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

```haskell
data Exp = T | F | Zero | Suc Exp | Eq Exp Exp |
| Cond Exp Exp Exp
```

Environment: Haskell (= metalanguage)

```haskell
one = Suc Zero
three = Suc (Suc one)

peano 0 = Zero
peano n = 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

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?

Let “x” (Suc Zero)
(Eq (Var “x”) (Var “x”))
= Eq (Suc Zero) (Suc Zero)

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

evaluate T _ = B True
evaluate F _ = B False
evaluate Zero _ = I 0
...
evaluate (Let v d e) m = evaluate e ((v, evaluate d m):m)
evaluate (Var v) m = fromJust (lookup v m)

type Env = [(Name, Val)]

Potential problem

- **Nonlocal variables**: Naive approach implements dynamic scoping (see CS 381)
          | Let (Exp -> Exp) Exp

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

Some syntactic sugar ...

parExpr x = Eq x x
with = Let
parExpr `with` Suc Zero

Let v d e
is represented in HOAS as
Let (\v->e) d
Let “x” (Suc Zero)
  (Eq (Var “x”) (Var “x”))
≡
Let (\x->Eq x x) (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

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)

\[ \text{data Exp} = \text{T} \mid \text{F} \mid \text{Zero} \mid \text{Suc Exp} \mid \text{Eq Exp Exp} \mid \text{Cond Exp Exp Exp} \]
\[ \quad \mid \text{Fun Name Name Exp Exp} \]
\[ \quad \mid \text{Call Name Exp} \]
\[ \quad \mid \text{Var Name} \]

\[ \text{data Val} = \text{B Bool} \]
\[ \quad \mid \text{I Int} \]
\[ \quad \mid \text{C Name Exp Env} \]

\[ \text{eval :: Exp \to Env \to Val} \]
\[ \ldots \]
\[ \text{eval (Fun f v d e) m} = \text{eval e m'} \]
\[ \text{where m' = ((f,C v d m'):m)} \]
\[ \text{eval (Call f e) m} = \text{case lookup f m of} \]
\[ \quad \text{Just (C v d m') \to eval d ((v,eval e m):m')} \]
\[ \text{eval (Var v) m} = \text{eval (fromJust (lookup v m))} \]
HOAS Representation of Function Definitions

```haskell
data Exp = T | F | Zero | Suc Exp | Eq Exp Exp | Cond Exp Exp Exp
         | Let ((Exp -> Exp) -> Exp) (Exp -> Exp)
```

**Definition** `eval` of remains unchanged!

```haskell
Fun "f" "x" (Suc (Suc (Var "x")))
   (Call "f" (Call "f" Zero))
≡

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

Fun `f` `v` `d` `e` is represented in HOAS as

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

More syntactic sugar ...

```haskell
use = id
within = Let
```

```haskell
f x = Suc (Suc x)
call f = f (f Zero)
use f `within` call
```