Functional programming, with and without the `IO` monad

COMP 150 - Applied Functional Programming

October 9, 2012

DVD Packing

(1) What were your best results packing DVDs, and how did you get them?

(2) Can you imagine multiple strategies for packing DVDs? How would you code them (preferably in a modular way)?

(3) In what ways does your solution to the DVD problem exploit laziness and higher-order functions for modularity?

(4) What are the *types* of the functions you define, and what, if anything, can you learn from looking at the types?

Randomness

(5) Suppose I give you a coin which, when flipped, lands heads with probability p (you choose p). Could you use such a coin to affect the results of a DVD packing? How?

(6) How would you use the coin to make a DVD packing *better*?

Monadic programming

(7) Back to the coin you can flip. Suppose it is embedded in the IO monad. What type of Haskell value represents the coin?

(8) Suppose you're given a random-number generator that produces a random `Double` between 0.0 and 1.0.

(a) If it is a pure function, what are some possible types?

(b) If it is in the I/O monad, what is the only reasonable type?

(c) How would you use the monadic version to implement your monadic coin flip?

(9) When we look at the interaction of lists and functions, we're able to derive such higher-order functions as `map`, `filter`, `exists`, and `foldr`. What if you look at the interaction of the I/O monad and functions? What higher-order functions can you derive? What are their implementations? [1]

Hint: At minimum, you should come up with something that can help with the coin problem.

[1] Note on notation: in today's haskell, `unitIO` is now called by the name `return` (Sorry, M@), and `bindIO` is now called by the infix name `(>>=)`. There have been minor changes in the other names as well.

(10) Suppose you mix lists into the picture? Any more higher-order functions? If the lists might not be finite, does it matter? (*Hint:* Finiteness matters for `length`, `reverse`, and `(++)`, but not for `take`, `drop`, or `takeWhile`.)

(11) John von Neumann invented many great mathematical tricks. In one of these tricks, he turns a biased coin ($p \neq \frac{1}{2}$) into a fair coin ($p = \frac{1}{2}$) by considering consecutive flips: If two consecutive flips are identical, ignore them; if they're different, take the second one.

(a) How would you program this algorithm using the I/O monad?

(b) Let's assume that you have written von Neumann's algorithm as a pure function of type `Bool -> Bool -> Maybe Bool`. What higher-order functions do you need, and how do you use them, to "lift" this pure computation into the IO monad?

Hint: You might find it useful to know about

```
catMaybes :: [Maybe a] -> [a]
```

(c) Assuming you're given a biased coin and you want to convert it to a fair coin. With what Haskell type or types would you find it most convenient to package this algorithm?

Random searches

(12) Let's return to the question of using randomness to improve DVD packing. What sort of randomizing transformation might result in a better packing?

(13) In class I'll describe a randomization technique invented by Michael Mitzenmacher and his colleagues at Mitsubishi research labs. You'll implement it later. Meanwhile, what *type* do you think it has?

Consolidation

(14) Summarize the "reusable glue" you've discovered around the IO monad and higher-order functions, both random and otherwise. Consider what operations in the `IO` monad you actually had to use.