cs96si; Assignment 2b; Functional Word Ladder

Joseph Victor

June 4, 2012

Introduction

The goal of this assignment is to get practice using monads, in particular, the State monad. You will first write a general BFS (breadth first search) using a special version of State you create. Then you will write functions to build a word ladder of english words. Finally, you will write a short IO program to interact with the user.

Remind me, what is a monad again?

A monad is, first off, a functor. This means you have the following function:

```
fmap :: (Functor f) => (a -> b) -> (f a) -> (f b)
```

Which lets you take the "result" of a monad and apply a purely functional function to it. More generally, it lets you upgrade a function of type $a \to b$ to one of $F(a) \to F(b)$, in a way which should be completely "natrual".

A monad, however, has two more functions. They are

```
return :: (Monad m) => a -> m a
 (>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

Return is just type upgrading (always possible, type downgrading is usually harder), while »=, pronounced "bind", is kind of like fmap, but instead the function being mapped returns a monad (and the arguments are in the other order). Bind takes something which is already a monad, gets the value "out of" the monad, and applies the function to get a new monadic value. For instance, to get a line and then print the result back out with the IO monad, one could write:

```
parrot :: IO String
parrot = getLine >>=
         \ln -> putStrLn ("You just said " ++ ln) >>=
         \_  -> putStrLn "BAAAAAK.  I'm a parrot" >>=
         \_ -> return ln
```

getLine has type IO String, and the λ in the second line has type String -> IO (). The next »= takes the result of the IO (), the (), and does another IO (). Finally, the third line takes the String from the original getLine and upgrades it to an IO String with return. If this upgrade didn't happen, you coudln't use a bind. The types make sure you can't leak something of type String after doing IO stuff.

When using »=, it is advised (for style) that each call to bind is on a sperate line. See the notes.hs for an example of nice style.

Anyhow, thats enough review. Lets write some functions.

Getting started

The starting code imports some modules, and has some weird looking data declarations for Queue. Ignore, just know that you can use the three methods provided. What they do should be obvious. We will be using a Queue and a Set for this. The interface for a Set is linked on the website. Notice that anything you do requres the word Set infront of it, for namespace reasons. For instance, if you want to see if x is a member of a set s, then you would say

```
Set.member x s
```

There is a thing called wordsSetIO, which we'll talk about later.

The last thing in the starter file is a type declaration for BFSState. This is the monad you will be using.

Recall how a BFS could work. You have a queue of lists of the type through which you are searching, and a set of things you've already seen. This is the "State" of the algorithm. If the queue is ever empty at the top of the loop, the search fails. Each time through the loop, you pop a path from the queue, see what all the possible next states are, and enqueue each one which has not already been seen, that is, is not in the "seen" set. In WordLadder, the next states are words with one letter different than the previous word, subject to the condition they are English words. You terminate when some condition is fullfilled, returning the list you had built up.

One important thing to notice is the following: the Queue is a Queue of [a], while the set is a Set a. That is, in the case of word ladder, you enqueue entire ladders, but the things you have seen before are individual words.

We are now ready to write something.

BFSState functions

We want some functions that work on BFSStates. Two in particular. The first is

```
bfsEnqueue :: (Ord a) => [a] -> BFSState a ()
```

What this does is this: let the list argument be (x:xs), having type [a]. The return type BFSState a (), meaning it does something with the state and then "returns" (). The state, recall, is a tupe (Queue [a], Set a). We want this function to, if x is not in the set, push (x:xs) onto the queue and add x to the set, else do nothing.

Second you want to write

```
bfsDequeue :: BFSState a (Maybe [a])
```

What this does if, if the queue in the state is empty, returns Nothing, else returns Just the top list in the queue, while modifying the state so that the top thing is no longer there.

Recall that you have two functions, get and put.

```
get :: BFSState a (Queue [a],Set a)
-- you can use get like this:
blah = get >>=
       \(q,s) -> -- something of type (BFSState a b), where b is any type

put :: (Queue [a], Set a) -> BFSState a ()
-- given a queue and a set, say, (q,s), put (q,s) has type (BFSState a ())
-- the following function does nothing to the state,
--   but is an example of using put and get
stupid = get >>=
         \(q,s) -> put (q,s)
```

You do not need to go back to the deffinition of State to do this.

You are now ready to write BFS.

```
breadthFirstSearch :: (Ord a) => (a -> Bool) -> (a -> [a]) -> a -> Maybe [a]
```

Where the (a -> Bool) is a function which returns true when you have found your target, the (a -> [a]) is a function which returns the list of "next states" from the previous one. For instance, in WordLadder when a is String, this function will return a list of all Enlish words one character different from the input.

The function should look like this

```
breadthFirstSearch winFun nextsFun x = evalState bfs (enqueue [x] emptyQueue,
                                                      Set.singleton x)
  where bfs :: BFSState a (Maybe [a])
        bfs = -- the bfs algorithm.
```

Since bfs has the type it does, you can call it on the right side of a »=, so you can think of this as your main loop. Recall the algorithm

```
while(true){
 if(the queue is empty) fail;
 pop the queue, let (x:xs) be the thing you got out;
 if(winFun(x)){
   return (x:xs);  //or reverse em if you'd like
 zs = nextsFun(x);
 for (z in zs){   // mapM_ maybe?
   enqueue (z:x:xs)
 }
 continue;  // aka, start again at the top, aka, call bfs
}
```

Now do that, except using cute monads. Remember, you can still ise if-then-else and case statements.

Congrats, you now have a functional monadic BFS algorithm. yaaaay.

Word Ladder

The last part is easy, make it work with Word Ladder. Write

```
wordLadderNext :: String -> [String]
```

This returns a list of all strings one letter different than the input word.

Now, write word ladder, given an english dictionary. That is

```
wordLadder :: Set.Set String -> String -> String -> Maybe [String]
wordLadder dict a b = breadthFirstSearch (==b)
                        (\wrd -> filter ((flip Set.member) dict)
                                        (wordLadderNext wrd))
                        a
```

That is, given a dictionary, a start word and an end word, this will find a word ladder.

Now you just need a dictionary. Luckily, the IO (Set String) wordsSetIO is just that. It is type IO (Set String) because it needs to look in a file to find the list of words. Write the function main of type IO (). This should open the dict, talk to the user, ask for two words, attempt to find a word ladder, and print it out if it finds it or else say it couldn't find it.

And thats it. How simple.

Deliveriables

This is due the last day of finals. Shoot me an email if you are having trouble. OH will be posted on the website soon. I want this file, and whatever else, especially comments about the assignment. Have fun :).