Lecture 6: More Haskell

CSC 181
Spring, 2016

Kim Bruce

According to Larry Wall (designer of PERL): ... a language by geniuses for geniuses

He's wrong — at least about the latter part though you might agree when we talk about monads

Defining New Types

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

  See more complex examples later

Type Classes Intro

- Specify an interface:
  - class Eq a where
    ```
    (==) :: a -> a -> Bool    -- specify ops
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)     -- optional implementations
    x /= y = not (x == y)
    ```
  - data TrafficLight = Red | Yellow | Green
    instance Eq TrafficLight where
    ```
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
    ```

Common Type Classes

- Eq, Ord, Enum, Bounded, Show, Read
  - See http://www.haskell.org/tutorial/stdclasses.html
- data defs pick up default if add to class:
  - data ... deriving (Show, Eq)
- Can redefine:
  - instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"

More Type Classes

- class (Eq a) => Num a where ...
  - instance of Num a must be Eq a
- Polymorphic function types are often prefixed w/type class specification
  - test x y = x < y *has type* (Ord a) => a -> a -> Bool
  - *Can be used w/ x, y of any Ord type.*
- *More later ...*
  - *Error messages often refer to actual parameter needing to be instance of a class -- to have an operation.*

Higher-Order Functions

- Functions that take function as parameter
  - Ex: map:: (a → b) → ([a] → [b])
    - map double [1,2,3,4,5]
  - filter:: (a → Bool) → ([a] → [a])
    - filter isEven [1,2,3,4,5]
    - filter (\n -> n `mod` 2 == 0) [1,2,3,4,5]
- Comprehension syntax
  - [double n | n <- [1,2,3,4,5], isEven n]

Higher-Order Functions

- any, all:: (a -> Bool) -> [a] -> Bool where
  - any p lst = or (map p lst) *all?*
  - any p = or . map p *alternative def*
- 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]
```

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)

Written like grammar productions — not an accident!!

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[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]
```

Monads Later

- Because Haskell is a purely functional language no function can have a side effect.
- Unfortunately input and output is a side effect period
- To cope with input and output Haskell has a new language construct known as a monad.
- We will discuss monads later in the course.

Formal Syntax and Propositional Logic

A Fragment of English

- S → NP VP
- NP → Snow White | Alice | Dorothy | Goldilocks | DET CN | DET RCN
- DET → the | every | some | no
- CN → girl | boy | princess | dwarf | giant | sword | dagger
- RCN → CN that VP | CN that NP TV
- VP → laughed | cheered | shuddered | TV NP | DV NP NP
- TV → loved | admired | helped | defeated | caught
- DV → gave

Derivation

- S ⇒ NP VP ⇒ Snow White VP
  ⇒ Snow White TV NP
  ⇒ Snow White admired NP
  ⇒ Snow White admired DET CN
  ⇒ Snow White admired the CN
  ⇒ Snow White admired the dwarf
- Draw parse tree
- Every girl admired the dwarf.

In Haskell

```
data Sent = Sent NP VP deriving Show
data NP   = SnowWhite | Alice | Dorothy | Goldilocks
          | NP1 DET CN | NP2 DET RCN deriving Show
data DET  = The | Every | Some | No  deriving Show
data CN   = Girl  | Boy  | Princess | Dwarf | Giant
          | Sword | Dagger deriving Show
data RCN  = RCN1 CN That VP | RCN2 CN That NP TV
          deriving Show
data That = That deriving Show
data VP   = Laughed | Cheered | Shuddered
          | VP1 TV NP | VP2 DV NP NP deriving Show
data TV   = Loved  | Admired | Helped | Defeated | Caught
          deriving Show
data DV   = Gave deriving Show
```

Example

- More details in file FSynF.hs
- "Snow White admired the dwarf" *becomes*
  - s:: Sent
  - s = Sent SnowWhite (VP1 Admired (NP1 The Dwarf))
- We will show later how to parse a sentence into a Haskell formula.

Questions?