Functional Programming, FPR

November 4, 2014

Mayur Pant

20 pages

Contents

Summary of contents
For clarity and flow, I have separated the solution into two sections (which reflects the order in which I approached the questions).

Section 1 contains the questions that create the turtle programs.
Section 2 contains the questions for the shallow embeddings.
Section 3 contains my review of what I learned from each question.

1 Questions 1-3, 5, 8, 11
e-copy: Questions_1_3_5_8_11.lhs

Questions 1-3, 5, 8, 11
=======================

```
> module Questions_1_3_5_8_11 (Dir, succ_wrap, forwards') where
```

To prevent duplication, Questions_4_6_7_9_10.lhs imports a data type and two functions.

Introduction
============

```
> data Prog = Move Prog
>           | Turn Prog
>           | SwapPen Prog
>           | Stop


> snail = SwapPen (Move (Turn (Move (Turn (Move (Move (Turn (Move (Move (Turn (Move (Move
(Stop))))))))))))


> data Dir = North | West | South | East deriving (Show, Enum, Eq, Bounded)
```

In function dir of question 2 we require Dir to be part of the Show typeclass.
In function dir of question 2 we require Dir to be part of the Enum typeclass.
In function forwards' of question 3 we require Dir to be part of the Eq typeclass.
In function succ_wrap of question 5 we require Dir to be part of the Bounded typeclass.

1.
==

```
> type PenDown = Bool
> penDown :: Prog -> PenDown -> PenDown
> penDown (Stop) b = b
> penDown (SwapPen x) b = penDown x (not b)
> penDown (Move x) b = penDown x b
> penDown (Turn x) b = penDown x b
```

By deriving "Eq" from Prog we could have instead used guards to perform equality tests for Stop and SwapPen, and then merged Move and Turn into a single otherwise clause; see function forwards' (q. 3) for an example of shorter code with guards.

Reducing the last two lines with an otherwise clause would make the code faster, but I have illustrated every pattern here for clarity.

2.
==

```
> dir :: Prog -> Dir -> Dir
> dir (Stop) d = d
> dir (Turn x) East = dir x North
> dir (Turn x) d = dir x (succ d)
> dir (Move x) d = dir x d
> dir (SwapPen x) d = dir x d
```

Here we need datatype Dir to be part of the Enum typeclass so we can iterate through Dir with function succ (module Prelude), and we derive Show so we can display the result in our tests.

Since function succ does not wrap around we include an exception for East (we could have written an alternate succ function that does wrap - see http://stackoverflow.com/questions/5684049/is-there-some-way-to-define-an-enum-in-haskell-that-wraps-around, answer by "T_S_"). We utilize this in q. 5 (function succ_wrap), which we export in our module as well.

3.
==

```
> type Pos = (Integer, Integer)
> forwards :: Dir -> Pos -> Pos
> forwards North (x, y) = (x, y+1)
> forwards West (x, y) = (x-1, y)
> forwards South (x, y) = (x, y-1)
> forwards East (x, y) = (x+1, y)
```

Alternative:
If datatype Dir is part of the Eq typeclass, we can save keystrokes by using guards (and could have used keyword "where" to make aliases like "oneUp", "oneLeft", etc. - we use "where" in function plot' of q. 8).

```
> forwards' :: Dir -> Pos -> Pos
> forwards' d (x, y)
>   | d == North = (x, y+1)
>   | d == West  = (x-1, y)
>   | d == South = (x, y-1)
>   | d == East  = (x+1, y)
```

5.
==

```
succ_wrap :: (Bounded a, Enum a, Eq a) => a -> a
```

We read this type signature as follows: succ_wrap's input and output values are of the same type, and must be a member of the Bounded, Enum and Eq classes (our 3 class constraints).
Terminology source: http://learnyouahaskell.com/types-and-typeclasses#typeclasses-101
(1988) R. Bird, Introduction to Functional Programming using Haskell, Ch. 2.1.1, p.32

Let's consider the definition for parametric polymorphism vs. ad hoc polymorphism:
"The idea that something is applicable to every type or holds for everything is called universal quantification. In mathematical logic, the symbol .. an upside-down A, read as "forall" .. is commonly used for that, it is called the universal quantifier .. parametric polymorphism = ignorant of the type actually used. => \forall" http://en.wikibooks.org/wiki/Haskell/Polymorphism#Parametric_Polymorphism

"Parametric polymorphism refers to when the type of a value contains one or more (unconstrained) type variables .. Since a parametrically polymorphic value does not "know" anything about the unconstrained type variables, it must behave the same regardless of its type."

"You can recognise the presence of ad-hoc polymorphism by looking for constrained type variables: that is, variables that appear to the left of =>, like in elem :: (Eq a) => a -> [a] -> Bool."
http://www.haskell.org/haskellwiki/Polymorphism#Ad-hoc_polymorphism

We say that succ_wrap is an ad-hoc polymorphic function as it is constrained to 3 classes.

```
> succ_wrap d | d == maxBound = minBound
>             | otherwise = succ d
```

We use this function in preference to succ because we want datatype Dir to wrap around. Source: See discussion of q.2.

```
> pos :: Prog -> Pos -> Dir -> Pos
> pos (Stop) p d = p
> pos (Move x) p d = pos x (forwards' d p) d
> pos (Turn x) p d = pos x p (succ_wrap d)
> pos (SwapPen x) p d = pos x p d
```

8.
==

```
> type LineSegment = (Pos, Pos)
> type Picture = [LineSegment]

> plot :: Prog -> Pos -> Dir -> PenDown -> Picture
> plot (Stop) p d b = []
> plot (Move x) p d True = (p, forwards' d p):plot x (forwards' d p) d True
> plot (Move x) p d False = plot x (forwards' d p) d False
> plot (Turn x) p d b = plot x p (succ_wrap d) b
> plot (SwapPen x) p d b = plot x p d (not b)
```

Alternatives:
We can see 2 pattern match lines for (Move x) which are related - we can make their relationship clearer by merging them into one line by using a "case" pattern match:

```
> plot' :: Prog -> Pos -> Dir -> PenDown -> Picture
> plot' (Stop) p d b = []
> plot' (Move x) p d b = case b of True -> (p, next_p):next_x
>                                  False -> next_x
>                         where next_p = forwards' d p
>                               next_x = plot' x next_p d b
> plot' (Turn x) p d b = plot' x p (succ_wrap d) b
> plot' (SwapPen x) p d b = plot' x p d (not b)
```

... or instead we could take advantage of Bool deriving Eq to test the boolean PenDown with guards:

```
> plot'' :: Prog -> Pos -> Dir -> PenDown -> Picture
> plot'' (Stop) p d b = []
> plot'' (Move x) p d b
>              | b == True = (p, next_p):next_x
>              | b == False = next_x
>              where next_p = forwards' d p
>                    next_x = plot'' x next_p d b
> plot'' (Turn x) p d b = plot'' x p (succ_wrap d) b
> plot'' (SwapPen x) p d b = plot'' x p d (not b)
```

... yet since we are testing a boolean with one of only 2 outcomes we can make it shorter and clearer still by using an if..else statement:

```
> plot''' :: Prog -> Pos -> Dir -> PenDown -> Picture
> plot''' (Stop) p d b = []
> plot''' (Move x) p d b = if b == True then (p, next_p):next_x else next_x
>                        where next_p = forwards' d p
>                              next_x = plot''' x next_p d b
> plot''' (Turn x) p d b = plot''' x p (succ_wrap d) b
> plot''' (SwapPen x) p d b = plot''' x p d (not b)
```

11.
===

```
> data XML = Element String [Attr] [XML]
> type Attr = (String, String)
```

First off we utilize some existing functions that help us to render the XML datatype:
Source: http://www.macs.hw.ac.uk/~dsg/events/ISS-AiPL-2014/materials/Gibbons/Shapes-complete.lhs

---

```
> instance Show XML where
>  show (Element n as []) = element n as
>  show (Element n as xs) = open n as ++ unlines (map show xs) ++ close n
```

Our XML datatype defines an element in the syntax:
Element "name" [("attribute name"),("attribute value")] []   for leaf elements
Element "name" [("attribute name"),("attribute value")] [child Element] for nesting elements

So instance declares two rendering functions for these two possible syntaxes.

For leaf elements (an element with no children): render a string which is an open tag, name, any attribute pairs (function attrs) and closed tag.

```
> element :: String -> [Attr] -> String
> element n as = "<" ++ n ++ attrs as ++ "/>"
```

For elements with children: open the tag, show the name, the list of nested element(s) (unlines (map show xs)), and close the tag.

```
> open :: String -> [Attr] -> String
> open n as = "<" ++ n ++ attrs as ++ ">\n"

> close :: String -> String
> close n = "</" ++ n ++ ">"

> attrs :: [Attr] -> String
> attrs as = concat [ " "++k++"="++ show v | (k,v)<-as ]
```

Useful! Now we create a function picture_lines that traverses our Picture (a list of line segments) and produces a list of Elements.

Given we are traversing an input list to generate an output list a fold sounds applicable. We have opted for a right fold because it allows us to accumulate elements to the head using (:), rather than adding them to the tail with (++) which is way more expensive ("Haskell has to walk through the whole list on the left side of ++ .. putting something at the beginning of a list using the : operator .. is instantaneous", source: http://learnyouahaskell.com/starting-out#an-intro-to-lists).

```
> picture_lines :: Picture -> [XML]
> picture_lines = foldr (\((x1, y1), (x2, y2)) acc -> (Element "line" [("x1", show $ x1*100), ("y1",
show $ y1*100), ("x2", show $ x2*100), ("y2", show $ y2*100)] []) : acc) []

> svg :: Picture -> XML
> svg p =  Element "svg" [("width","204"), ("height","204"), ("viewBox", "-102,-102,204,204"),
("xmlns", "http://www.w3.org/2000/svg"), ("version", "1.1")] [
>          Element "g" [("transform","scale(1,-1)"), ("stroke-width","4"), ("stroke-linecap","round"),
("stroke","black"), ("fill","none")] (
>          picture_lines p ) ]
```

Throughout this assignment I have provided multiple versions of functions, where the final version is the most efficient.
Let's try it with our 4 versions of plot, using the favourite version last:

```
$ writeFile "snail-1.svg" (show (svg (plot snail (0, 0) North False)))
$ writeFile "snail-2.svg" (show (svg (plot' snail (0, 0) North False)))
$ writeFile "snail-3.svg" (show (svg (plot'' snail (0, 0) North False)))
$ writeFile "snail-4.svg" (show (svg (plot''' snail (0, 0) North False)))
$ diff --report-identical-files --from-file snail-{1..4}.svg model_answer.svg
Files snail-1.svg and snail-2.svg are identical
Files snail-1.svg and snail-3.svg are identical
Files snail-1.svg and snail-4.svg are identical
Files snail-1.svg and model_answer.svg are identical
```

2 Questions 4, 6, 7, 9, 10
e-copy: Questions_4_6_7_9_10.lhs

Questions 4, 6, 7, 9, 10
========================

```
> import Questions_1_3_5_8_11
```

To prevent duplication we import datatype Dir and functions succ_wrap, forwards'.

4.
==

```
> type Pos = (Integer, Integer)
> type Dir2 = Pos -> Pos

> north, west, south, east :: Dir2

> north (x, y) = (x, y+1)
> west  (x, y) = (x-1, y)
> south (x, y) = (x, y-1)
> east  (x, y) = (x+1, y)
```

We can nest these functions:

```
$ north (north (0,0))
(0,2)
```

A contrast of deep embedding versus shallow embedding is given in the paper "Folding Domain-Specific Languages: Deep and Shallow Embeddings" (2014, J. Gibbons, N. Wu, Source: http://www.cs.ox.ac.uk/jeremy.gibbons/publications/embedding.pdf).
These two contrasts are shown in the file ./deep_vs_shallow.png.
If required, we might imitate the shallow embedding example ad verbatim by providing a function (eval)uation to represent the whole expression and (lit)eral to represent a coordinate:

```
> type Expr = Pos
> lit n = n
> eval :: Expr -> Pos
> eval n = n

$ eval (north (north (lit (0,0))))
(0,2)
```

... this seems optional for illustrating shallow embedding, so we won't perform this additional step with our subsequent questions.

6.
==

Attempt 1: Reverse engineering q. 10

If forwards provides a semantics for language Dir (forwards :: Dir -> Pos -> Pos), such that
type Dir2 = Pos -> Pos,
then if pos provides a semantics for language Prog (pos :: Prog -> Pos -> Dir -> Pos) we conclude
type Prog2 = Pos -> Dir -> Pos.

We have already imported the necessary datatype Dir and its constructors from module
Questions_1_3_5_8_11, so no need to repeat them here.

```
> type A = (Pos -> Dir -> Pos)                            -- pos semantics

> type Prog2 = (A -> A, A -> A, A -> A, A) -> A
> move2, turn2, pen2 :: Prog2  -> Prog2
> stop2 :: Prog2
> move2 x    = \(m, t, p, s) -> m(x (m, t, p, s))
> turn2 x    = \(m, t, p, s) -> t(x (m, t, p, s))
> pen2 x     = \(m, t, p, s) -> p(x (m, t, p, s))
> stop2      = \(m, t, p, s) -> s

$ let program_A = turn2 (move2 (pen2 (move2 (pen2 stop2)))) :: Prog2
```

Attempt 2: researched method

A contrast of deep embedding versus shallow embedding is given in the paper "Functional Programming for Domain-Specific Languages" (2013, J. Gibbons, Source: http://www.cs.ox.ac.uk/jeremy.gibbons/publications/fp4dsls.pdf).
An example of a shallow embedding for type IntegerSet (and its four associated functions) is given (see file ./shallow.png), which is quite similar to our DSL; note the similarity in program expressions, and how 'Empty' is equivalent to our 'Stop'. We can use this as an example.

As in Attempt 1,
If forwards provides a semantics for language Dir (forwards :: Dir -> Pos -> Pos), such that
type Dir2 = Pos -> Pos,
then if pos provides a semantics for language Prog (pos :: Prog -> Pos -> Dir -> Pos) we conclude
type Prog2' = type Pos -> Dir -> Pos.

We have already imported the necessary functions (forwards', succ_wrap) from module
Questions_1_3_5_8_11, so no need to repeat them here. We just need to write some laws:

```
> type Prog2' = Pos -> Dir -> Pos
> move2', turn2', pen2' :: Prog2' -> Prog2'
> stop2' :: Prog2'
> move2' f p d = f (forwards' d p) d
> turn2' f p d = f p (succ_wrap d)
> pen2' f p d = f p d
> stop2' p d = p

$ let program_A' = turn2' (move2' (pen2' (move2' (pen2' stop2')))) :: Prog2'
```

Note here that pen2' does nothing, as the PenDown boolean does not exist in our semantic! I expect that's why the function was referred to as pen2 instead of swapPen2.

You will find additional discussion of why I attempted questions 6 and 9 twice in section "3.2 Review of Section 2".

7.
==

We cannot use Dir2 as a drop in replacement for Dir in our existing functions because they represent two different things.
Whereas Dir represents a stationary direction, the four Dir2 functions represent the coordinate for a step in that direction.
Each Dir2 function in answer 4 is equivalent to the pos function; we are stating the origin and the direction (whether as a parameter or by the function name itself), and returning the final position.

This confuses matters because it means for a program x, we need to state every position we wish to turn in (thereby replacing the turn function, as the robot can now move to any direction from any direction in a single step, rather than 1, 2 or 3 chained turn commands).
Specifically for the replacement in the function pos, it means also that the forwards command is made redundant, since the four Dir2 commands provide the result of forward.

If I was going to provide a shallow embedding of directions I would define a function that emulates function succ_wrap without our existing enumerated datatype Dir, but rather using directions of type String as shown:

```
> dsucc_wrap :: String -> String
> dsucc_wrap d
>   | d == "north" = "west"
>   | d == "west" = "south"
>   | d == "south" = "east"
>   | d == "east" = "north"

$ dsucc_wrap "east"
"north"
```

now I can replace the type signature for function pos as:

```
pos :: Prog -> Pos -> String -> Pos
```

replacing the Turn command pattern match from:

```
pos (Turn x) p d = pos x p (succ_wrap d)
```

to:

```
pos (Turn x) p d = pos x p (dsucc_wrap d)
```

... and perform a similar substitution from datatype Dir to type synonym String in any other functions that pos uses (i.e. function forwards').

I would have made a list, but we can only head and tail (or index, take and drop) on a list ... a list is not enumerated or bounded, so we cannot use function succ as we would on a datatype ("[Haskell-beginners] Just clarifying the "pred" and "succ" functions in Haskell", Source: http://www.haskell.org/pipermail/beginners/2010-February/003449.html). This teaches us the value of creating a datatype is that it allows us to benefit from any functions that belong to the same typeclass (such as function succ for typeclass Enum, or function maxBound for typeclass Bounded).

9.
==

Attempt 1: Reverse engineering q. 10

```
> type PenDown = Bool
> type LineSegment = (Pos, Pos)
> type Picture = [LineSegment]

> type B = (Pos -> Dir -> PenDown -> Picture)     -- plot semantics - similar reasoning to q. 6

> type Prog3 = (B -> B, B -> B, B -> B, B) -> B
> move3, turn3, swapPen3 :: Prog3  -> Prog3
> stop3 :: Prog3
> move3 x    = \(m, t, p, s) -> m(x (m, t, p, s))
> turn3 x    = \(m, t, p, s) -> t(x (m, t, p, s))
> swapPen3 x = \(m, t, p, s) -> p(x (m, t, p, s))
> stop3      = \(m, t, p, s) -> s

$ let program_B = turn3 (move3 (swapPen3 (move3 (swapPen3 stop3)))) :: Prog3
```

Attempt 2: researched method

```
> type Prog3' = Pos -> Dir -> PenDown -> Picture
> move3', turn3', swapPen3' :: Prog3' -> Prog3'
> stop3' :: Prog3'
> move3' f p d True = (p, forwards' d p):f (forwards' d p) d True
> move3' f p d False = f (forwards' d p) d False
> turn3' f p d b = f p (succ_wrap d) b
> swapPen3' f p d b = f p d (not b)
> stop3' p d b = []

$ let program_B' = turn3' (move3' (swapPen3' (move3' (swapPen3' stop3')))) :: Prog3'
```

You will find additional discussion of why I attempted questions 6 and 9 twice in section "3.2 Review of Section 2".

10.
===

```
$ let program_A = turn2 (move2 (pen2 (move2 (pen2 stop2)))) :: Prog2
$ let program_A' = turn2' (move2' (pen2' (move2' (pen2' stop2')))) :: Prog2'
```

We have a few commands to extract information from our programs (see "bash $ man ghc"):

```
$ :t program_A
program_A :: Prog2

$ :t program_A'
program_A' :: Prog2'
```

Prints the type of program_A or program_A'.

```
$ :info program_A
program_A :: Prog2  -- Defined at <interactive>:38:5

$ :info program_A'
program_A' :: Prog2'    -- Defined at <interactive>:39:5
```

if program_A or program_A' were a class, then the class methods and their types would be printed;
if program_A or program_A' were a type constructor, then its definition would be printed;
since program_A or program_A' is a function, its type is printed (as in command :t).

```
$ :info Prog2
type Prog2 = (A -> A, A -> A, A -> A, A) -> A       -- Defined at Questions_4_6_7_9_10.lhs:68:1
```

Since ‘Prog2’ is a type constructor, its definition is printed.

```
$ :info A
type A = Pos -> Dir -> Pos  -- Defined at Questions_4_6_7_9_10.lhs:66:1
```

Since ‘A’ is a type constructor, its definition is printed.

```
$ :info Prog2'
type Prog2' = Pos -> Dir -> Pos     -- Defined at Questions_4_6_7_9_10.lhs:92:1
```

Since ‘Prog2'’ is a type constructor, its definition is printed.

These GHCi commands are illustrated for the shallow embeddings of question 6, and can also be applied to question 9's program_B and program_B'.

3 What I have learned

Throughout this exercise (and throughout my life) I have been creating a log of what I have learned during programming. The last month was naturally dominated by Haskell, and I'm glad to have come across this succinct language. Due to its neat syntax 90% of the work can be done on paper with another 5 minutes to type it up!

Here is my recap of what I picked up from this assignment.

1 Review of Section 1

Question 1. The value of pattern matching.

This is a remarkable feature of Haskell. It enabled me to list all the expected function inputs in an itemized way.

One thing I noted down though — I was struggling to turn a data constructor & argument into a wild card (so as to provide a single wildcard to capture both *Move* and *Turn* constructors with argument x).
I kept coming across this: http://stackoverflow.com/questions/12520438/pattern-matching-on-constructor-wildcard, so instead I explored the alternative of guards (which I use in question 3). At this point I realized the purpose of deriving type classes like *Eq*.

Question 2. This was my first attempt at sourcing relevant functions from Prelude and elsewhere.

I learned to use function *succ* on data types, and recognized its limitations (were *succ* a UNIX userland program it would have had a flag to wrap, but in Haskell we have to find or write a completely new function — or perhaps overload the existing definition). So I needed to add function *succ_wrap* to my code snippets library.

Question 3. Easy enough.

I was enjoying pattern matching, but the alternative let me write less using guards.

Question 5. Now I'm sourcing functions, and using those that I have already defined.

Haskell seems to prefer this notion of defining very brief functions (2 lines), and then promoting the reuse of those functions elsewhere to make the caller functions cleaner and more readable.
Evidently function reuse is a fundamental concept of programming, but questions 1–5 hit this message home. In order to do that though, you first have to design it

all on paper. So when creating my programs, designing them first with an assignment style 'define this function which depends on that function etc.' specification is good, and one I have imitated in private projects since the start of this assignment.

Once you have defined the assignment functions on paper you have basically written the program!

Question 8. This answer offered 4 alternative definitions of plot.

The first step was just getting it working, and then 3 attempts at utilizing book knowledge to make the notation cleaner.

This question taught me how to build lists with recursion. Tuples (consisting of a variety of datatypes) collected into a list could probably capture anything, and we can traverse this structure easily ..

Question 11. OK. I picked up a lot from this one.

My first idea at this was totally non-recursive and lame. The plan was to create a header string, a body string and a footer string and glue them all together.
However, I wanted to do things your way. I was fortunate that I decided to web search for the "data XML" expression, and having found the library I had to figure it out.

What I picked up from this is that recursion can be used in more places than is initially apparent.

My initial approach was to make a hard coded string (header), make a recursive function to generate the body (line elements) which were clearly repeating, and then another single string for the footer.
It became clear that every element line itself is a recursive structure (a list of name / value pairs), and that the whole document (with its nested elements) is a recursive structure too! A whole new perspective of breaking problems down rather than settling on the first approach. I did not recognize that recursion in the first look, but now I'm going to be on the lookout for evidence of recursive structures elsewhere. A really good example.

2 Review of Section 2

I found the questions in section 2 harder because I wasn't sure where to look for information (I searched through a number of books via Amazon.com for the term 'shallow'). Fortunately Google lead me to your two papers which gave me a good understanding of shallow vs. deep embedding. I have stuck in a number of quotes from both book and online sources here to try to illustrate the research I did.

Question 4. That was simple enough.

I appreciated the difference between deep embedding and shallow embedding from a couple of online sources. We know that when deep embedding is used an abstract syntax tree is constructed[1, 2], yet nonetheless if we avoid deep embedding to produce shallow embedding, the two are still related by folds (I know folds!). I only read the first 1.5 pages of the 2014 paper[2] though because it gave me what I needed — see picture `./deep_vs_shallow.png` (below).

```
type Expr1 = ...
lit  :: Integer          → Expr1
add  :: Expr1 → Expr1 → Expr1

data Expr2 :: ∗ where
  Lit  :: Integer        → Expr2
  Add  :: Expr2 → Expr2 → Expr2
lit n    = Lit n
add x y  = Add x y

> eval2 (Add (Add (Lit 3) (Lit 4)) (Lit 5))
```

```
type Expr3 = Integer
lit n    = n
add x y  = x+y
eval3 :: Expr3 → Integer
eval3 n  = n

> eval3 (add (add (lit 3) (lit 4)) (lit 5))
```

---

eval2 uses constructors Lit, Add (deep embedding)
eval3 avoids the constructors (shallow embedding)[1,3].

Question 6, 7, 9, 10. These questions were related.

My inclination here was to start rewriting my functions using shallow embedding (Attempt 2), but I played with Attempt 1 (because question 10 implied we might be able to derive answers 6 and 9 from its listing).

---

[1] "Deep embedding: Haskell operations only build an interim Haskell data structure that reflects the expression tree. E.g. the Haskell expression '*a+b*' is translated to the Haskell data structure *Add (Var "a") (Var "b")*. This structure allows transformations like optimizations before translating to the target language."
`http://www.haskell.org/haskellwiki/Embedded_domain_specific_language`

[2] "With a deep embedding, terms in the DSL are implemented simply to construct an abstract syntax tree (AST), which is subsequently transformed for optimization. With a shallow embedding, terms in the DSL are implemented directly by their semantics, bypassing the intermediate AST and its traversal."
(2014) J. Gibbons, N. Wu, "*Folding Domain-Specific Languages:Deep and Shallow Embeddings*".
`http://www.cs.ox.ac.uk/jeremy.gibbons/publications/embedding.pdf`

[3] Another contrast of a deep embedding and its shallow embedding alternative is found in chapter 8.3 of (2014) R. Bird, "*Thinking Functionally with Haskell*".

It became clear that *Dir2* was not quite the same as *Dir*, as it made functions *forwards'* and *turn* redundant. My biggest hurdle here was appreciating the decision to use *Dir* (despite the fact that I was trying to provide shallow embedding). I pre-empted question 7 - but since this was addressing the fact that *Dir2* was not to be used (and that we recognize *Dir* is a deep embedding[4] for which we will provide an alternative later), I continued with Attempt 1, laying me up for some discussion and another solution in question 7.

I felt that since question 10 encompassed both *Prog2* and *Prog3*, surely I could reverse engineer it to answer questions 6 and 9.

I had to figure out the meaning of the listing in question 10:

program	(m)ove	(t)urn	(p) swapPen	(s)top	program
ProgS a =	(*a* → *a*,	*a* → *a*,	*a* → *a*,	*a*)	→ *a*

Where the first 3 commands (*move, turn, swapPen :: ProgS a* → *ProgS a*) were functions to functions, and the final command (*stop :: progS a*) did not take a function as an argument.

When I made my second attempt at question 6, I followed my instinct a bit more, by recreating the functions based on the type definition I had concluded (as shown in the 2013 paper[5]). Trying to implement this with *Dir2* and attempting to write a *swapPen* function without a boolean seemed impossible, but then I realized that you had given us a hint by the function name. In this second attempt, I found myself writing 4 empty function definitions ahead of time, and then padding them all out line by line as I had seen you doing on the projector (because there was no way it would compile unless I had them all — unless I wrote them individually, as in file `./shallow.png`). These definitions are starting to look very like formal notation.

[4] "In a shallow embedding logical formulas are written directly in the logic of the theorem prover, whereas in a deep embedding logical formulas are represented as a datatype."
`http://cstheory.stackexchange.com/questions/1370/shallow-versus-deep-embeddings`

[5] "Whereas in a deep embedding the constructors do nothing and the observers do all the work, in a shallow embedding it is the other way round: the observers are trivial, and all the computation is in the constructors.
Deep embedding makes it easier to extend the DSL with new observers, such as new analyses of programs in the language: just define a new function by induction over the abstract syntax. But it is more difficult to extend the syntax of the language with new operators, because each extension entails revisiting the definitions of all existing observers. Conversely, shallow embedding makes new operators easier to add than new observers ... The challenge of getting the best of both worlds —– extensibility in both dimensions at once —– has been called the expression problem."
(2013) J. Gibbons, *"Functional Programming for Domain–Specific Languages"*.
`http://www.cs.ox.ac.uk/jeremy.gibbons/publications/fp4dsls.pdf`

type *IntegerSet* = ...

```
empty  :: IntegerSet
insert :: Integer → IntegerSet → IntegerSet
delete :: Integer → IntegerSet → IntegerSet
member :: Integer → IntegerSet → Bool
```

For example, one might evaluate the expression

```
member 3 (insert 1 (delete 3 (insert 2 (insert 3 empty))))
```

type *IntegerSet* = [*Integer*] -- unsorted, duplicates allowed

```
empty :: IntegerSet
empty = []

insert :: Integer → IntegerSet → IntegerSet
insert x xs   = x : xs

delete :: Integer → IntegerSet → IntegerSet
delete x xs   = filter (≠ x) xs

member :: Integer → IntegerSet → Bool
member x xs = any (≡ x) xs
```

Another example of shallow embedding.
type *IntegerSet and its 4 programs*[5]

In the example from question 10 I recognized how programs *move*, *turn*, *swapPen* and *stop* were captured in the type brackets — *progS* was taking a program that could consist of any of those 4 programs (where the final *stop* command did not lead to another function), and how the pattern matches (those that took another function: *m*, *t*, *p*) would bring that command to the head and execute the remainder program. It's still a bit confusing, but there's a definite symmetry that can be reverse engineered and altered. It reminded me of constructing a parser in `lex` (a common undergraduate topic). The robot has given us a definite illustration of a language of nested commands (from a limited dictionary) to create output (**data** *Prog* and *XML* have shown one language can be transformed to another for designing markup). Once you've created a DSL (the deeply embedded language for the robot), you can automatically translate its structure to another (e.g. *XML*). That's neat.

You speak about this approach in *practical exercise 12* - in future I'm going to consider designing a domain specific language for depicting a chain of functions, because with this assignment as a reference it's really not as complicated as it sounds.

3 End

Thanks for introducing me to Haskell!

Honestly, of the many languages I have come across, this is the best.

I'm going to use it for prototyping my final project (which will be in objective-c), probably deconstructing the Haskell into some formal notation (or vice versa). I've started practising it for quick scripting already (see `http://users.ox.ac.uk/~kell3138/code/HAS/wordlist.hs`). Since I could now write files, I had a go at reading them and interacting with the user.