Lecture 3: More Haskell

CSC 131
Spring, 2019

Kim Bruce

Homework Posted

- Turn in using submit.cs.pomona.edu
- For first homework, make sure entire solution can be read into ghci without error.
  - Anything non-executable should be a comment:
    - From -- to end of line or between {- stuff -}
  - *Later teach how to write "literate" Haskell.*

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
    ```

Exercise

- Is listify left or right associative?
  - What is listify (-) 0 [3,2,1]? 2 or -6 or 0 or ???
- How can we change definition to associate the other way?

See built-in foldl and foldr

Folding

- listify is in library as foldr
  - Expands as:
    - foldr f b [a1,...,an] = f a1 (f a2 (... (f an b) ...))
- foldl also exists — it associates from left
  - foldl :: (b -> a -> b) -> b -> [a] -> b
  - foldl _ b [] = b
  - foldl f b (a:as) = foldl f (f b a) as
  - Expands as:
    - foldl f b [a1,...,an] = f (... (f (f b a1) a2) ...) an

Insertion Sort using foldl

```
insert :: [Int] -> Int -> [Int]
insert [] x = [x]
insert (y:ys) x | x < y = x:y:ys
                | otherwise = y:insert ys x

insertionSort :: [Int] -> [Int]
insertionSort = foldl insert []
```

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]
```

Recursive Datatype Examples

- data IntTree = Leaf Integer |
                 Interior (IntTree,IntTree)
                    deriving Show
  - Example values: Leaf 3, Interior(Leaf 4,Leaf -5), ...
- 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)
```

CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

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[5000,4999..1]) is faster than qsort[5000,4999..1}
- Compare time of computations of:
  - fib 32
  - dble (fib 32) where dble x = x + x
- Computations based on *graph reduction*
  - *like tree rewriting, except w/computation graphs - sharing*

Lazy Lists

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

complexity O(fib n) ~ O(2^n)

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

complexity O(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 may evaluate expression at most once, because can never change.
- Hence graph instead of tree works!
  - dble(fib 32)

Monads

The ontological essence of a **monad** is its ***irreducible simplicity***. Unlike atoms, monads possess no material or spatial character. They also differ from atoms by their complete mutual independence, so that interactions among monads are only apparent. Instead, by virtue of the principle of pre-established harmony, **each monad follows a preprogrammed set of "instructions" peculiar to itself, so that a monad "knows" what to do at each moment**.

–wikipedia

Monads

In category theory, a branch of mathematics, a *monad*, or triple is an (endo-)functor, together with two natural transformations. Monads are used in the theory of pairs of adjoint functors, and they generalize closure operators on partially ordered sets to arbitrary categories.

–wikipedia

Start Simpler: Functor

- Modeled on map function on lists

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor ([]) where
  fmap = map
```

  - [] here means operator that takes a type and makes it into a list type
- See later how can use with trees or other structured data

Functor Laws

- fmap id = id   -- 1st functor law
- fmap (g . f) = fmap g . fmap f -- 2nd functor law
  - "." is function composition
  - Can write fmap as an infix operator with <$>
    - Thus fmap f elts = f <$> elts
    - Makes it look more like function application