Lecture 8: Defining Haskell Types & Monads

CSCI 131
Fall, 2011

Kim Bruce

Higher-Order Functions

- Functions that take function as parameter
  - Ex: map:: (a → b) → ([a] → [b])
- Build new control structures
  - listify oper identity [] = identity
    listify oper identity (fst:rest) =
    oper fst (listify oper identity rest)
  - sum' = listify (+) 0
    mult' = listify (*) 1
    and' = listify (&&) True
    or' = listify (||) False

Quicksort

```
partition (pivot, []) = ([],[])
partition (pivot, first : others) =
   let
      (smalls, bigs) = partition(pivot, others)
   in
      if first < pivot
         then (first:smalls, bigs)
         else (smalls, first:bigs)
```

Type is:

```
partition :: (Ord a) => (a, [a]) -> ([a], [a])
```


Quicksort

```
qsort [] = []
qsort [singleton] = [singleton]
qsort (first:rest) =
   let
      (smalls, bigs) = partition(first,rest)
   in
      qsort(smalls) ++ [first] ++ qsort(bigs)
```

Type is:

```
qsort :: (Ord t) => [t] -> [t]
```


Quicksort - parametrically

```
partition (pivot, []) lThan = ([],[])
partition (pivot, first : others) lThan =
  let
    (smalls, bigs) = partition(pivot, others) lThan
  in
    if (lThan first pivot)
      then (first:smalls, bigs)
      else (smalls, first:bigs)

partition ::
        (t, [a]) -> (a -> t -> Bool) -> ([a], [a])

*Main> partition(6,[8,4,6,3])(>)
```


Quicksort

```
qsort [] lt = []
qsort [singleton] lt = [singleton]
qsort (first:rest) lt =
   let
      (smalls, bigs) = partition (first,rest) lt
   in
      qsort smalls lt ++ [first]
                      ++ qsort bigs lt

qsort :: [a] -> (a -> a -> Bool) -> [a]

*Main> qsort [33,66,32,87,999,2](>)
[999,87,66,33,32,2]
```

Defining New Types

- Type abbreviations
  - type Point = (Integer, Integer)
  - type Pair a = (a,a)
- data definitions
  - create new type with constructors as tags.
  - generative

Datatype Examples

- data Color = Red | Green | Blue
- data IntTree = Leaf Integer |
  Interior (IntTree,IntTree)
  deriving Show
- data Tree a = Niltree |
  Maketree (a, Tree a, Tree a)

Binary Search Using Trees

```
insert new Niltree = Maketree(new,Niltree,Niltree)
insert new (Maketree (root,l,r)) =
  if new < root
     then Maketree (root,(insert new l),r)
     else Maketree (root,l,(insert new r))

buildtree [] = Niltree
buildtree (fst : rest) =
                insert fst (buildtree rest)
```


Binary Search Tree

```
find elt Niltree = False
find elt (Maketree (root,left,right)) =
  if elt == root
     then True
     else if elt < root then find elt left
     else find elt right      -- elt > root

bsearch elt list = find elt (buildtree list)
```


Haskell is Lazy!

Lazy vs. Eager Evaluation

- Eager: Evaluate operand, substitute operand value in for formal parameter, and evaluate.
- Lazy: Substitute operand for formal parameter and evaluate body, evaluating operand only when needed.
  - Each actual parameter evaluated either not at all or only once! (Essentially cache answer once computed)
  - Like left-most outermost, but more efficient

Lazy evaluation

- Compute f(1/0,17) where f(x,y) = y
- Computing head(qsort[1000,999..1]) is faster than qsort[1000,999..1]
- Compare time of computations of:
  - fib 32
  - dble (fib 32) where dble x = x + x
- Computations based on *graph reduction*.

Lazy Lists

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fibList = f 1 1
  where f a b = a : f b (a+b)
fastFib n = fibList!!n

fibs = 1:1:[ a+b | (a,b) <- zip fibs (tail fibs)]

primes = sieve [ 2.. ]
       where
         sieve (p:x) = p :
              sieve [ n | n <- x, n `mod` p > 0]
```


Call-by-need

- Efficient implementation of call-by-name (Algol 60)
- If purely functional language then only evaluate expression at most once, because can never change.
- Use graph instead of tree

Monads

Toward Monads

- data Maybe a = Nothing | Just a deriving (Eq, Show)
  - Useful for computations that may not have a result
  - Look up a phone number for a person.
- Hard to work with type as values "wrapped"

Maybe Monad

- dormRooms =[("Jack",10),("Jill",20),("Ann",20)]
- phonesForRooms = [(10,23434),(20,23435),(30,23438)]
- getDormFor name [] = Nothing
  getDormFor name ((nm,rm):rest) = if nm == name
  then Just rm
  else getDormFor name rest
- getPhoneForRoom rm [] = Nothing
  getPhoneForRoom rm ((rmnum,phone):rest) =
  if rm == rmnum then Just phone
  else getPhoneForRoom rm rest

Awkward to Compose

```
- getPhoneForName name rooms phones =
            case getDormFor name rooms of
                Nothing -> Nothing
                Just rm -> getPhoneForRoom rm phones
```

- Must unwrap values to use and then rewrap
- Easier:

```
- getPFN name rooms phones =
        do rm <- getDormFor name rooms
           num <- getPhoneForRoom rm phones
           return num
```