Assignment 5: Continuations and Monads

CS 442/642

Due March 24th in class

For this assignment, you will use ML to complete part A, Haskell to complete part B, and either Scheme or Haskell to complete part C.

Part A—Continuations

On Assignment 1, you were asked to code the Scheme function `exists` using `foldl` or `foldr`. You were then presented with the following, direct implementation of `exists`:

```
(define exists (lambda (p L)
   (if (null? L) #f
       (if (p (car L)) #t (exists p (cdr L)))
   )
))
```

Most of you correctly observed that the direct version of `exists` is more efficient because it stops traversing the list as soon as it finds an element that satisfies `p`, whereas the version based on folding always traverses the entire list. For this question, you will correct this inefficiency inherent in folding by implementing a short-circuiting version of foldr, in ML, which you will call `scfoldr`. This function will take three parameters: a function `f`, a default value `i`, and a list `l`, as always. However, the function `f` will now be a ternary function, whose third parameter is a continuation `k` to which it can throw a value if you wish to abort computation immediately and return an answer. `scfoldr` will arrange that when `f` is called, it will be passed the continuation of `scfoldr` itself, so that throwing to this continuation will transfer control all the way out of `scfoldr`.

Implement `scfoldr` in ML, give its type (you can just report what SML/NJ returns, but make sure it makes sense to you!), and then use `scfoldr` to implement a short-circuiting version of `exists`.

Part B—Monads

1. Write a Haskell program to play a simple guessing game. You will think of a number. Your program will ask for a guessing range, and then repeatedly guess numbers in that range until it either guesses the number, or can prove that you are cheating. Each time your program guesses a number, you must either answer `yes`, `higher` or `lower`. A sample interaction follows (your responses shown after the colon(:)):

```
Enter guessing range: 1 10
Is it 5?: higher
Is it 8?: lower
Is it 6?: yes
Got it!
```

If the guessing range is down to one possibility, and you don't answer `yes`, then your program should output `Cheating!` instead of `Got it!`.

Your program should employ binary search (with halves rounded down) on the guessing range to output an answer or accuse you of cheating in $O(\log n)$ time, where n is the size of the guessing range.

Your guessing game should be called `guess`, and its type should be `IO ()`.

2. The following imperative ML program produces the first n elements of the Fibonacci sequence:

```
val current = ref 0
val next = ref 1

fun fib 0 = []
|   fib n =
  let val answer = !current
  in
     (current := !next;
      next := !current + answer;
      answer :: fib (n - 1))
  end
```

Rewrite this function in Haskell, using the State monad from class.

Hints: The state you maintain should be the counters `current` and `next`, so your state type should be `(Int,Int)`. Use the examples from class (especially the one that counts nodes in a tree) as inspiration. Your `fib` function will have type `Int -> [Int]`, but it will likely call a helper function whose type will be `Int -> State (Int, Int) [Int]`. Your solution should start with the definition of the State monad, including the instance declaration, as given in class. You should not need to modify this part, and you can paste it in from the tree-labelling example that was posted on Piazza.

Part C—Unification revisited

Even moreso than ML and Haskell, the Prolog programming language relies fundamentally on unification. Unification in Prolog occurs in a more general setting than what we have seen so far. Whereas types consist of a small number of constructors applied to one or two arguments, the terms we unify in Prolog consist of function symbols ("functors") applied to arguments. For example:

$$f(X_1, a, b, X_2)$$

Here, f is the function symbol, with arity 4, X_1 and X_2 are variables, and a and b are constants. To simplify the presentation, we may regard constants like a and b as simply function symbols with arity 0.

In order for two terms to be unified, their function symbols must match and have the same arity, and then corresponding arguments must match (note that we do *not* permit variables to act as function symbols). Full details of the unification algorithm can be found in Chapter 8 of your course notes. The algorithm in the notes omits the occurs-check; however, you must perform it.

In this question, you will implement unification in this most general setting. You have two choices for your implementation: you may solve the problem either in Scheme or in Haskell. If you solve the problem in Scheme, you must use `call-with-current-continuation`. If you solve the problem in Haskell, you must use at least one monad in your solution.

If you wish to solve the problem in Haskell, use the following declarations:

```
data Term = T String [Term] | V String deriving (Eq, Show)
```

```
-- mgci = most general common instance, i.e., what do the types unify to?
mgci :: Term -> Term -> Maybe Term
```

If you solve the problem in Scheme, you must also provide the function `mgci`, as given above. In both cases, this function will probably call your `unify` function as a helper. The type of `unify` is not prescribed, so as to give freedom in how you choose to implement it.

Submission

Submit electronically:

- Part A solution (`a5a.sml`).
- Part B solution (`a5b1.hs`, `a5b2.hs`).
- Part C solution (`a5c.scm` or `a5c.hs`).

Submit to Crowdmark:

- All source code.

The relative weighting of the components of the assignment is roughly 20% for part A, 40% for part B, and 40% for part C.