Functional and Imperative Programming

Variable assignment in an environment

Components:

- Variable environment, mimics computer memory, we limit to integers

```
type Var = String
type Env = Var -> Integer
```

- Datatype for expressions

```
data Expr = I Integer | V Var
          | Add Expr Expr
          | Subtr Expr Expr
          | Mult Expr Expr
          deriving (Eq,Show)
```

- Evaluation of an expression in an environment

```
eval :: Expr -> Env -> Integer
eval (I i) _ = i
eval (V name) env = env name
eval (Add e1 e2) env = (eval e1 env) + (eval e2 env)
eval (Subtr e1 e2) env = (eval e1 env) - (eval e2 env)
eval (Mult e1 e2) env = (eval e1 env) * (eval e2 env)
```

Variable assignment in an environment

- Environment initialisation

```
initEnv :: Env
initEnv = \ _ -> undefined
-- initEnv = const undefined
```

- Environment update

```
update :: Eq a => (a -> b) -> (a,b) -> a -> b
update f (x,y) = \ z -> if x == z then y else f z

updates :: Eq a => (a -> b) -> [(a,b)] -> a -> b
updates = foldl update
```

Assignment:

```
assign :: Var -> Expr -> Env -> Env
assign var expr env =  update env (var, eval expr env)

*Lecture4> assign "b" (I 9) (assign "a" (I 1) initEnv) "a"
1
```

Recap: 4 ingredients of imperative programming

1. Variable Assignment: `<var> := <expr>`
2. Conditional Execution: `if <bexpr> then <statement1> else <statement2>`
3. Sequential Composition: `<statement1> ; <statement2>`
4. Iteration: `while <expr> do <statement>`

Implementation of WHILE Language in Haskell

- Conditions

```
data Condition = Prp Var
          | Eq Expr Expr
          | Lt Expr Expr
          | Gt Expr Expr
          | Ng Condition
          | Cj [Condition]
          | Dj [Condition]
          deriving (Eq,Show)
```

- Statements

```
data Statement = Ass Var Expr
          | Cond Condition Statement Statement
          | Seq [Statement]
          | While Condition Statement
          deriving (Eq,Show)
```

Implementation of WHILE Language in Haskell

- Condition evaluation

```
evalc :: Condition -> Env -> Bool
evalc (Eq e1 e2) env = eval e1 env == eval e2 env
evalc (Lt e1 e2) env = eval e1 env <  eval e2 env
evalc (Gt e1 e2) env = eval e1 env >  eval e2 env
evalc (Ng c) env = not (evalc c env)
evalc (Cj cs) env = and (map (\ c -> evalc c env) cs)
evalc (Dj cs) env = or  (map (\ c -> evalc c env) cs)
```

- Statement execution

```
exec :: Statement -> Env -> Env
exec (Ass v e) env = assign v e env
exec (Cond c s1 s2) env =
 if evalc c env then exec s1 env else exec s2 env
exec (Seq ss) env = foldl (flip exec) env ss
exec w@(While c s) env =
 if not (evalc c env) then env
 else exec w (exec s env)
```

The return of Fibonacci

```
fib n
x := 0; y := 1;
while n > 0 do { z := x; x := y; y := z+y; n := n-1 }
```

```
fibonacci :: Integer -> Integer
fibonacci n = fibon (0,1,n) where
  fibon = whiler
          (\ (_,_,n) -> n > 0)
          (\ (x,y,n) -> (y,x+y,n-1))
          (\ (x,_,_) -> x)
while = until . (not.)
whiler p f r = r . while p f
```

Q: show the correctness of the imperative version of the Fibonacci algorithm

```
fib :: Statement
fib = Seq [Ass "x" (I 0), Ass "y" (I 1),
           While (Gt (V "n") (I 0))
            (Seq [Ass "z" (V "x"),
                  Ass "x" (V "y"),
                  Ass "y" (Add (V "z") (V "y")),
                  Ass "n" (Subtr (V "n") (I 1))])]

run :: [(Var,Integer)] -> Statement -> [Var] -> [Integer]
run xs program vars =
  exec program (updates initEnv xs) $$
    \ env -> map (\c -> eval c env) (map V vars)
```

Try it out:

```
runFib n = run [("n",n)] fib ["x"]
```

Loop invariant corresponds to induction step.

While loops as fixpoints

- Fixpoint = an element of the domain that is mapped to itself by the function.

```
fp :: Eq a => (a -> a) -> a -> a
fp f = until (\ x -> x == f x) f
```

- Naturally, Fibonacci

```
fbo n = (0,1,n) $$
       fp (\ (x,y,k) -> if k == 0 then (x,y,k) else (y,x+y,k-1))
```

- Approximate square roots (Babylonian method)

```
bab a = \ x -> ((x + a/x)/2)
sr a = fp (bab a) a
```

Q: How would you test `sr a`?

While loops as fixpoints

- Fixpoint = an element of the domain that is mapped to itself by the function.

```
fp :: Eq a => (a -> a) -> a -> a
fp f = until (\ x -> x == f x) f
```

- Naturally, Fibonacci

```
fbo n = (0,1,n) $$
      fp (\ (x,y,k) -> if k == 0 then (x,y,k) else (y,x+y,k-1))
```

- Approximate square roots (Babylonian method)

```
bab a = \ x -> ((x + a/x)/2)
sr a = fp (bab a) a
```

Q: How would you test `sr a`?

Hint: use iterate f x = [x, f x, f (f x), ...]

```
iterateFix :: Eq a => (a -> a) -> a -> [a]
iterateFix f = apprx . iterate f where
  apprx (x:y:zs) = if x == y then [x] else x: apprx (y:zs)

*Lecture4>iterateFix (bab 4) 1
```

Haskell fix operator

Special fixpoint operator to implement recursion:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

Or:

```
fix f = let x = f x in x
```

Fibonacci with fixpoint (check in pencil that `fbx` computes a fixed point, h = g h):

```
fbx n = (0,1,n) $$
        fix (\ f (x,y,k) -> if k == 0 then x else f (y,x+y,k-1))  -- fix g
```

Fibonacci without fixpoint:

```
fbb n = fbbb (0,1,n) where
  fbbb (x,y,n) = if n == 0 then x else fbbb (y,x+y,n-1)
```

Fibonacci curried without fixpoint:

```
fbc n = fbbc 0 1 n where
  fbbc x y n = if n == 0 then x else fbbc y (x+y) (n-1)
```

Fibonacci curried with fixpoint:

```
fbxc n = fbxcHelper 0 1 n where
  fbxcHelper = fix (\ f x y k -> if k == 0 then x else f y (x+y) (k-1))
```

Haskell Fix

Q: what is the difference between `fp` and `fix`?

Haskell Fix

Q: what is the difference between `fp` and `fix`?
A: fix is unconditional

```
fp' :: Eq a => (a -> a) -> a -> a
fp' f = fix (\ g x -> if x == f x then x else g (f x))
```

Q: define `until` using `fix`

Haskell Fix

Q: what is the difference between `fp` and `fix`?

A: fix is unconditional

```
fp' :: Eq a => (a -> a) -> a -> a
fp' f = fix (\ g x -> if x == f x then x else g (f x))
```

Q: define `until` using `fix`

A:

```
until' :: (a -> Bool) -> (a -> a) -> a -> a
until' p f = fix
          (\ g x -> if p x then x else g (f x))
```

Haskell Fix

Q: what is the difference between `fp` and `fix`?

A: fix is unconditional

```
fp' :: Eq a => (a -> a) -> a -> a
fp' f = fix (\ g x -> if x == f x then x else g (f x))
```

Q: define `until` using `fix`

A:

```
until' :: (a -> Bool) -> (a -> a) -> a -> a
until' p f = fix
           (\ g x -> if p x then x else g (f x))
```

Q: define `while` using `fix`

Haskell Fix

Q: what is the difference between `fp` and `fix`?

A: fix is unconditional

```
fp' :: Eq a => (a -> a) -> a -> a
fp' f = fix (\ g x -> if x == f x then x else g (f x))
```

Q: define `until` using `fix`

A:

```
until' :: (a -> Bool) -> (a -> a) -> a -> a
until' p f = fix
             (\ g x -> if p x then x else g (f x))
```

Q: define `while` using `fix`

A:

```
while' :: (a -> Bool) -> (a -> a) -> a -> a
while' p f = fix
             (\ g x -> if not (p x) then x else g (f x))
```

Quickcheck Example: The Pebble Game

```
data Color = W | B deriving (Eq,Show)

drawPebble :: [Color] -> [Color]
drawPebble [] = []
drawPebble [x] = [x]
drawPebble (W:W:xs) = drawPebble (B:xs)
drawPebble (B:B:xs) = drawPebble (B:xs)
drawPebble (W:B:xs) = drawPebble (W:xs)
drawPebble (B:W:xs) = drawPebble (W:xs)
```

Try out:

```
*Lecture4> drawPebble [W,W,B,B]
[B]
```

How do we go about generating random pebble tests?

Color Instance of Arbitrary

```
instance Arbitrary Color where
  arbitrary = oneof [return W, return B]
```

Now Quickcheck can “use” Color when generating testcases
Let us check some samples (`sampleF` is our own adaptation):

```
sampleF f g =
 do cases <- sample' g
    sequence_ (map (print.f) cases)

*Lecture4> sample $ (arbitrary :: Gen [Color])
[]
[W]
[W,B]

*Lecture4> sampleF drawPebbleList $ (arbitrary :: Gen [Color])
([],[])
([B],[B])
([B],[B,W,W])
```

Testing DrawPebble with Quickcheck

- Parity Invariant

```
numberW :: [Color] -> Int
numberW = length . (filter (== W))

parityW :: [Color] -> Int
parityW xs =  mod (numberW xs) 2

prop_invariant xs =
  parityW xs == parityW (drawPebble xs)

*Lecture4> quickCheck prop_invariant
+++ OK, passed 100 tests.
```

- Length Invariant

```
prop_length xs = length xs == length (drawPebble xs)

*Lecture4> quickCheck prop_length
*** Failed! Falsifiable (after 3 tests)
```

http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html

Recap properties of relations

Let R be a relation on set A, xRy means $(x,y) \in R$

- Irreflexive: $\nexists x \in A$, such that xRx
- Reflexive: $\forall x \in A$, xRx
- Antisymmetric: $\forall x, y \in A$, $xRy \land yRx \Rightarrow x=y$
- Asymmetric: both irreflexive and antisymmetric
- Symmetric: $\forall x, y \in A$, $xRy \Leftrightarrow yRx$
- Transitive: $\forall x, y, z \in A$, $xRy \land yRz \Rightarrow xRz$
    - Transitive closure: smallest transitive relation on A that contains R
- Linear: $\forall x, y \in A$, $xRy \lor yRx \lor x=y$

Composition of relations: let S, R be relations $S.R=\{(x,z) | \exists y \in Y\ xRy \land ySz\}$

- $R^{i+1}=R.R^{i}$
- Is associative
- The inverse is $R^{-1}.S^{-1}$