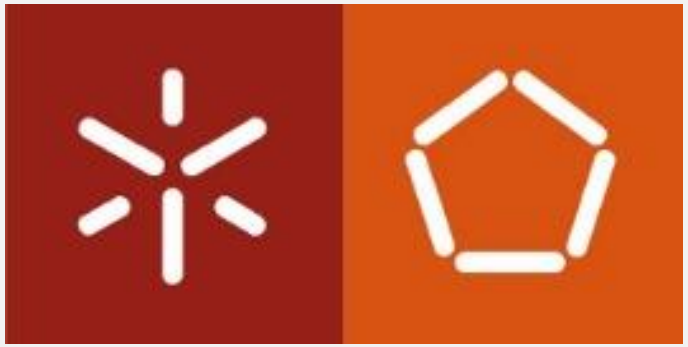
Software Architecture and Calculi - Assignment 2

Modelling and analysis of cyber-physical systems now with monads

A85700 Pedro Costa

A85954 Luís Ribeiro



First Part

❑ Exercise 1

- ❑ Using tensorial strength (str) because of the sharing contexts in the defined composition
- ❑ Powerset Monad and its corresponding signature: choice

❑ Exercise 2

- ❑ Concepts of Hamiltonian path and Hamiltonian cycle
- ❑ addtoEnd: Adding nodes to the path under some conditions

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

- ❑ hCycles: Takes into account the ideas behind Hamiltonian path and Hamiltonian cycle

```
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
```

Second Part

- Exercise 3
  - Assigning semantics to each *lambda* term and identifying the corresponding interpreted function (Deduction rules)
  - Simplifying the retrieved functions by applying calculi rules (Exponential and Product rules mainly)
  - Proving the equality between interpreted functions
- Exercise 4
  - Labelling edges with costs: Introduction to the Duration Monad
  - tadjacentNodes: Builds the adjacent nodes to some other, putting each of the adjacent in the Duration context

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

Second Part

- taddToEnd: Same thought process as the last one, but now we accumulate the cost

```
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]))
```

- hCyclesCost: Same thought process as the last one, but now we need to deal with the accumulating costs

```
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)
```

Why monads matter?

✓ **Advantages**

- ❑ Adding functionality (Maybe Monad as a pointer binder, Writer has a logger)
- ❑ Compact way to develop complex solutions (removes the need to keep a visited node list in the TSP)
- ❑ Modular

▪ **Disadvantages**

- ❑ Steep learning curve
- ❑ Monad transformers have a poor scaling. However, monad stacking does not (we were mistaken and changed our minds, only realizing this when finishing this slides)

Fourth Part

- ❑ IO Monad – Generic graphs
- ❑ State Monad – Electric bus and finding battery sufficient paths
- ❑ The necessity of a list transformer
- ❑ Exception Monad – Traffic into and other potential pathing disablers
- ❑ Writer Monad – Logging of the intermediate path
- ❑ Our thoughts and what we learned

Conclusion

- **Functional Programming**
  - Real world Haskell and develop software recurring to functional programming
  - The *lambda* calculus perspective
- **Difficulties**
  - Last exercise: The understanding before coding
  - *Lambda* calculus: Applying the right rule in order to respect the logical process of deducting
- **Final thoughts**