Binders Unbound

Stephanie Weirich

Brent Yorgey

Tim Sheard

Noah David Yorgey,
Sat Sept 17th, 10:24 PM

From Inspiration to …

```
data N = String
data Exp = Var N
         | Lam N Exp
         | App Exp Exp
```

```
fv (Var x)    = [x]
fv (Lam x e) = fv e // [x]
fv (App e e') = fv e ++ fv e'
```

Bug source: may need to freshen when recurring under binders

Alpha-equivalence & Capture-avoiding substitution

Frustration!
This should be "easy"

Declarative Specification for Binding

```
data N = Name Exp
data Exp = Var N
         | Lam (Bind N Exp)
         | App Exp Exp
```

Generic programming available for fv, aeq, substitution

Monadic destructor for binding ensures freshness

```
unbind :: Fresh m => Bind N Exp -> m (N,Exp)
```

Get right to the interesting part of the implementation!!

James Cheney,
Scrap Your Nameplate
ICFP 2005

Unbound Library

cabal install unbound

```
aeq :: (Alpha a) => a -> a -> Bool

data N = Name Exp
data Exp = Var N
     | Lam (Bind N Exp)
     | App Exp Exp

$(derive [''Exp])
instance Alpha Exp
```

```
> let x = string2Name "x" :: N
> let y = string2Name "y" :: N
> Lam x (Var x) `aeq` Lam y (Var y)
True
```

Unbound library

- Several small improvements to FreshLib:
  - Documentation and cabal distribution
  - Support for multiple atom sorts
  - Improved substitution interface
  - Two different monads for "freshness"
- *Expressive general binding specification language*

```
bind :: (Alpha b) => N -> b -> Bind N b
bind :: (Alpha a, Alpha b) => a -> b -> Bind a b
```

What sort of binding patterns can be specified by type structure?

Beyond Single Binding

\ x y z -> (x z) (y z)

```
data N = Name Exp
data Exp = Var N
         | Lam (Bind [N] Exp)
         | App Exp [Exp]
```

\ (x, Just y) -> x + y

All names in the pattern expression are bound in the body of the Bind

```
data N = Name Exp
data Pat = PVar N
         | PCon String [Pat]
data Exp = Var N
         | Lam (Bind Pat Exp)
         | App Exp Exp
         | Con String [Exp]
```

Embedded Terms in Patterns

let x = e in e'

x bound in e' but not e

```
data Exp = ...
     | Let Exp (Bind N Exp)
```

All names in the pattern *except in Embeds* are bound in the body of the Bind

```
data Exp = ...
     | Let (Bind (N, Embed Exp) Exp)
```

let x1 = e1
x2 = e2
...
in e'

x1,x2... bound in e'

```
data Exp = ...
     | Let [Exp] (Bind [N] Exp)
```

```
     | Let (Bind [(N, Embed Exp)] Exp)
```

Can enforce equal number of LHSs and RHSs

Double Binding (recursive)

let rec x = e in e'

x bound in e and e'

```
data Exp = ...
  | Let (Bind (Rec (N, Embed Exp)) Exp)
  | Let (Bind N (Exp, Exp))
```

All names in a rec pattern are bound in *both* the Embeds and the body of the Bind

let rec x1 = e1
x2 = e2
...
in e'

x1, x2… bound in e1,e2… and e'

```
data Exp = ...
  | Let (Bind (Rec [(N, Embed Exp)]) Exp)
  | Let (Bind [N] ([Exp], Exp))
```

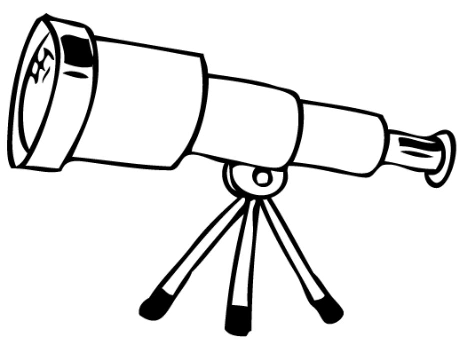
Double Binding (non-recursive)

```
let*  x1 = e1
      x2 = e2
      x3 = e3
      ...
in e'
```

x1 bound in e2, e3, e'
x2 bound in e3, e'
x3 bound in e'

```
data Exp = ...
   | Let (Bind LetPat Exp)

data LetPat =
     Nil
   | Cons (Rebind (N, Embed Exp)
                  LetPat)
```



Binding Specification Language

- T ::=   (Terms)
  - | Primitive types, Int, Char, etc.
  - | Regular datatypes of terms, i.e. [T], (T,T)
  - | Name T
  - | Bind P T
- P ::=   (Patterns)
  - | Name T
  - | Primitive types, Int, Char, etc.
  - | Regular datatypes of patterns, i.e. [P], (P,P)
  - | Embed T
  - | Rec P
  - | Rebind P P
  - | Shift P

Semantics

- Paper gives precise semantics for fv, aeq and subst for terms and patterns composed of these types
- Semantics based on *locally nameless representation*
  - Simple definitions of operations
  - Rec/Shift inspired by semantics
- Proofs of basic properties of the operations
- Implementation follows semantics & uses RepLib library for generic programming (~2500 loc)

Future work

- Scope preservation   (see Pouillard and Westbrook)
- Declarative semantics, independent of variable representation
- Alternative implementations (nominal, canonical, optimized?)
- Integration with theorem prover

Related Work

- Cheney, FreshLib
- Pottier, CαML
- Charguéraud, *The Locally Nameless Representation*
- Sewell et al. OTT
  Urban, *General Bindings and Alpha-Equivalence in Nominal Isabelle*

Summary

- Separate specification of binding structure from implementation
- Abstract types define a EDSL for binding
- Type-generic programming automates boilerplate
- Locally nameless representation simplifies semantics