7 Naming and Binding

"His name is Fluffy? I thought his name was 'STOP IT!'"
7 Binding Constructs

Names and bindings
Roles of bindings in the language
Extraneous bindings
Bindings as part of the language
First-order vs. higher-order abstract syntax
Names and Bindings

- **Name**: placeholder for language objects
- Two main uses of names:
  1. values & functions/procedures/methods
  2. parameters
- **Binding**: a pair (Name, Object)
- **Definition**: creation of a binding in some name space
- **Use**: lookup object bound to name
Roles of Bindings

- **Integral part** of language:
  Language defines names and binding constructs
  *Examples: functions, variables in PLs*

- **Extraneous** to language:
  Binding constructs of the language environment are used to define and use names
  *Examples: registers in pocket calculators, (named) clipboards, address books (phone, email)*
Extraneous Bindings

Language: Expression Language

\[
\text{data } \text{Exp} = T \mid F \mid \text{Zero} \mid \text{Suc Exp} \mid \text{Eq Exp Exp} \mid \\
\mid \text{Cond Exp Exp Exp}
\]

Environment: Haskell (= metalanguage)

\[
\text{one} = \text{Suc Zero} \\
\text{three} = \text{Suc (Suc one)}
\]

\[
\text{peano} \ 0 = \text{Zero} \\
\text{peano} \ n = \text{Suc (peano (n-1))}
\]
Extraneous Bindings (2)

data Music =
    Note Pitch Duration [Attr]
    | Rest Duration
    | Music :+: Music
    | Music :=: Music
    | ...

cMajor :: [Music]
cMajor = [Note p 1 [] | p <- [c,d,e,f,g,a,b,c]]

ostinato :: Music -> Music
ostinato m = m :+: ostinato m
Bindings within DSLs

- **First-order abstract syntax (FOAS):** Binding structure is represented by names and scoping rules (needs extra rules)

- **Higher-order abstract syntax (HOAS):** Binding structure is completely represented in the abstract syntax tree (no extra rules needed)

- **Graphs:** Edge represents reference to binding site

- **deBruijn indices:** Number $k$ refers to $k$th enclosing binder

- **Binding constructs of metalanguage:** Reuse binding structure of host language
**FOAS Bindings**

**Expression Language**

```haskell
type Name = String

data Exp = T | F | Zero | Suc Exp | Eq Exp Exp |
        | Cond Exp Exp Exp
        | Let Name Exp Exp
        | Var Name
```

**How to define the semantics of binding constructs?**

- **Substitution**: Transform expressions by replacing names by their definition

  ```haskell
  Let "x" (Suc Zero) (Eq (Var "x") (Var "x"))
  = Eq (Suc Zero) (Suc Zero)
  ```

- **Environment**: Store and retrieve bindings in a data structure
Substitution Semantics

data Exp = T | F | Zero | Suc Exp | Eq Exp Exp |
          | Cond Exp Exp Exp |
          | Let Name Exp Exp |
          | Var Name

eval :: Exp -> Val
eval T = B True
eval F = B False
eval Zero = I 0
...
eval (Let v d e) = eval (subst d v e)

Problems

• Free variables: Expression Suc (Var “x”) is undefined

• Nested bindings: Proper implementation of subst needs to do renaming and requires a supply of fresh variables
Environment Semantics

data Exp = T | F | Zero | Suc Exp | Eq Exp Exp | Cond Exp Exp Exp | Let Name Exp Exp | Var Name

eval :: Exp -> Env -> Val

eval T _ = B True

eval F _ = B False

eval Zero _ = I 0

... 

eval (Let v d e) m = eval e ((v,eval d m):m)

eval (Var v) m = fromJust (lookup v m)

Potential problem

• **Nonlocal variables**: Naive approach implements dynamic scoping (see CS 381)
**HOAS Bindings**

```haskell
```

```haskell
eval :: Exp -> Val
eval T = B True
eval F = B False
eval Zero = I 0
...
eval (Let f d) = eval (f d)
```

Let \(v\) \(d\) \(e\) is represented in HOAS as

Let \((\lambda v -> e)\) \(d\)

Let "x" (Suc Zero)
(Q (Var "x") (Var "x"))
\equiv
Let (\(\lambda x -> Q x\) x) (Suc Zero)

Some syntactic sugar ...

parExpr \(x\) = Eq \(x\) \(x\)

with = Let

parExpr "with" Suc Zero
HOAS Evaluation

Advantages

• *No* need for *substitution* and/or *renaming* operations

• Built-in alpha equivalence

• When using binding constructs of metalanguage: *Trivial implementation* of binding semantics (because it is provided by the metalanguage)

Disadvantages

• *Restricted syntax* (imposed by metalanguage)

• Printing requires tricks
Adding Function Definitions (FOAS)

data Exp = T | F | Zero | Suc Exp | Eq Exp Exp |
       | Cond Exp Exp Exp
       | Fun Name Name Exp Exp
       | Call Name Exp
       | Var Name

function definition

function call

eval (Fun "f" "x" (Suc (Suc (Var "x"))))
  (Call "f" (Call "f" Zero))
  =
  Suc (Suc (Suc (Suc Zero)))
Functions (FOAS)

data Exp = T | F | Zero | Suc Exp | Eq Exp Exp | Cond Exp Exp Exp |
          | Fun Name Name Exp Exp | Call Name Exp |
          | Var Name

data Val = B Bool |
          | I Int |
          | C Name Exp Env

eval :: Exp -> Env -> Val
...
eval (Fun f v d e) m = eval e ((f,C v d m):m)
eval (Call f e) m = case lookup f m of
  Just (C v d m') ->
    eval d ((v,eval e m):m')
eval (Var v) m = eval (fromJust (lookup v m))
Functions (FOAS)

data Exp = T | F | Zero | Suc Exp | Eq Exp Exp |
       | Cond Exp Exp Exp |
       | Fun Name Name Exp Exp | Call Name Exp |
       | Var Name

data Val = B Bool |
       | I Int |
       | C Name Exp Env

eval :: Exp -> Env -> Val
... 
eval (Fun f v d e) m = eval e m' ......... Recursion

where m' = ((f,C v d m') : m)

eval (Call f e) m = case lookup f m of

Just (C v d m') ->

    eval d ((v,eval e m) : m')

eval (Var v) m = eval (fromJust (lookup v m))
HOAS Representation of Function Definitions

\[
\text{data } \text{Exp} = \text{T} \mid \text{F} \mid \text{Zero} \mid \text{Suc } \text{Exp} \mid \text{Eq } \text{Exp } \text{Exp} \mid \\
| \text{Cond } \text{Exp } \text{Exp} \text{Exp} \mid \text{Let } ((\text{Exp } \to \text{Exp}) \to \text{Exp}) (\text{Exp } \to \text{Exp})
\]

**Definition eval of remains unchanged!**

Fun "f" "x" (Suc (Suc (Var "x")))
\(\text{(Call } "f" \text{ (Call } "f" \text{ Zero)})\)
≡

Let (\f->f (f Zero)) (\x->Suc (Suc x))

**More syntactic sugar ...**

use = id
within = Let

Fun \(f\) \(v\) \(d\) \(e\) is represented in HOAS as

Let (\f->e) (\v->d)