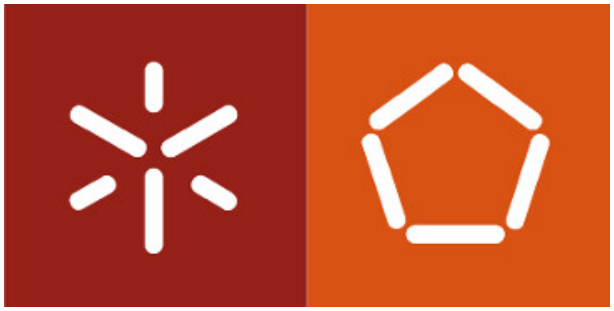
Software Architecture and Calculi - Assignment 2

Modelling and analysis of cyber-physical systems now with monads

a85700 Pedro Costa
a85954 Luís Ribeiro



Mestrado em Engenharia Informática
Universidade do Minho

Contents

1 First Part **2**

1.1 Exercise 1 . . . 2

1.2 Exercise 2 . . . 4

1.2.1 addtoEnd . . . 4

1.2.2 hCycles . . . 4

2 Second Part **5**

2.1 Exercise 3 . . . 5

2.2 Exercise 4 . . . 7

2.2.1 tadjacentNodes . . . 7

2.2.2 taddToEnd . . . 7

2.2.3 hCyclesCost . . . 7

3 Third Part **8**

3.1 Exercise 5 . . . 8

4 Fourth Part **9**

4.1 Exercise 6 . . . 9

4.1.1 IO monad - Generic graphs . . . 9

4.1.2 State monad - Eletric bus and the free, complete, rides! . . . 9

4.1.3 The necessity of a list monad transformer . . . 9

4.1.4 Exception monad - Traffic info and other potential mishap's . . . 9

4.1.5 Writer Monad - Knowing what path was being followed until a failure . . . 9

4.1.6 Summing up . . . 10

5 Conclusion **11**

1 | First Part

1.1 Exercise 1

In order to compute the output of this λ-term for input 0, first we need to interpret this term as a function by assigning functional semantics to the latter. Note that, this term represents a sequential composition:

$$\frac{[\![x : \mathbb{N} \vdash_c \mathit{choice}(\mathit{return}(x+1), \mathit{return}(x-1)) : \mathbb{N}]\!] = f \qquad [\![x : \mathbb{N}, y : \mathbb{N} \vdash_c \mathit{choice}(\mathit{return}(y-1), \mathit{return}(y+1)) : \mathbb{N}]\!] = g}{[\![x : \mathbb{N} \vdash_c y \leftarrow \mathit{choice}(\mathit{return}(x-1), \mathit{return}(x+1)); \mathit{choice}(\mathit{return}(y-1), \mathit{return}(y+1)) : \mathbb{N}]\!] = g^* \cdot \mathit{str} \cdot \langle \mathit{id}, f \rangle}$$

This composition was calculated using tensorial strength str, because of the sharing contexts between terms, later denoted as functions f and g, that compose.
Now, we will proceed to extend both functions f and g.

— Extending function f

$$(+,2)\ \frac{\dfrac{[\![x : \mathbb{N} \vdash x-1 : \mathbb{N}]\!] = \mathit{dec}}{[\![x : \mathbb{N} \vdash_c \mathit{return}(x-1) : \mathbb{N}]\!] = \eta \cdot \mathit{dec}} \qquad \dfrac{[\![x : \mathbb{N} \vdash x+1 : \mathbb{N}]\!] = \mathit{inc}}{[\![x : \mathbb{N} \vdash_c \mathit{return}(x+1) : \mathbb{N}]\!] = \eta \cdot \mathit{inc}}}{[\![x : \mathbb{N} \vdash_c \mathit{choice}(\mathit{return}(x-1), \mathit{return}(x+1)) : \mathbb{N}]\!] = [\![\mathit{choice}]\!] \cdot \langle \eta \cdot \mathit{dec}, \eta \cdot \mathit{inc} \rangle}$$

— Extending function g

$$(+,2)\ \frac{\dfrac{[\![y : \mathbb{N} \vdash y-1 : \mathbb{N}]\!] = \mathit{dec} \quad [\![x : \mathbb{N}, y : \mathbb{N} \vdash y : \mathbb{N}]\!] = \pi_2}{[\![x : \mathbb{N}, y : \mathbb{N} \vdash_c \mathit{return}(y-1) : \mathbb{N}]\!] = \eta \cdot \mathit{dec} \cdot \pi_2} \qquad \dfrac{[\![y : \mathbb{N} \vdash y+1 : \mathbb{N}]\!] = \mathit{inc} \quad [\![x : \mathbb{N}, y : \mathbb{N} \vdash y : \mathbb{N}]\!] = \pi_2}{[\![x : \mathbb{N}, y : \mathbb{N} \vdash_c \mathit{return}(y+1) : \mathbb{N}]\!] = \eta \cdot \mathit{inc} \cdot \pi_2}}{[\![x : \mathbb{N}, y : \mathbb{N} \vdash_c \mathit{choice}(\mathit{return}(y-1), \mathit{return}(y+1)) : \mathbb{N}]\!] = [\![\mathit{choice}]\!] \cdot \langle \eta \cdot \mathit{dec} \cdot \pi_2, \eta \cdot \mathit{inc} \cdot \pi_2 \rangle}$$

By that we have,

$$f = [\![\mathit{choice}]\!] \cdot \langle \eta \cdot \mathit{dec}, \eta \cdot \mathit{inc} \rangle$$
$$g = [\![\mathit{choice}]\!] \cdot \langle \eta \cdot \mathit{dec} \cdot \pi_2, \eta \cdot \mathit{inc} \cdot \pi_2 \rangle$$
$$\mathit{dec}\ n = n - 1 \qquad\qquad \mathit{inc}\ n = n + 1$$

Since *choice* is the corresponding signature to the **PowerSet** Monad, we must compute its triple $(T, \eta, (-)^*)$ and its tensorial strength (str).

$$T : X \mapsto \{A \mid A \subseteq X\}$$
$$\eta : x \mapsto \{x\}$$

$$f^* : PX \to PY$$
$$f^*(A) = \bigsqcup_{x \in A} f(x)$$

$$str : X \times PX \to P(X \times Y)$$
$$(x, A) \mapsto \{(x, a) \mid a \in A\}$$

Now that the definition of every needed function have been determined, we can now calculate the output of this λ-term for input 0.

$$(g^* \cdot str \cdot \langle id, f \rangle)\ 0$$
$$\equiv \{ \text{ expand } f \text{ and } g \ \}$$
$$(\llbracket choice \rrbracket \cdot \langle \eta \cdot dec \cdot \pi_2, \eta \cdot inc \cdot \pi_2 \rangle)^* \cdot str \cdot \langle id, \llbracket choice \rrbracket \cdot \langle \eta \cdot dec, \eta \cdot inc \rangle \rangle\ 0$$
$$\equiv \{ \text{ split definition } \}$$
$$(\llbracket choice \rrbracket \cdot \langle \eta \cdot dec \cdot \pi_2, \eta \cdot inc \cdot \pi_2 \rangle)^* \cdot str \cdot (0, \llbracket choice \rrbracket \cdot \langle \eta \cdot dec, \eta \cdot inc \rangle\ 0)$$
$$\equiv \{ \text{ split definition; } \eta \text{ definition } \}$$
$$(\llbracket choice \rrbracket \cdot \langle \eta \cdot dec \cdot \pi_2, \eta \cdot inc \cdot \pi_2 \rangle)^* \cdot str \cdot (0, \llbracket choice \rrbracket\ (\{-1\}, \{1\}))$$
$$\equiv \{ \text{ choice definition: } choice\ (a, b) = a \cup b \}$$
$$(\llbracket choice \rrbracket \cdot \langle \eta \cdot dec \cdot \pi_2, \eta \cdot inc \cdot \pi_2 \rangle)^* \cdot str\ (0, \{-1, 1\})$$
$$\equiv \{ \ str \text{ definition } \}$$
$$(\llbracket choice \rrbracket \cdot \langle \eta \cdot dec \cdot \pi_2, \eta \cdot inc \cdot \pi_2 \rangle)^*\ \{(0, -1), (0, 1)\}$$
$$\equiv \{ \ f^* \text{ definition} \to \bigsqcup_{for\ each\ pair} \}$$
$$\llbracket choice \rrbracket \cdot \langle \eta \cdot dec \cdot \pi_2, \eta \cdot inc \cdot \pi_2 \rangle\ (0, -1) \cup \llbracket choice \rrbracket \cdot \langle \eta \cdot dec \cdot \pi_2, \eta \cdot inc \cdot \pi_2 \rangle\ (0, 1)$$
$$\equiv \{ \text{ split definition; projection definition; } \eta \text{ definition } \}$$
$$\llbracket choice \rrbracket\ (\{-2\}, \{0\}) \cup \llbracket choice \rrbracket\ (\{0\}, \{1\})$$
$$\equiv \{ \text{ choice definition: } choice\ (a, b) = a \cup b; \cup\text{-distribution } \}$$
$$\{-2\} \cup \{0\} \cup \{0\} \cup \{2\}$$
$$\equiv \{ \text{ union } \}$$
$$\{-2, 0, 2\}$$

1.2 Exercise 2

The described problem is based on important concepts of graph theory: **Hamiltonian path** and **Hamiltonian cycle**.

— A path in this graph is called Hamiltonian if it visits each node of the graph exactly once.

— A Hamiltonian cycle is a path $X_1 \rightarrow X_n$ such that $X_1 = X_n$ and $X_2 \rightarrow X_n$ is a Hamiltonian path.

Hamiltonian paths and Hamiltonian cycles can be computed via backtracking using the **List Monad**.
Our implementation of the functions `addtoEnd` and `hCycles` is based on the concepts described above.

1.2.1 addtoEnd

```
addtoEnd :: Path -> [Node] -> [Path]
addtoEnd p ns = ns >>= (\n -> if adj(n, last p) && not (elem n p)
                              then return $ p ++ [n]
                              else [])
```

The idea behind this function is to compute each node from the list of nodes (`[Node]`), and check if they are not defined in the path given as input and if they are adjacent to the last node of the path (`last p`). If they respect both conditions, the node will be appended to the end of the path, and this "new" path is returned. If one of the conditions fail, an empty list is returned.

1.2.2 hCycles

```
hCycles :: Node -> [Path]
hCycles n = addtoEnd [n] allNodes >>= hCyclesAux where
    hCyclesAux p | ended p = bool [] (return $ p ++ [head p]) $ adj (head p, last p)
                 | otherwise = addtoEnd p allNodes >>= hCyclesAux
    ended p = sort p == sort allNodes
```

This implemented function takes into account the concepts depicted above, where its described the ideas behind Hamiltonian path and Hamiltonian cycle.
As we can see, before computing the `addtoEnd` function, we start by creating a path with the starting node in it. By running the `addtoEnd` function, we are creating possible paths, starting from the starting node. Then, we compute each returned path and check if they represent a Hamiltonian path. If they do, then the path is returned, otherwise the path will be computed again following the same method.

2 | Second Part

2.1 Exercise 3

$$[\![x : \mathbb{A} \vdash_c (\lambda y : \mathbb{A} \to \mathbb{A}.\ wait_2(y\ x))(\lambda z : \mathbb{A}.\ wait_1(return\ z)) : \mathbb{A}]\!] = [\![x : \mathbb{A} \vdash_c wait_3(return\ x) : \mathbb{A}]\!]$$

This equality can be either proved using **point-free** notation or **point-wise** notation.
The idea behind point-wise notation, is to prove that both λ-terms, that will be represented as functions, are equal for every input given x. Whereas, the point-free notation requires extra steps, in order to remove the λ notation from each term.
Since we used point-free notation, before proving the equality, we first need to remove every *lambda* notation entirely, by simplifying the term's semantics and retrieving each as functions.

— First we will simplify the left-hand term of the equality.

$$[\![x : \mathbb{A} \vdash_c (\lambda y : \mathbb{A} \to \mathbb{A}.\ wait_2(y\ x))(\lambda z : \mathbb{A}.\ wait_1(return\ z)) : \mathbb{A}]\!]$$

$$\equiv \{\ \text{app definition}\ \}$$

$$app \cdot \langle([\![x : \mathbb{A} \vdash (\lambda y : \mathbb{A} \to \mathbb{A}.\ wait_2(y\ x)) : \mathbb{A} \to \mathbb{A} \to \mathbb{A}]\!]), ([\![x : \mathbb{A} \vdash (\lambda z : \mathbb{A}.\ wait_1(return\ z)) : \mathbb{A} \to \mathbb{A}]\!])\rangle$$

$$\equiv \{\ \text{simplifying each } \lambda\text{-term as } \langle \lambda f, \lambda g\rangle\ \}$$

$$app \cdot \langle \lambda f, \lambda g\rangle$$

$$\equiv \{\ \text{absorption-x}\ \}$$

$$app \cdot (\lambda f \times id) \cdot \langle id, \lambda g\rangle$$

$$\equiv \{\ \text{cancellation-exp}\ \}$$

$$f \cdot \langle id, \lambda g\rangle$$

Now, we will proceed to extend both functions f and g.

— Extending function f

$$\dfrac{\dfrac{(wait_2, 1)\quad \dfrac{[\![x : \mathbb{A}, y : \mathbb{A} \to \mathbb{A} \vdash y : \mathbb{A} \to \mathbb{A}]\!] = \pi_2 \quad [\![x : \mathbb{A}, y : \mathbb{A} \to \mathbb{A} \vdash x : \mathbb{A}]\!] = \pi_1}{[\![x : \mathbb{A}, y : \mathbb{A} \to \mathbb{A} \vdash_c y\ x : \mathbb{A}]\!] = app \cdot \langle \pi_2, \pi_1\rangle}}{[\![x : \mathbb{A}, y : \mathbb{A} \to \mathbb{A} \vdash_c wait_2(y\ x) : \mathbb{A}]\!] = [\![wait_2]\!] \cdot app \cdot \langle \pi_2, \pi_1\rangle}}{[\![x : \mathbb{A} \vdash (\lambda y : \mathbb{A} \to \mathbb{A}.\ wait_2(y\ x)) : \mathbb{A} \to \mathbb{A} \to \mathbb{A}]\!] = \lambda([\![wait_2]\!] \cdot app \cdot \langle \pi_2, \pi_1\rangle)}$$

— Extending function g

$$\dfrac{\dfrac{(wait_1, 1)\quad \dfrac{[\![x : \mathbb{A}, z : \mathbb{A} \vdash z : \mathbb{A}]\!] = \pi_2}{[\![x : \mathbb{A}, z : \mathbb{A} \vdash_c return\ z : \mathbb{A}]\!] = \eta \cdot \pi_2}}{[\![x : \mathbb{A}, z : \mathbb{A} \vdash_c wait_1(return\ z) : \mathbb{A}]\!] = [\![wait_1]\!] \cdot \eta \cdot \pi_2}}{[\![x : \mathbb{A} \vdash (\lambda z : \mathbb{A}.\ wait_1(return\ z)) : \mathbb{A} \to \mathbb{A}]\!] = \lambda([\![wait_1]\!] \cdot \eta \cdot \pi_2)}$$

By that we have,

$$f \cdot \langle id, \lambda g \rangle$$
$$\equiv \{ \text{ extending } f \ \}$$
$$[\![wait_2]\!] \cdot app \cdot \langle \pi_2, \pi_1 \rangle \cdot \langle id, \lambda g \rangle$$
$$\equiv \{ \text{ fusion-x; projection definition } \}$$
$$[\![wait_2]\!] \cdot app \cdot \langle \lambda g, id \rangle$$
$$\equiv \{ \text{ absorption-x } \}$$
$$[\![wait_2]\!] \cdot app \cdot (\lambda g \times id) \cdot \langle id, id \rangle$$
$$\equiv \{ \text{ cancellation-exp } \}$$
$$[\![wait_2]\!] \cdot g \cdot \langle id, id \rangle$$
$$\equiv \{ \text{ extending } g \ \}$$
$$[\![wait_2]\!] \cdot [\![wait_1]\!] \cdot \eta \cdot \pi_2 \cdot \langle id, id \rangle$$
$$\equiv \{ \text{ cancellation-x; natural-id: } f \cdot id = f \ \}$$
$$[\![wait_2]\!] \cdot [\![wait_1]\!] \cdot \eta$$
$$\equiv \{ \ wait \text{ composition: } wait_n(wait_m(M)) = wait_{n+m}(M) \ \}$$
$$[\![wait_3]\!] \cdot \eta$$

— Simplifying the right-hand term of the equality.

$$\frac{(wait_3, 1) \quad \dfrac{[\![x : \mathbb{A} \vdash x : \mathbb{A}]\!] = id}{[\![x : \mathbb{A} \vdash_c return\ x : \mathbb{A}]\!] = \eta \cdot id}}{[\![x : \mathbb{A} \vdash_c wait_3(return\ x) : \mathbb{A}]\!] = [\![wait_3]\!] \cdot \eta \cdot id}$$

— Since both terms have been introduced as functions, we can easily prove the equality.

$$[\![x : \mathbb{A} \vdash_c (\lambda y : \mathbb{A} \to \mathbb{A}.\ wait_2(y\ x))(\lambda z : \mathbb{A}.\ wait_1(return\ z)) : \mathbb{A}]\!] = [\![x : \mathbb{A} \vdash_c wait_3(return\ x) : \mathbb{A}]\!]$$
$$\equiv \{ \text{ definitions } \}$$
$$[\![wait_3]\!] \cdot \eta = [\![wait_3]\!] \cdot \eta \cdot id$$
$$\equiv \{ \text{ natural-id: } f \cdot id = f \ \}$$
$$[\![wait_3]\!] \cdot \eta = [\![wait_3]\!] \cdot \eta$$
$$\equiv \{ \text{ trivial } \}$$
$$True$$

2.2 Exercise 4

The second part of this assignment aims to address precisely this problem, by labelling the edges with natural numbers, in order to represent costs within edges. The cost of a cycle is calculated by summing all the labels encountered when traversing the edges stipulated by the cycle.
In the previous task, we used the List monad to compute those Hamiltonian cycles. The cost of paths, on the other hand, can be computed via the **Duration Monad**. Both monads working together thus allow to compute a solution for TSP.

2.2.1 tadjacentNodes

```
tadjacentNodes :: Node -> [Node] -> [Duration Node]
tadjacentNodes n ns = ns >>= \x -> case adjT(n,x) of
                                   Just y -> return $ Duration (y, x)
                                   Nothing -> []
```

In this exercise, it was also requested to do a function that builds the adjacent nodes to some other, putting each of the adjacent in the **Duration context**. To do that, we recurred to the predefined adjT, discarding every result that gave Nothing.

2.2.2 taddToEnd

```
taddToEnd :: Duration Path -> [Duration Node] -> [Duration Path]
taddToEnd p@(Duration (cp, ps)) ns = ns >>=
                                     (\d@(Duration (c,n)) -> if checkAdj n && alreadyInPath n ps
                                     then return $ addCost p d
                                     else [])
where
checkAdj n = adj (n, last ps)
alreadyInPath n = not . elem n
addCost p (Duration (c,n)) = p >>= (\x -> Duration (c,x ++ [n]))
```

In comparison with the previous **addtoEnd**, the only difference is that now we also accumulate the cost with our current. For this purpose we added the **addCost** auxiliar function.

2.2.3 hCyclesCost

```
hCyclesCost :: Node -> [Duration Path]
hCyclesCost n = branch (return [n]) >>= hCyclesAux
where
hCyclesAux d@(Duration (c,p)) | ended p = bool [] (return $ d >>= addLast) $ adj (head p, last p)
                              | otherwise = branch d >>= hCyclesAux
ended p = sort p == sort allNodes
addLast p = Duration (fromJust $ adjT (last p, head p), p ++ [head p])
branch d@(Duration (_,p)) = taddToEnd d (tadjacentNodes (last p) allNodes)
```

In comparison with the previous **hCycles**, the key difference is, again, dealing with the accumulating costs. During the base recursion this is covered by **taddToEnd** so we only had to explicitly cover the finishing case.

3 | Third Part

3.1 Exercise 5

Monads allows us to improve our programming practices in various ways. It definitely comes associated with some drawbacks but, hopefully, the idea is still strong enough to prove itself useful in several scenarios.

A interesting example of a monad being very useful passes through converting partial functions into total ones which allows us to express the absence of value really well. This concept exists in none functional programming languages too, like in C with pointers, for example. Just as the Maybe monad captures this functionality, other monads cover other interesting useful features. We can use the Writer monad to log information, the Powerset monad to bruteforce problems, the IO monad to interact with real-time input/output operations and many others. Therefore, they are a way to approach and easily solve specific problems (Adding functionality).

Now, the cherry on top using monads to add functionality also passes through them being able to solve, otherwise rather difficult, tasks more naturally. The previous exercises are good examples of how the Powerset monad recurs to backtracking to obtain every possible solution to a problem. It managed to turn an algorithm that usually needs to keep a list of visited nodes into a much more trivial problem where we easily guarantee to touch every node. (Compact way to develop complex solutions).

Another big advantage that we only saw a glimpse off during the previous exercises passes by the fact of monads having a very generic meaning. It's basically guaranteed that one can just replace the outer monad of a certain function and arrive to something with a valuable meaning in a different concept.

However, as we mentioned right on opening, they do have drawbacks. The first and, probably, the one that justifies the most the lack of effort to actually interact with this interfaces passes through the steep learning curve. Coming from category theory, monads definitely require some background understanding to actually see how the concept manages to apply so well to different problems. (Steep learning curve)

Another, more important, drawback comes from the fact that monads don't scale particularly well. It's all fine when we are using maybe 2 or 3 monads on a stack because the boilerplate code is still tolerable. With very big stacks, despite having access to various composed functionalities that can cover entirely anything that a piece of software might need (state, reading data, logging data, etc), comes a lot of boilerplate code due to constantly needing to relocate the context to the operation that one needs to do. We do lack some experience when it comes to working with real-world examples of monad stacks so we might actually be missing our point here, and perhaps it's actually not as bad as we've noticed so far.

Either way, monads are definitely worth learning especially due to the way that they influence our way of thinking about a problem. Modelling before solving is the way to actually develop software and monads are one of the best proofs of that fact.

4 | Fourth Part

4.1 Exercise 6

As requested in this exercise, we tried our best to create significant functionalities whilst recurring to monadic machinery. In order to try to explain our motivation for each chosen monad, in the following subsections we describe every functionality that we added and how it was naturally (and easily) represented with a certain monad.

4.1.1 IO monad - Generic graphs

Our first intention was to generalize the base graph, allowing the user to adjust the graph to his own needs. As mentioned by the teacher, this was done by using the IO monad. At first, this required us to have a lot more arguments in our functions (a list of adjacencies and nodes) but later they were simply included in our state.

4.1.2 State monad - Eletric bus and the free, complete, rides!

Despite the teacher mentioning that we should try to avoid using functionalities that other groups might use, we've always been particularly curious about the State monad. It looks a lot harder than what it actually is. It was used, as suggested, to create a notion of battery and understand if there were paths that could be done by an electric bus without ever losing it's battery entirely.

4.1.3 The necessity of a list monad transformer

While we were implementing the above functionality we incurred in a problem. The State was covering every path which didn't allowed us to represent the battery "individually". To solve that problem, we started using a list monad transformer that gave us the ability to produce a state for each path.

4.1.4 Exception monad - Traffic info and other potential mishap's

Another interesting functionality that we felt would fit well in the problem was the possibility of, by some specific reason, not being able to complete the path. This could occur because there was a sinkhole on a street or maybe the bus had a flat tire! To implement this, we used the Exception monad.
We created 2 types of errors, one representing local failures that might happen to a specific bus and another type to represent failures that are known by external sources, for example, a road block on an edge.

4.1.5 Writer Monad - Knowing what path was being followed until a failure

The previous functionality, despite being relevant, brought some redundancy to our final output. It could happen that multiple paths failed for the same reason, which would be displayed simply as the same error 3 times. This motivated us to use the Writer monad to be aware of what path was being followed until the failure happened!

4.1.6 Summing up

We know we might have gone a bit overboard by using so many functionalities, but our main motivation to doing this was to "test" our written essay about why monads matter. Every time we added a new monad to our stack, the changes were very, very minimal. We also managed to isolate each monad functionality in a very visible way and managed to have somewhat generic functions all around our program. None of us had a lot of experience with monad transformers until this point, but this project definitely showed us that they are very interesting and powerful.

5 | Conclusion

In conclusion, this was a great project to finish our last "typical" semester. We have always been told that formal methods associates with a lot of advantages but, in many occasions, we don't get to see their use in complex, real world scenarios.

With this project, however, we got a taste of what seems to be real world Haskell and worked with very advanced concepts that are core to develop software recurring to functional programming.

On pair with the above, we also grasped this concepts in a more theoretical manner due to the lambda calculus exercises which gave us a different perspective of these matters.

Our main difficulties differed between the Haskell and the lambda calculus parts. Regarding the Haskell exercises, we only encountered difficulties in the last exercise, especially when it came to understanding exactly what real world concept could be represented by a certain monad. Code-wise it went fairly well since we already had some background in the language.

Regarding lambda calculus, our main difficulties were finding which was the right applying rule, so that the logical process of deducting was preserved and consequently each λ-term had the proper interpreted function.

It was an interesting project that shed a light on many important concepts and we think that other previous projects should try to force us to deal with more real examples and concepts.