Quiz (Week 7)

Effects and Purity

Question 1

Which of the following C functions would be considered *pure*?

1. ✔ `sqrt()`
2. ✗ `printf()`
3. ✗ `rand()`
4. ✔ `strcmp()`

Computing a square root is pure, as the result depends solely on the input to the function. Indeed, the square root is a function in the mathematical sense and therefore is, by definition, pure.

The `printf()` function is not pure as it performs I/O, a type of effect.

The `rand()` function is not pure as each time it is evaluated it can return different results. That is, the results are not dependent *solely* on the input arguments.

The `strcmp()` function is pure as it returns a comparison result based solely on the two input strings, and it doesn't change those strings or any other data in any way. Thus it can be expressed as a mapping from inputs (two strings) to outputs (booleans).

Question 2

Imagine we had a function `adder` that added on to a running total each time it was called:

```
*> adder 7
7
*> adder 10
17
*> adder 0
17
*> adder 7
24
```

Why is `adder` impure?

1. ✗ It performs I/O
2. ✗ It manipulates memory
3. ✔ It does not depend solely on its arguments
4. ✗ It doesn't indicate effects in its type

The `adder` function does not perform I/O, it *does* manipulate memory (but so do pure functions), and its type is neither here nor there. The thing that makes it impure is that the expression `adder x` does not always mean the same thing for a given `x`. That is, it depends on some internal state *in addition to* its argument. Thus option 3 is the correct answer.

Question 3

Which of the following effects is considered an *internal* effect?

1. ✗ Modifying global variables
2. ✗ Drawing on the screen
3. ✔ Modifying local variables
4. ✔ Allocating a data structure
5. ✗ Throwing an exception

Modifying global variables can have a non-local influence on other parts of the program, therefore is not internal. Drawing on the screen similarly is not internal as its effect can clearly be observed from outside the function. Modifying local variables is internal as no other part of the program can observe the modification (neither can the user). Allocating data structures is also considered internal (under the common abstraction that we have infinite memory) as such an allocation also cannot be observed externally. Throwing an exception can be observed externally, however (by catching it), and thus is not an internal effect.

Question 4

What does the type `IO Int` signify?

1. ✗ An embedded program that may perform side effects before returning an `Int`
2. ✗ A function from the abstract state of the `RealWorld` to a pair of the `RealWorld` state and an `Int`.
3. ✗ An effectful computation that produces an `Int`.
4. ✔ All of the above views are valid interpretations

GHC internally models `IO a` as being the same as a type:

```
IO a =~ RealWorld -> (RealWorld, a)
```

This is a common view of `IO` and option 2, but perhaps a more common view of `IO` is that it denotes *embedded programs*. For example,

```
getChar :: IO Char
```

Could be viewed as a type representing an (effectful) *program* that will *produce* a `Char` when executed. This helps us to view `IO` as a type just like any other, and one that we can pass into and return from functions just like `Maybe Int` or `[Int]` .

IO and State

Question 5

Imagine we had the following `IO` based API for manipulating a robot:

```
data Direction = L | R
forward    :: IO ()
obstructed :: IO Bool
turn       :: Direction -> IO ()
```

We wish to write a program that will move `forward` unless `obstructed`, in which case the robot should `turn` towards the `L` direction.

Which of the following is a type-correct implementation of the above procedure?

1. ✔

```
robot = do
  sensed <- obstructed
  if sensed
    then turn L
    else forward
  robot
```

2. ✗

```
robot = do
  if obstructed
    then turn L
    else forward
  robot
```

3. ✗

```
robot = do
  let sensed = obstructed
  if sensed
    then turn L
    else forward
  robot
```

4. ✗

```
robot = do
  sensed <- obstructed
  if sensed
    then turn L
         robot
    else forward
         robot
```

Option 1 is correct. Option 2 uses an `IO Bool` (`obstructed`) where a `Bool` is required (in the `if`). Option 3 uses `let` to bind `sensed` to `obstructed` . That is, `sensed` now has type `IO Bool` , which once again is incorrectly used within the `if` . Option 4 places the `robot` looping call at the same indentation as `turn L` and `forward` , but without the `do` keyword they do not form a block and so Haskell would parse this as `turn L robot` which is not well-typed.

Question 6

Check all of the following programs that are equivalent to the `IO` action `a` :

```
a = do x <- getLine
       putStrLn (filter isDigit x)
       a
```

1. ✔ `a = getLine >>= putStrLn . filter isDigit >> a`
2. ✔ `a = getLine >>= \x -> putStrLn (filter isDigit x) >>= \_ -> a`
3. ✗ `a = (getLine >>= \x -> putStrLn (filter isDigit x)) >>= a`
4. ✗ `a = do getLine >>= \x -> putStrLn . filter isDigit; a`

5. ✗ `a = do x <- getLine; putStrLn . filter isDigit; a`
6. ✔ `a = do x <- getLine; putStrLn . filter isDigit $ x; a`
7. ✔ `a = do getLine >>= \x -> putStrLn (filter isDigit x); a`

Options 3,4, and 5 don't type-check. The others are all equivalent.

Question 7

Below is an example of a small program using `State String`. As a refresher, here's the basic API for `State`:

```
get :: State String String
put :: String -> State String ()
runState :: State String a -> String -> (String, a)
```

Now, our program will repeatedly pad a string with spaces until it reaches a certain length:

```
leftPad :: Int -> State String ()
leftPad l = while ((< l) . length) $ do
              str <- get
              put (' ':str)
```

What is the type of `while` in this example?

1. ✗ `Bool -> State String () -> State String ()`
2. ✗ `State String Bool -> State String () -> State String ()`
3. ✗ `(Bool -> State String ()) -> State String ()`
4. ✗ `(String -> Bool) -> State String ()`
5. ✔ `(String -> Bool) -> State String () -> State String ()`

The `while` loop takes a state-dependent conditional, i.e a function that returns a `Bool` for a given `String`, and a stateful monadic action for the loop body, `State String ()`, finally producing a stateful monadic action that runs the loop, `State String ()`, hence option 5 is correct.

Question 8

Here is a program to detect if a string has balanced parentheses, ignoring all other characters.

```
matching :: String -> Int -> Bool
matching []       n = (n == 0)
matching ('(':xs) n = matching xs (n+1)
matching (')':xs) n = n > 0 && matching xs (n-1)
matching (oth:xs) n = matching xs n
```

Which of the following is an accurate translation of the above program to use the `State` monad?

1. ✗

```
matching xs = snd (runState (go xs) 0) == 0
  where
    go [] = pure True
    go (x:xs) | x == '('  = modify (+1) >> go xs
              | x == ')'  = modify (-1) >> go xs
              | otherwise = go xs
```

2. ✗

```
matching xs = snd (runState (go xs) 0) == 0
  where
    go [] = pure True
    go (x:xs) | x == '('  = modify (+1) >> go xs
              | x == ')'  = do n <- get
                               if n > 0 then put (n - 1) >> go xs
                                        else pure False
              | otherwise = go xs
```

3. ✗

```
matching xs = fst (runState (go xs) 0)
  where
    go [] = pure True
    go (x:xs) | x == '('  = modify (+1) >> go xs
              | x == ')'  = do n <- get
                               if n > 0 then put (n - 1) >> go xs
                                        else pure False
              | otherwise = go xs
```

4. ✗

```haskell
matching xs = let (b,n) = runState (go xs) 0
               in b && n == 0
  where
    go [] = pure True
    go (x:xs) | x == '('  = modify (+1) >> go xs
              | x == ')'  = modify (-1) >> go xs
              | otherwise = go xs
```

5. ✔

```haskell
matching xs = fst (runState (go xs) 0)
  where
    go [] = get >>= pure . (== 0)
    go (x:xs) | x == '('  = modify (+1) >> go xs
              | x == ')'  = do n <- get
                                if n > 0 then put (n - 1) >> go xs
                                         else pure False
              | otherwise = go xs
```

Option 1 checks if the final count is zero, but does not check if the count sinks below zero at any point, matching the strings `)(` for example. Option 2 does check if the count drops below zero, but then doesn't do anything with that information. Option 3 only checks if the count drops below zero, and not that the count is zero at the end. Option 4 does check both the boolean and the count at the end, however does not set the boolean to false when the count drops below zero. Option 5 does all the required checks and is therefore correct.

Submission is already closed for this quiz. You can click here to check your submission (if any).