mapsto t](\langle b \rangle x) = \langle b \rangle [a \mapsto t]x$$

for $b \# a, t$.

Example: Lambda

- A nominal Σ algebra Λ_α for untyped λ terms:

$$v_\Lambda : V \to \Lambda_\alpha \quad @ : \Lambda_\alpha \times \Lambda_\alpha \to \Lambda_\alpha \quad \lambda : \langle V \rangle \Lambda_\alpha \to \Lambda_\alpha$$

  Encoding of ordinary λ terms Λ:

$$\ulcorner x \urcorner = v_\Lambda(x) \quad \ulcorner t\ u \urcorner = @(\ulcorner t \urcorner, \ulcorner u \urcorner) \quad \ulcorner \lambda x.t \urcorner = \langle x \rangle \ulcorner t \urcorner$$

- Define α-equivalence $\equiv_\alpha$: $\Lambda \times \Lambda$ and CAS $\{x \mapsto t\}$ "as usual"

Some more pre-theorems

- Believe this to be the case given appropriate definitions:

  Pre-Theorem 2. $\Lambda/_{\equiv_\alpha}$ *is a nominal* Σ *algebra and* $\ulcorner\cdot\urcorner : \Lambda/_{\equiv_\alpha} \longrightarrow \Lambda_\alpha$ *is a nom.* Σ *algebra isomorphism.*

- Then it follows immediately that

  Corollary 1 (Adequacy). *For any* x, t, u*:*

$$\ulcorner\{x \mapsto u\}t\urcorner = [x \mapsto \ulcorner u\urcorner]\ulcorner t\urcorner$$

So we're done... right?

- This shows *in principle* that we can get CAS in a nice algebraic way.
- At this point, mathematicians generally call it a day and go home.
- But I'm a computer scientist.
- I want an implementation that does **all** the work for me
- This takes a bit of doing.

I have implemented this and it works.

FreshLib

- FreshLib is a small Haskell class library
- It implements NAS/swapping/freshness/$\approx_\alpha$ for all “nominal” types, including user-defined ones and “name” and “abstraction” types
- It provides CAS and FV functions “for free”, if you specify the variable constructor of a type.
- Almost no boilerplate code needs to be written by user for new datatypes.

Scrap your nameplate

Here is the specification of Λ in $\mathit{FreshLib}$.

$$\mathbf{data}\ \mathit{Exp} = \mathit{Var}\ \mathit{Name} \mid \mathit{App}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Lam}\ (\mathit{Name} \setminus\!\!\setminus\!\!\setminus \mathit{Exp})$$

$$\begin{aligned}
&\mathbf{instance}\ \mathit{HasVar}\ \mathit{Exp}\ \mathbf{where}\\
&\quad \mathit{is_var}\ (\mathit{Var}\ x) = \mathit{Just}\ x\\
&\quad \mathit{is_var}\ _ \qquad\quad\ \ = \mathit{Nothing}
\end{aligned}$$

plus a few imports and other things.

Scrap more nameplate

Here's System F.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Exp} = \mathit{Var}\ \mathit{Name} \mid \mathit{App}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Lam}\ (\mathit{Name} \setminus\!\!\setminus\!\!\setminus \mathit{Exp}) \\
&\qquad\qquad\ \mid \quad \mathit{TApp}\ \mathit{Exp}\ \mathit{Ty} \mid \mathit{TLam}\ (\mathit{Name} \setminus\!\!\setminus\!\!\setminus \mathit{Ty}) \\
&\textbf{data}\ \mathit{Ty} = \ \ \mathit{TVar}\ \mathit{Name} \mid \mathit{FnTy}\ \mathit{Ty}\ \mathit{Ty} \mid \mathit{AllTy}\ (\mathit{Name} \setminus\!\!\setminus\!\!\setminus \mathit{Ty}) \\
&\textbf{instance}\ \mathit{HasVar}\ \mathit{Exp}\ \textbf{where} \\
&\quad \mathit{is\_var}\ (\mathit{Var}\ x) = \mathit{Just}\ x \\
&\quad \mathit{is\_var}\ \_ \qquad\quad\ \ = \mathit{Nothing} \\
&\textbf{instance}\ \mathit{HasVar}\ \mathit{Ty}\ \textbf{where} \\
&\quad \mathit{is\_var}\ (\mathit{TVar}\ x) = \mathit{Just}\ x \\
&\quad \mathit{is\_var}\ \_ \qquad\qquad\ = \mathit{Nothing}
\end{aligned}
$$

The scrapping continues

Here's LF.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Exp} = & & \mathit{Cnst}\ \mathit{String} \mid \mathit{Var}\ \mathit{Var} \mid \mathit{App}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Lam}\ (\mathit{Var} \setminus\!\!\setminus \mathit{Exp}) \\
\textbf{data}\ \mathit{Ty} & = & \mathit{TCnst}\ \mathit{String} \mid \mathit{PiTy}\ \mathit{Ty}\ (\mathit{Var} \setminus\!\!\setminus \mathit{Ty}) \mid \mathit{TVApp}\ \mathit{Ty}\ \mathit{Exp} \\
 & \mid & \mathit{TVar}\ \mathit{Name} \mid \mathit{TApp}\ \mathit{Ty}\ \mathit{Ty} \mid \mathit{TLam}\ \mathit{Kind}\ (\mathit{Name} \setminus\!\!\setminus \mathit{Ty}) \\
\multicolumn{3}{l}{\textbf{data}\ \mathit{Kind} = \mathit{KType} \mid \mathit{KPi}\ \mathit{Kind}\ (\mathit{Name} \setminus\!\!\setminus \mathit{Kind})} \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{HasVar}\ \mathit{Exp}\ \textbf{where}} \\
\multicolumn{3}{l}{\quad \mathit{is\_var}\ (\mathit{Var}\ x) = \mathit{Just}\ x} \\
\multicolumn{3}{l}{\quad \mathit{is\_var}\ \_ \qquad\quad\ \ = \mathit{Nothing}} \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{HasVar}\ \mathit{Ty}\ \textbf{where}} \\
\multicolumn{3}{l}{\quad \mathit{is\_var}\ (\mathit{TVar}\ x) = \mathit{Just}\ x} \\
\multicolumn{3}{l}{\quad \mathit{is\_var}\ \_ \qquad\qquad = \mathit{Nothing}}
\end{array}
$$

Yet more scrapping

The π-calculus:

$$\begin{aligned}
\textbf{data}\ \mathit{Proc} \ &= \mathit{Tau} \mid \mathit{Plus}\ \mathit{Proc}\ \mathit{Proc} \mid \mathit{Par}\ \mathit{Proc}\ \mathit{Proc} \mid \mathit{Repl}\ \mathit{Proc} \\
&\mid\ \mathit{In}\ \mathit{Name}\ (\mathit{Name} \mathbin{\backslash\!\!\backslash\!\!\backslash} \mathit{Proc}) \mid \mathit{Out}\ \mathit{Name}\ \mathit{Name}\ \mathit{Proc} \\
&\mid\ \mathit{Res}\ (\mathit{Name} \mathbin{\backslash\!\!\backslash\!\!\backslash} \mathit{Proc}) \mid \mathit{Match}\ \mathit{Name}\ \mathit{Name} \\
\textbf{data}\ \mathit{Trans} &= \mathit{TTau}\ \mathit{Proc}\ \mathit{Proc} \\
&\mid\ \mathit{TIn}\ \mathit{Proc}\ \mathit{Name}\ (\mathit{Name} \mathbin{\backslash\!\!\backslash\!\!\backslash} \mathit{Proc}) \\
&\mid\ \mathit{TBOut}\ \mathit{Proc}\ \mathit{Name}\ (\mathit{Name} \mathbin{\backslash\!\!\backslash\!\!\backslash} \mathit{Proc}) \\
&\mid\ \mathit{TFOut}\ \mathit{Proc}\ \mathit{Name}\ \mathit{Name}\ \mathit{Proc}
\end{aligned}$$

Note: $HasVarName$ already has an instance (CAS of name for name always makes sense)

How it works

- Types $Name$, $Name : a$: represent names, name-abstractions.
- Class Nom: provides swapping, freshness, α-equivalence
- Class $HasVar$: says what case of user-defined type acts as variable of that type.
- Class $Subst$, $FreeVars$: substitution and free variable sets
- Class instances & SYB library used to automatically extend to new datatypes (hot off the press)

Demo

- Details and implementation at

  http://homepages.inf.ed.ac.uk/jcheney/FreshLib.html

- Also implemented in αProlog (by hacking CAS operator into the language)

  http://homepages.inf.ed.ac.uk/jcheney/projects/aprolog.html

What’s next

Free variable sets

- This is also a homomorphism, but onto a nom. Σ algebra of sets of names.

- It can be (and has been) implemented as a generic function too.

Multiple name types

- Right now only one name type $Name$ allowed.
- This is bad because bindings can “interfere” causing undesired effects.
- Working on this, but appears tricky.

Nonstandard binding

- Nonstandard = binding some distinguished names of one term within another

- E.G. $\Gamma \vdash e : T$, $case\ e\ of\ p(x, y) \to e'$

- Handle using class $BType$ with methods $bound :: a \to [Name]$ and $equiv :: a \to a \to MaybePerm$

- Bound: says what names are bound. Equiv: says when two a's are equal *up to a permutation.*

Making substitution pure

- In FreshML, $subst$ is a “pure” (side-effect free) function.
- In FreshLib, $subst$ is not, so monadic.
- Peyton Jones and Thompson suggest a way around this (used in Haskell inliner)
- Their idea: Track set of names in scope, use hashing to guess a fresh name when needed.
- They say it works surprisingly well.

Why not FreshML?

- FreshML provides even better built-in support for NAS!
- But FreshML (and ML family generally) have almost no support for GP techniques.
- Haskell type classes + generics give us 90% of what FreshML does with much less relative coding effort.
- It might be easy to hack built-in generic CAS into FreshML (it was in αProlog).

Theory

- I know this works, but the theory should be worked out.
- if it hasn't been already.

Theoretical support

- Theoretical support (e.g., “free” substitution lemmas) is a key advantage of HOAS.

- Future direction: can generic CAS be integrated into theorem provers?

- Can *proofs* of substitution principles be derived automatically?

- This would, I believe, establish NAS as competitive alternative to HOAS beyond any question.

Conclusion

- Support for capture avoiding substitution is one apparent advantage of higher-order abstract syntax over other approaches.
- In NAS, however, CAS can be treated algebraically extending standard techniques from universal algebra.
- Type classes and generic programming techniques for Haskell can be used to provide NAS and CAS “for free”, as a black-box library
- Interesting extensions appear possible, current work.

Plug

- Details and implementation at

  http://homepages.inf.ed.ac.uk/jcheney/FreshLib.html

- Also implemented in αProlog (by hacking CAS operator into the language)

  http://homepages.inf.ed.ac.uk/jcheney/projects/aprolog.html