

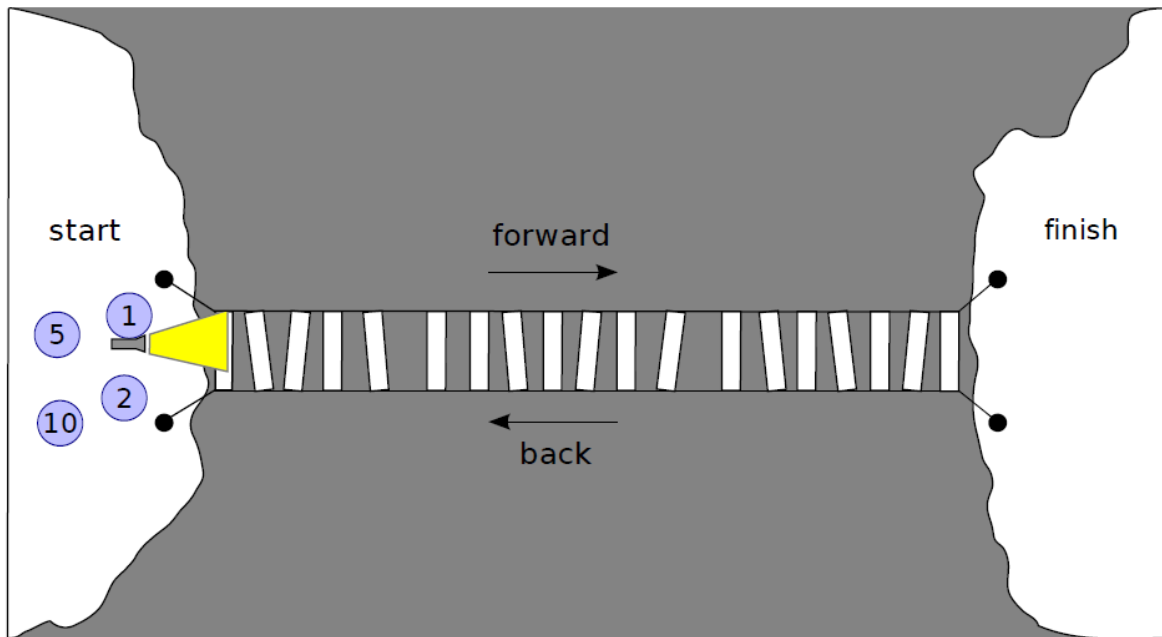
The Bridge and Lantern Riddle

Practical Assignment 2
Cyber-Physical Computation
Docente: Renato Neves

Márcio Mano
(PG47446)
MEFis

M^a João Portela
(PG47478)
MEFis

Pedro Teixeira
(A75049)
MEFis



Conteúdo

1	Enunciado	2
2	First Task: Your first task is to verify these claims using Haskell.	2
2.1	Model the system above using what you learned about monads, in particular the duration and non-deterministic ones.	2
2.1.1	Função <i>allValidPlays</i>	2
2.1.2	Optional Task 1	3
2.1.3	Função <i>exec</i>	4
2.2	Show that it is indeed possible for all adventurers to be on the other side in 17 minutes	5
2.2.1	Função <i>lep17</i>	5
2.3	Show that it is impossible for all adventurers to be on the other side in less than 17 minutes	6
2.3.1	Função <i>l17</i>	6
2.4	Optional Task 2	7
2.5	Implementation of the monad used for the problem of the adventurers	7
2.6	Decisões de implementação	8
3	Second Part	8
3.1	Your second task is to compare both approaches (via <i>UPPAAL</i> and <i>Haskell</i>) to the adventurer's problem. Specifically, you should provide strong and weak points of the two approaches: what are the (dis)advantages of UPPAAL for this problem? And what about Haskell?	8
4	Optional Task 3 - Adventurers.hs	9
5	Optional Task 4 - Adventurers.hs	11
6	Optional Task 5 - Adventurers2.hs	12
7	Optional Task 6 - Adventurers.hs	14
8	Conclusion	15
9	Attachments	17
10	Referências	29

1 Enunciado

In the middle of the night, four adventurers encounter a shabby rope-bridge spanning a deep ravine. For safety reasons, they decide that no more than 2 people should cross the bridge at the same time and that a flashlight needs to be carried by one of them in every crossing. They have only one flashlight. The 4 adventurers are not equally skilled: crossing the bridge takes them 1, 2, 5, and 10 minutes, respectively. A pair of adventurers crosses the bridge in an amount of time equal to that of the slowest of the two adventurers.

One of the adventurers claims that they cannot be all on the other side in less than 19 minutes. One companion disagrees and claims that it can be done in 17 minutes.

2 First Task: Your first task is to verify these claims using Haskell.

2.1 Model the system above using what you learned about monads, in particular the duration and non-deterministic ones.

O modelo do sistema e correspondentes *monads* encontram-se nos ficheiros *Adventurers.hs* e *DurationMonad.hs*.

De forma a modelar o sistema descrito recorreu-se a funções chave que permitem gerar, filtrar e invocar recursivamente os estados do sistema, nomeadamente dos aventureiros e lanterna. Estas funções, chamadas ***allValidPlays***, ***exec***, ***leq17*** e ***l17***, serão discutidas em mais pormenor a seguir.

2.1.1 Função *allValidPlays*

Com esta função pretende-se derivar todos os estados possíveis de obter ao final de uma iteração aplicada a um estado passado como argumento. Para isso começa-se por encontrar quais as jogadas validas , isso é feito tendo em conta o lado em que a lanterna se encontra, e filtrar os aventureiros que estiverem no mesmo. Com base nisso cria-se uma lista com todas as combinações de 2,1 ou nenhum aventureiro e adiciona-se a lanterna. Uma vez criada essa lista de objetos podemos aplicar a função dada ***mChangeState*** e obter um novo estado.

Este raciocínio é repetido para cada elemento da lista de objetos, criando também uma lista de *Duration State* possíveis.

```

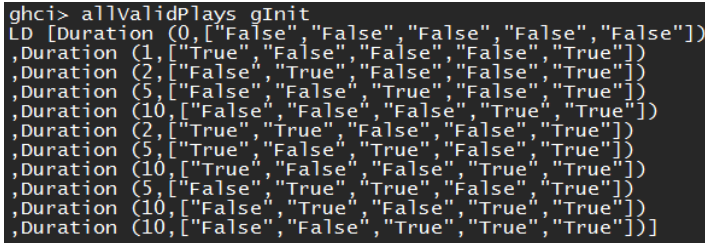
1  -- Initial state of the Game
2  gInit :: State
3  gInit = const False
4
5  -- Desired final state of the Game
6  gEnd :: State
7  gEnd = const True
8
9  -- Changes the 'State' of the game for a given 'Object'
10 changeState :: Objectc -> State -> State
11 changeState a s = let v = s a
12                  in (\x -> if x == a then not v else s x)
13
14 -- Changes the 'State' of the Game of a list of 'Objects'
15 mChangeState :: [Objectc] -> State -> State
16 mChangeState os s = foldr changeState s os
17
18 -- List of all moves
19 possibleMoves :: [[Objectc]]
20 possibleMoves = [
21   [], -- No 'Object' changes state / position
22   [(Left P1)], -- 'Adventure P1' changes state / position
23   [(Left P2)], -- 'Adventure P2' changes state / position
24   [(Left P5)], -- 'Adventure P5' changes state / position
25   [(Left P10)], -- 'Adventure P10' changes state / position
26   [(Left P2), (Left P1)], -- 'Adventure P2 & P1' changes state / position
27   [(Left P5), (Left P1)], -- 'Adventure P5 & P1' changes state / position
28   [(Left P10), (Left P1)], -- 'Adventure P10 & P1' changes state / position
29   [(Left P5), (Left P2)], -- 'Adventure P5 & P2' changes state / position

```

```

30 [(Left P10), (Left P2)], -- 'Adventure P10 & P2' changes state / position
31 [(Left P10), (Left P5)] -- 'Adventure P10 & P5' changes state / position
32
33 -- Given a 'State' calculates the list of possible moves with respect to the 'Latern' position
34 moves :: State -> [[Objetc]]
35 moves s = filter (\list -> and (map (\a -> (s a) == (s (Right ()))) list)) possibleMoves
36
37 -- Changes the 'State' of the 'Objects' inside a list and gives a 'Duration State' with the
38 -- time needed to perform the move and the resulting 'State'
39 fun :: State -> [Objetc] -> Duration State
40 fun s [] = Duration (0, s)
41 fun s (h:t) = Duration (getTimeAdv((\ (Left a) -> a) h), mChangeState ((h:t)++[(Right ())] s)
42
43 {-- For a given state of the game, the function presents all the
44 possible moves that the adventurers can make. --}
45 allValidPlays :: State -> ListDur State
46 allValidPlays s = LD (map (fun s) (moves s))

```

Listing 1: Função *allValidPlays* e as suas funções auxiliares.


```

ghci> allValidPlays gInit
LD [Duration (0,["False","False","False","False","False"]),
,Duration (1,["True","False","False","False","True"]),
,Duration (2,["False","True","False","False","True"]),
,Duration (5,["False","False","True","False","True"]),
,Duration (10,["False","False","False","True","True"]),
,Duration (2,["True","True","False","False","True"]),
,Duration (5,["True","False","True","False","True"]),
,Duration (10,["True","False","False","True","True"]),
,Duration (5,["False","True","True","False","True"]),
,Duration (10,["False","True","False","True","True"]),
,Duration (10,["False","False","True","True","True"])]

```

Figura 1: Execução da função *allValidPlays* e o seu output.

2.1.2 Optional Task 1

De modo a otimizar o nosso modelo introduzimos a possibilidade de numa jogada não realizarmos nenhuma alteração de estado nos aventureiros ou na lanterna. Tal "movimento" é sinalizada por [], como podemos observar na lista de *possibleMoves*.

```

1 ...
2 -- List of all moves
3 possibleMoves :: [[Objetc]]
4 possibleMoves = [
5   [], -- No 'Object' changes state / position
6   [(Left P1)], -- 'Adventure P1' changes state / position
7   [(Left P2)], -- 'Adventure P2' changes state / position
8   [(Left P5)], -- 'Adventure P5' changes state / position
9   [(Left P10)], -- 'Adventure P10' changes state / position
10  [(Left P2), (Left P1)], -- 'Adventure P2 & P1' changes state / position
11  [(Left P5), (Left P1)], -- 'Adventure P5 & P1' changes state / position
12  [(Left P10), (Left P1)], -- 'Adventure P10 & P1' changes state / position
13  [(Left P5), (Left P2)], -- 'Adventure P5 & P2' changes state / position
14  [(Left P10), (Left P2)], -- 'Adventure P10 & P2' changes state / position
15  [(Left P10), (Left P5)] -- 'Adventure P10 & P5' changes state / position
16 ...

```

Listing 2: Optional Task 1, lista de movimentos.

Exemplo de um caso favorável à introdução desta otimização: caso em que o número de movimentos é 6 e queremos um tempo de duração menor ou igual a 17 minutos.

Neste caso como sabemos é possível os aventureiros estarem todos no lado direito da ponte, em segurança, a partir de 5 movimentos. Caso não existisse a possibilidade de não realizar nenhum movimento, ao realizar uma

simulação com 6 iterações seríamos forçados a retirar um dos aventureiros de uma posição de segurança e no final não obteríamos uma solução para o nosso problema.

Com a introdução da possibilidade de não realizar um movimento passamos a evitar este tipo de situações, pois nas nossas simulação vai haver um caminho em que o sexto movimento não é tomado, pois após cinco movimentos os nossos aventureiros já se encontram em segurança.

```
ghci> test3 6 (<=17)
(False,0,[])
```

Figura 2: Resultado da simulação sem a opção de um movimento sem alteração de estado.

```
ghci> test3 6 (<=17)
(True,12,[Duration2 (17,[["True","True","True","True","True"],
[["False","False","False","False","False"]
[["True","True","False","False","True"]
[["False","True","False","False","False"]
[["False","True","True","True","True"]
[["False","False","True","True","False"]
[["True","True","True","True","True"]])
,Duration2 (17 [["True","True","True","True","True"]
[["False","False","False","False","False"]
[["True","True","False","False","True"]
[["True","False","False","False","False"]
[["True","False","True","True","True"]
[["False","False","True","True","False"]
[["True","True","True","True","True"]])])
...
```

Figura 3: Resultado da simulação com a opção de um movimento sem alteração de estado.

Outra vantagem desta otimização é o fornecer soluções em que encurtamos o número de jogadas para o mesmo tempo de duração. Por exemplo, para um tempo de duração de 30 segundos e um máximo de 7 movimentos podemos obter como solução uma sequência de apenas 5 passos.

```
Duration2 (30,[["True","True","True","True","True"]
,[Duration (10,[["False","False","False","False","False"])
,Duration (2,[["False","True","False","True","True"])
,Duration (5,[["False","False","False","True","False"])
,Duration (5,[["False","True","True","True","True"])
,Duration (5,[["False","False","False","True","False"])
,Duration (1,[["True","False","True","True","True"])
,Duration (2,[["False","False","True","True","False"])
,Duration (0,[["True","True","True","True","True"]])])])
```

Figura 4: Resultado da simulação sem a opção de um movimento sem alteração de estado.

Agora com a otimização, figura 5.

```
Duration2 (30,[["True","True","True","True","True"]
,[Duration (10,[["False","False","False","False","False"])
,Duration (5,[["False","False","True","True","True"])
,Duration (5,[["False","False","False","True","False"])
,Duration (5,[["False","True","True","True","True"])
,Duration (5,[["False","True","False","True","False"])
,Duration (0,[["True","True","True","True","True"]])])])
```

Figura 5: Resultado da simulação com a opção de um movimento sem alteração de estado, o que leva a uma menor quantidade de movimentos necessários.

2.1.3 Função *exec*

O objetivo desta próxima função é o de obter todos os *Duration State* possíveis ao fim de um dado número de iterações a partir de um estado inicial dado como argumento. Para isso foi feita uma propagação da função *allValidPlays* aplicada a todos os *Duration State* resultantes de cada iteração *n*.

- 1 — Receives the number *n* (number of individual moves), a function 'allValidPlays' and a 'State',
- 2 — and returns a 'List Duration' with all the moves that the adventures can make.
- 3 `propagate :: Int -> (State -> ListDur State) -> State -> ListDur State`

```

4 propagate 0 _ s = pure s
5 propagate 1 s_l s = s_l s
6 propagate n s_l s = do
7     r <- (s_l s)
8     propagate (n-1) s_l r
9
10 {--- For a given number n and initial state, the function calculates all possible n-sequences
11 of moves that the adventures can make. ---}
12 exec :: Int -> State -> ListDur State
13 exec n s = propagate n (allValidPlays) s

```

Listing 3: Função *exec* e sua função auxiliar *propagate*, a função *exec* faz uso da função *allValidPlays*.

```

ghci> exec 2 gInit
LD [Duration (0,["False","False","False","False","False"]),Duration (1,["True","False","False","False","True"])]
,Duration (2,["False","True","False","False","True"]),Duration (5,["False","False","True","False","True"])]
,Duration (10,["False","False","False","True","True"]),Duration (2,["True","True","False","False","True"])]
,Duration (5,["True","False","True","False","True"]),Duration (10,["True","False","False","True","True"])]
,Duration (5,["False","True","True","False","True"]),Duration (10,["False","True","False","True","True"])]
,Duration (10,["False","False","True","True","True"]),Duration (1,["True","False","False","False","True"])]
,Duration (2,["False","False","False","False","False"]),Duration (2,["False","True","False","False","True"])]
,Duration (4,["False","False","False","False","False"]),Duration (5,["False","False","True","False","True"])]
,Duration (10,["False","False","False","False","False"]),Duration (10,["False","False","False","True","True"])]
,Duration (20,["False","False","False","False","False"]),Duration (2,["True","True","False","False","True"])]
,Duration (3,["False","True","False","False","False"]),Duration (4,["True","False","False","False","False"])]
,Duration (4,["False","False","False","False","False"]),Duration (5,["True","False","True","False","True"])]
,Duration (6,["False","False","True","False","False"]),Duration (10,["True","False","False","False","False"])]
,Duration (10,["False","False","False","False","False"]),Duration (10,["True","False","False","True","True"])]
,Duration (11,["False","False","True","False","False"]),Duration (20,["True","False","False","False","False"])]
,Duration (20,["False","False","False","False","False"]),Duration (5,["False","True","True","False","True"])]
,Duration (7,["False","False","True","False","False"]),Duration (10,["False","True","False","False","False"])]
,Duration (10,["False","False","False","False","False"]),Duration (10,["False","True","False","True","True"])]
,Duration (12,["False","False","False","True","False"]),Duration (20,["False","True","False","False","False"])]
,Duration (20,["False","False","False","False","False"]),Duration (10,["False","False","True","True","True"])]
,Duration (15,["False","False","False","True","False"]),Duration (20,["False","False","True","False","False"])]
,Duration (20,["False","False","False","False","False"])]

```

Figura 6: Execução da função *exec* e o seu *output*.

2.2 Show that it is indeed possible for all adventurers to be on the other side in 17 minutes

2.2.1 Função *lep17*

Esta função diverge das anteriores, permitindo agora testar uma condição: **se é possível todos os aventureiros e lanterna estarem do lado direito ao fim de no máximo 17 minutos e 5 iterações**. Para isso calculou-se a lista de todos os *Duration State* possíveis ao fim das 5 iterações.

Ao resultado do processo anterior aplica-se uma função a cada *Duration State* verifica se a duração da *Duration State* é inferior ou igual a 17 e o estado presente na as *Duration State* tem todos os aventureiros no lado direito da ponte, *Safe location*, retornando *True* ou *False* consuante a *Duration State*.

No final obtemos uma lista de booleanos e verificamos se existe pelo menos um elemento verdadeiro na lista. Se tal for verdade, isso indica que é possível que todos os aventureiros e lanterna estejam do lado direito ao fim de no máximo 17 minutos e 5 iterações.

```

1 --- List of all 'Objects' that are 'Adventurers'
2 players :: [Objetc]
3 players = [(Left P1), (Left P2), (Left P5), (Left P10)]
4
5 --- For a given 'Duration State' checks if the time is <=17 and if all players are at the 'State'
6 --- True, right side of the bridge (Safe!)
7 fun' :: Duration State -> Bool
8 fun' (Duration (i, x)) = ((<= 17) i) && (and (map x players))
9
10 {--- Is it possible for all adventurers to be on the other side in <=17 min and not exceeding
11 5 moves ? ---}
12 lep17 :: Bool
13 lep17 = let r = remLD (exec 5 gInit)

```

```
14      in or (map (fun' ) r)
```

Listing 4: Função *leq17* e suas funções auxiliares, a função *leq17* faz uso da função *exec*.

```
ghci> leq17
True
```

Figura 7: Execução da função *leq17* e o seu output, que como podemos verificar a condição é verdadeira.

2.3 Show that it is impossible for all adventurers to be on the other side in less than 17 minutes

2.3.1 Função *l17*

Esta função pretende verificar **se é possível os aventureiros estarem todos do lado direito sem restrição de iterações em menos de 17 minutos**. O raciocínio adotado foi o seguinte: em 17 minutos o máximo de iterações que poderá existir é 17 uma vez que o aventureiro mais rápido demora 1 minuto a atravessar a ponte. De modo que, colocar um teto de 17, apesar de ser uma estimativa demasiado abrangente, garante que todos os **Duration State** possíveis dentro do tempo limite são incluídos.

De modo que a primeira implementação adotada para este estudo foi idêntico ao anterior mas trocando o número de iterações de 5 para 17 e a condição temporal para < 17 .

```
15  -- For a given 'Duration State' checks if the time is <17 and if all players are at the 'State'
16  -- True, right side of the bridge (Safe!)
17  funn :: Duration State -> Bool
18  funn (Duration (i, x)) = ((< 17) i) && (and (map x players))
19
20  {-- Is it possible for all adventurers to be on the other side in < 17 min ? --}
21  l17 :: Bool
22  l17 = let r = remLD (exec 17 gInit)
23        in or (map (funn) r)
```

Listing 5: Funcao *l17* e suas funcoes auxiliare (a funcao *l17* faz uso da funcao *exec*)

```
ghci> l17
```

Figura 8: Execução da função *l17* e o seu output.

Quando corremos a nossa função verificamos que nenhum dos nossos computadores tem capacidade de chegar a um resultado. Devido às necessidades computacionais do problema quando modelado em **Haskell** não podemos chegar a um resultado.

No entanto se tentarmos verificar a mesma condição, **se é possível os aventureiros estarem todos do lado direito sem restrição de iterações em menos de 17 minutos**, quando o nosso sistema é modelado em **Uppaal** conseguimos chegar á resposta em um segundo. O que demonstra que o **Uppaal** está em vantagem nesta situação quando comparado com o **Haskell**.

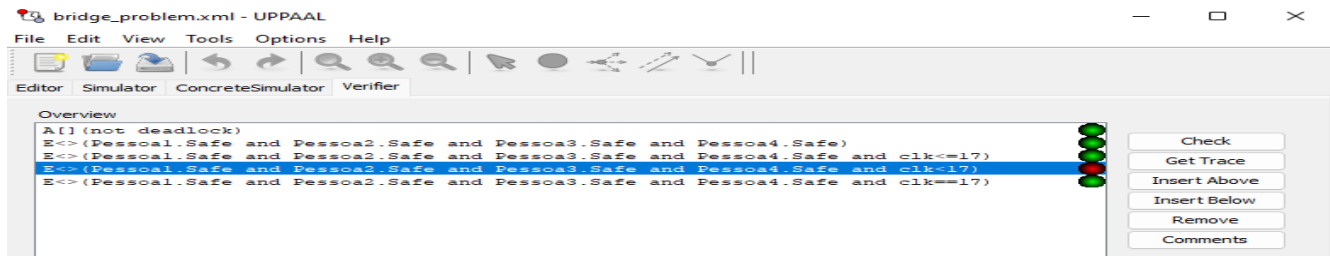


Figura 9: Verificação da nossa condição em Uppaal.

2.4 Optional Task 2

Por forma a melhorar o ambiente de teste foram introduzidas as seguintes funções:

```

24 -- Another way to see if "it is possible for all adventurers to be on the other side in < 17 min"
25 -- is to start with n=1 and see if there are at least one 'State' with time <17 and all the
26 -- 'Adventurers' are in the safe state, if False we increase n to n+1 until we can conclude if
27 -- it is possible for all adventurers to be on the other side in < 17 min.
28
29 -- Gets a condition, p.e (<17), and a 'Duration State' and checks if the time satisfies the
30 -- condition 'f' and all adventures are in the safe state.
31 fun2'' :: (Int -> Bool) -> Duration State -> Bool
32 fun2'' f (Duration (i, x)) = (f i) && (and (map x players))
33
34 -- Gets a n, number of execution, and a condition for the time of a 'Duration State' and checks
35 -- if for all the possible moves in n executions, there are at least one 'Duration State' with
36 -- time that respects the condition 'f' and all the 'Adventurers' in the safe state.
37 test :: Int -> (Int -> Bool) -> Bool
38 test n f = let r = remLD (exec n gInit)
39             in or (map (fun2'' f) r)

```

Listing 6: Optional Task 2, ambiente de teste mais complexo.

Com a função *test* podemos colocar qualquer restrição ao número de jogadas, **n**, e qualquer condição temporal, **f**, às *Duration State*, e verificar se existe pelo menos uma lista de *Duration State* em que estas condições sejam satisfeitas e os aventureiros estejam no lado direito, em segurança.

```

ghci> test 5 (<17)
False
ghci> test 6 (<17)
False
ghci> test 4 (<=17)
False
ghci> test 5 (<=17)
True
ghci> test 6 (<=17)
True
ghci> test 7 (<=30)
True

```

Figura 10: Exemplo da utilização da nova função de teste, *test*

2.5 Implementation of the monad used for the problem of the adventurers

```

40
41 {-- Implementation of the monad used for the problem of the adventurers. Recall the Knight's
42 quest --}
43 data ListDur a = LD [Duration a] deriving Show
44
45 -- Transforms a "ListDur a" to a list of "Duration a"
46 remLD :: ListDur a -> [Duration a]
47 remLD (LD x) = x
48
49 -- Functor ListDur
50 instance Functor ListDur where
51     fmap f = let r = \(Duration (i, a)) -> Duration (i, f a)
52              in LD . ((map r).remLD)
53
54 -- Applicative ListDur
55 instance Applicative ListDur where
56     pure x = LD [Duration (0, x)]
57     l1 <*> l2 = LD $ do r1 <- remLD l1
58                       r2 <- remLD l2

```



```

59         m (r1, r2) where
60             m (Duration (i1, f), Duration(i2, a))
61             = return (Duration(i1 + i2, f a))
62
63 -- Monad ListDur
64 instance Monad ListDur where
65     return = pure
66     l >>= k = LD $ do r <- remLD l
67                   m r where
68                       m (Duration (il, x)) =
69                           let v = remLD (k x)
70                           in map (\(Duration (im, x)) -> Duration (il + im, x)) v
71
72 manyChoice :: [ListDur a] -> ListDur a
73 manyChoice = LD . concat . (map remLD)

```

Listing 7: Instância *Functor ListDur*

2.6 Decisões de implementação

Uma vez que o nosso sistema não apresenta qualquer tipo de memória, rapidamente nos deparamos com o seguinte problema ao testar a condição “é possível todos os aventureiros e lanterna estarem do lado direito ao fim de no máximo 17 minutos e 5 iterações” com `leq17`. Ao recorrer a esta função, o que nós estávamos realmente a testar era se com exatamente 5 iterações era possível encontrar uma solução, ao invés de com um máximo de 5 iterações.

A solução adotada foi introduzir a possibilidade de nenhum aventureiro avançar numa iteração da função `exec`. Deste modo caso seja possível obter o estado final pretendido em menos que 5 iterações, como os aventureiros podem não se mover então obrigatoriamente ira existir uma opção onde eles não se movimentaram mais depois de chegar à solução. Por exemplo, caso se testasse `leq17` para 6 iterações o passávamos de não encontra soluções para encontrar as duas de 5 iterações. Deste modo, ao testar `leq17` para n iterações estamos na verdade a testar para todo o número de iterações inferior a n . Esta solução tenta complementar em parte a ausência de memória uma vez que qualquer estado possível intermédio irá constar na lista de estados finais possíveis.

Deve-se ter em conta contudo complica consideravelmente a complexidade do nosso sistema, tornando-o muito menos eficiente.

3 Second Part

3.1 Your second task is to compare both approaches (via *UPPAAL* and *Haskell*) to the adventurer’s problem. Specifically, you should provide strong and weak points of the two approaches: what are the (dis)advantages of *UPPAAL* for this problem? And what about *Haskell*?

Uma das metodologias adotadas para analisar este problema foi através do programa *UPPAAL*, onde construímos autómatos temporais. Uma das principais vantagens deste processo é a observação de todo o percurso do sistema, estados, transições, etc. Esta análise é uma consequência inerente do uso da ferramenta de simulação do *UPPAAL* e não algo que seja necessário construir, como em *Haskell*. A segunda vantagem encontrada neste trabalho, é a facilidade de estabelecer uma noção temporal ao modelo. Uma vez que o *UPPAAL* é uma ferramenta desenhada para modelar sistemas reais, ao comparar com linguagens de programação genéricas como *Haskell* esta ferramenta claramente é mais vantajosa. Outra característica a favor desta ferramenta é a sua simplicidade. Enquanto que em linguagens de programação existem constrangimentos de tipos de data, variáveis, funções, etc. com o uso de autómatos o programador pode focar-se mais no raciocínio e menos nessas formalidades. Ainda dentro do tópico de opções da própria ferramenta, conceitos como canal ou estado urgentes, entre outros, são uma noção fundamental para vários sistemas reais cuja tradução em linguagens convencionais seria mais trabalhosa.

Por outro lado, esta prática exige uma separação entre objetos ao construir o sistema que não é necessária em código tradicional, onde se modela o sistema como um todo. Este ponto não é necessariamente uma vantagem para quem tiver prática com este tipo de abordagem. Contudo para o grupo com mais experiência em linguagens

de programação tradicionais, a construção individual de um autómato por objeto e a ligação entre ambos é um processo de pensamento diferente ao qual um utilizador terá de se familiarizar.

A segunda metodologia foi implementada neste trabalho e corresponde ao uso de Haskell. A primeira vantagem encontrada é uma maior liberdade na construção do sistema, não só devido a construtores básicos em Haskell mas não disponíveis em UPPAAL como **if then else** como pela ausência de restrições associadas a um autómato.

Em contrapartida a maior dificuldade encontrada foi o aumento rápido da complexidade ao passar para um registo com mais iterações. Esta característica é particularmente relevante ao escolher a permissividade do sistema, onde tem de existir um *trade-off* entre ambos. O facto de maior permissividade implicar o sistema tomar decisões que não são eficientes e um maior número de estados resultantes faz com que o sistema fique mais complexo rapidamente com o aumento de iterações. Outra inconveniência deste mecanismo é a introdução ao conceito de tempo no sistema. Ao invés do UPPAAL, que foi desenhado para ter esse fator em conta, esta linguagem obriga o utilizador a desenvolver um construtor temporal e respetivas instâncias do *monad* temporal.

4 Optional Task 3 - Adventurers.hs

Implementamos uma funcionalidade que permite adicionar uma lista de estados a cada um dos **Duration State**. Para tal foi necessário criar novos *data types* e instanciar novos *monads*.

```
75 {-- Implementation of the monad used for the problem of the adventurers . Recall the Knight's
76 quest --}
77 -- New type of data , ' Duration2 ', same as ' Duration ' but this one has a list of the final and
78 -- previous states , [ States ].
79 data Duration2 a = Duration2 (Int, a, [State]) deriving Show
80
81 getDuration2 :: Duration2 a -> Int
82 getDuration2 (Duration2 (d,_,_)) = d
83
84 getValue2 :: Duration2 a -> a
85 getValue2 (Duration2 (_,x,_)) = x
86
87 getStates2 :: Duration2 a -> [State]
88 getStates2 (Duration2 (_,_,l)) = l
89
90 -- Functor Duration2
91 instance Functor Duration2 where
92   fmap f (Duration2 (i,x,l)) = Duration2 (i, f x, l)
93
94 -- Applicative Duration2
95 instance Applicative Duration2 where
96   pure x = (Duration2 (0,x,[]))
97   (Duration2 (i,f,l)) <*> (Duration2 (j,x,m)) = (Duration2 (i+j, f x, m))
98
99 -- Monad Duration2
100 instance Monad Duration2 where
101   (Duration2 (i,x,l)) >=> k =
102     Duration2 (i + (getDuration2 (k x)), getValue2 (k x), l)
103   return = pure
104
105 -----
106 -- New type of data , ' ListDur2 ', same as ' ListDur ' but this one has a list ' Duration2 '.
107 data ListDur2 a = LD2 [Duration2 a] deriving Show
108
109 -- Transforms a " ListDur2 a " to a list of " Duration2 a "
110 remLD2 :: ListDur2 a -> [Duration2 a]
111 remLD2 (LD2 x) = x
112
113 -- Functor ListDur2
114 instance Functor ListDur2 where
115   fmap f = let r = \(Duration2 (i, a, l)) -> Duration2 (i, f a, l)
116             in LD2 . ((map r).remLD2)
```

```

117
118 -- Applicative ListDur2
119 instance Applicative ListDur2 where
120   pure x = LD2 [Duration2 (0, x, [])]
121   l1 <*> l2 = LD2 $ do r1 <- remLD2 l1
122                       r2 <- remLD2 l2
123                       m (r1, r2) where
124                         m (Duration2 (i1, f, l), Duration2(i2, a, m))
125                           = return (Duration2(i1 + i2, f a, m))
126
127 -- Monad ListDur2
128 instance Monad ListDur2 where
129   return = pure
130   l >>= k = LD2 $ do r <- remLD2 l
131                     m r where
132                       m (Duration2 (il, x', l)) =
133                         let v = remLD2 (k x')
134                         in map (\(Duration2 (im, x, n')) -> Duration2 (il + im, x, l++n')) v

```

Listing 8: Optinal task 3, data types e instancias.

De seguida redefinimos as funções já definidas, **allValidPlays2**, **propagate2** e **exec3**, mas agora para os novos tipos e *monads*.

```

136 -- Changes the 'State' of the 'Objects' inside a list and gives a 'Duration2 State' with the
137 -- time needed to perform the move, the resulting 'State' and puts the state in the list of
138 -- states .
139 fun4 :: State -> [Objetc] -> Duration2 State
140 fun4 s [] = Duration2 (0, s, [])
141 fun4 s (h:t) = let s' = mChangeState ((h:t)+[(Right ())]) s
142               in Duration2 (getTimeAdv((\ (Left a) -> a) h), s', [s])
143
144 {-- For a given state of the game, the function presents all the possible moves that the
145 adventurers can make. --}
146 allValidPlays2 :: State -> ListDur2 State
147 allValidPlays2 s = LD2 (map (fun4 s) (moves s))
148
149 -----
150 -- Receives the number n (number of individual moves), a function 'allValidPlays' and a 'State',
151 -- and returns a 'List Duration' with all the moves that the adventures can make.
152 propagate2 :: Int -> (State -> ListDur2 State) -> State -> ListDur2 State
153 propagate2 0 _ s = LD2 [Duration2 (0, s, [s])]
154 propagate2 n s_l s = do
155   r <- (s_l s)
156   propagate2 (n-1) s_l r
157
158 {-- For a given number n and initial state, the function calculates all possible n-sequences
159 of moves that the adventures can make and the path to get there. --}
160 exec2 :: Int -> State -> ListDur2 State
161 exec2 n s = propagate2 n (allValidPlays2) s

```

Listing 9: Optinal task 3, redefinir de funções.

5 Optional Task 4 - Adventurers.hs

Criamos um novo ambiente de teste que permite não só verificar uma condição, como também saber o número de soluções e os diferentes *Duration State*'s para chegar a essas soluções, isto claro se existirem soluções, pode não existir.

```

162 {-- Test functions --}
163 -- Gets a conditon , f, to be applied to the time of the 'Duration2 State', checks if the
164 -- condition is True and if all the 'Adventurers' are in the safe state , returns True or
165 -- False if both conditions are satisfied .
166 fun'2 :: (Int -> Bool) -> Duration2 State -> Bool
167 fun'2 f (Duration2 (i, x, 1)) = (f i) && (and (map x players))
168
169 -- Gets a n, number of exectution , and a condition for the time of a 'Duration State' and checks
170 -- if for all the possible moves in n executions there are at least one 'Duration State' with
171 -- time that respects the contition 'f' and all the 'Adventurers' in the safe state .
172 -- Returns True or False .
173 test1 :: Int -> (Int -> Bool) -> Bool
174 test1 n f = let r = remLD2 (exec2 n gInit)
175             in or (map (fun'2 f) r)
176
177 -- Gets a n, number of exectution , and a condition for the time of a 'Duration State' and checks
178 -- if for all the possible moves in n executions there are at least one 'Duration State' with
179 -- time that respects the contition 'f' and all the 'Adventurers' in the safe state .
180 -- Returns (tt or ff, n), where 'n' is the number of diferrent solutions to get to a final
181 -- 'Duration State' that satisfies the previous conditions .
182 test2 :: Int -> (Int -> Bool) -> (Bool, Int)
183 test2 n f = let r = remLD2 (exec2 n gInit)
184             l = (filter (fun'2 f) r)
185             in (length(l)/=0, length(l))
186
187 -- Gets a n, number of exectution , and a condition for the time of a 'Duration State' and checks
188 -- if for all the possible moves in n executions there are at least one 'Duration State' with
189 -- time that respects the contition 'f' and all the 'Adventurers' in the safe state .
190 -- Returns (tt or ff, n, l), where 'n' is the number of diferrent solutions to get to a final
191 -- 'Duration State' that satisfies the previous conditions , and 'l' is a list of 'Duration State'
192 -- that that satisfies the previous conditions .
193 test3 :: Int -> (Int -> Bool) -> (Bool, Int, [Duration2 State])
194 test3 n f = let r = remLD2 (exec2 n gInit)
195             l = (filter (fun'2 f) r)
196             in (length(l)/=0, length(l), l)

```

Listing 10: Optinal task 4, novo ambiente de teste.

```

> test1 5 (<=17)
True
> test2 5 (<=17)
(True,2)
> test3 5 (<=17)
(True,2,
 [Duration2 (17,["True","True","True","True","True"],
 ["False","False","False","False","False"]
 ["True","True","False","False","True"]
 ["False","True","False","False","False"]
 ["False","True","True","True","True"]
 ["False","False","True","True","False"]
 ["True","True","True","True","True"]
 ],
 Duration2 (17,["True","True","True","True","True"],
 ["False","False","False","False","False"]
 ["True","True","False","False","True"]
 ["True","False","False","False","False"]
 ["True","False","True","True","True"]
 ["False","False","True","True","False"]
 ["True","True","True","True","True"]]))

```

Figura 11: Exemplo da utilização das novas funções de teste, *test*

As funções `teste1`, `teste2` e `teste3` definem diferentes funções de teste e cada uma vai fornecendo mais informação face à anterior.

6 Optional Task 5 - Adventurers2.hs

Para um melhor análise da sequência de estados dos movimentos dos aventureiros, tivemos a ideia de complementar a nova estrutura de dados anteriormente introduzida, *Duration2 State*, com uma lista de *Duration States*, em que cada *Duration States* dá-nos acesso ao estado e à sua duração. Com esta alteração temos a possibilidade de acompanhar os diferentes estados mas também a sua duração ao longo dos movimentos dos aventureiros.

Isto levou à criação de novos *data type's* e novas instâncias de *monads*. Esta nova implementação encontra-se no ficheiro **Adventurers2.hs**.

```

197 -- New type of data, 'Duration2', same as 'Duration' but this one has a list of the final and
198 -- previous duration states, [Duration State].
199 data Duration2 a = Duration2 (Int, a, [Duration State]) deriving Show
200
201 getDuration2 :: Duration2 a -> Int
202 getDuration2 (Duration2 (d,_,_)) = d
203
204 getValue2 :: Duration2 a -> a
205 getValue2 (Duration2 (_,x,_)) = x
206
207 getStates2 :: Duration2 a -> [Duration State]
208 getStates2 (Duration2 (_,_,l)) = l
209
210 -- Functor Duration2
211 instance Functor Duration2 where
212   fmap f (Duration2 (i,x,l)) = Duration2 (i, f x, l)
213
214 -- Applicative Duration2
215 instance Applicative Duration2 where
216   pure x = (Duration2 (0,x,[]))
217   (Duration2 (i,f,l)) <*> (Duration2 (j,x,m)) = (Duration2 (i+j, f x, m))
218
219 -- Monad Duration2
220 instance Monad Duration2 where
221   (Duration2 (i,x,l)) >>= k =
222     Duration2 (i + (getDuration2 (k x)), getValue2 (k x), l)
223   return = pure
224
225 -----
226 -- New type of data, 'ListDur2', same as 'ListDur' but this one has a list 'Duration2'.
227 data ListDur2 a = LD2 [Duration2 a] deriving Show
228
229 -- Transforms a "ListDur2 a" to a list of "Duration2 a"
230 remLD2 :: ListDur2 a -> [Duration2 a]
231 remLD2 (LD2 x) = x
232
233 -- Functor ListDur2
234 instance Functor ListDur2 where
235   fmap f = let r = \(Duration2 (i, a, l)) -> Duration2 (i, f a, l)
236             in LD2 . ((map r).remLD2)
237
238 -- Applicative ListDur2
239 instance Applicative ListDur2 where
240   pure x = LD2 [Duration2 (0, x, [])]
241   l1 <*> l2 = LD2 $ do r1 <- remLD2 l1
242                       r2 <- remLD2 l2
243                       m (r1, r2) where
244                         m (Duration2 (i1, f, l), Duration2 (i2, a, m))

```

```

245         = return (Duration2(i1 + i2, f a, m))
246
247 -- Monad ListDur2
248 instance Monad ListDur2 where
249     return = pure
250     l >=> k = LD2 $ do r <- remLD2 l
251                   m r where
252                   m (Duration2 (il, x', l)) =
253                     let v = remLD2 (k x')
254                     in map (\(Duration2 (im, x, n')) -> Duration2 (il + im, x, l+n')) v

```

Listing 11: Optimal task 5, data types e instâncias.

De seguida redefinimos as funções já definidas, **allValidPlays2**, **propagate2** e **exec3**, mas agora para os novos tipos e *monads*.

```

256 -- Changes the 'State' of the 'Objects' inside a list and gives a 'Duration2 State' with the
257 -- time needed to perform the move, the resulting 'State' and puts the state in the list of
258 -- states .
259 fun4 :: State -> [Objetc] -> Duration2 State
260 fun4 s [] = Duration2 (0, s, [])
261 fun4 s (h:t) = let s' = mChangeState ((h:t)++[(Right ())]) s
262               i = getTimeAdv((\ (Left a) -> a) h)
263               in Duration2 (i, s', [Duration (i,s)])
264
265 allValidPlays2 :: State -> ListDur2 State
266 allValidPlays2 s = LD2 (map (fun4 s) (moves s))
267
268 -----
269 -- Receives the number n (number of individual moves), a function 'allValidPlays' and a 'State',
270 -- and returns a 'List Duration' with all the moves that the adventures can make.
271 propagate2 :: Int -> (State -> ListDur2 State) -> State -> ListDur2 State
272 propagate2 0 _ s = LD2 [Duration2 (0, s, [Duration (0, s)])]
273 propagate2 n s_l s = do
274     r <- (s_l s)
275     propagate2 (n-1) s_l r
276
277 {-- For a given number n and initial state, the function calculates all possible n-sequences
278 of moves that the adventures can make and the path to get there. --}
279 exec2 :: Int -> State -> ListDur2 State
280 exec2 n s = propagate2 n (allValidPlays2) s

```

Listing 12: Optimal task 5, redefinir de funções.

```

ghci> allValidPlays2 gInit
LD2 [Duration2 (0,["False","False","False","False","False"],[]),
Duration2 (1,["True","False","False","False","True"],
[Duration (1,["False","False","False","False","False"])]),
Duration2 (2,["False","True","False","False","True"],
[Duration (2,["False","False","False","False","False"])]),
Duration2 (5,["False","False","True","False","True"],
[Duration (5,["False","False","False","False","False"])]),
Duration2 (10,["False","False","False","True","True"],
[Duration (10,["False","False","False","False","False"])]),
Duration2 (2,["True","True","False","False","True"],
[Duration (2,["False","False","False","False","False"])]),
Duration2 (5,["True","False","True","False","True"],
[Duration (5,["False","False","False","False","False"])]),
Duration2 (10,["True","False","False","True","True"],
[Duration (10,["False","False","False","False","False"])]),
Duration2 (5,["False","True","True","False","True"],
[Duration (5,["False","False","False","False","False"])]),
Duration2 (10,["False","True","False","True","True"],
[Duration (10,["False","False","True","True","True"])]),
Duration2 (10,["False","False","True","True","True"],
[Duration (10,["False","False","False","False","False"])])]

```

Figura 12: Exemplo da utilização da nova função *allValidPlays2*.

Figura 13: Exemplo da utilização da nova função *exec2*.

7 Optional Task 6 - Adventurers.hs

Criamos um novo ambiente de teste que permite não só verificar uma condição, como também saber o número de soluções e os diferentes *Duration State's* para chegar a essas soluções, isto claro se existirem soluções, pode não existir.

Para este novo tipo de *data type* também implementamos um novo ambiente de teste.

```

281 -- Gets a n, number of execution , and a condition for the time of a 'Duration State' and checks
282 -- if for all the possible moves in n executions there are at least one 'Duration State' with
283 -- time that respects the condition 'f' and all the 'Adventurers' in the safe state .
284 -- Returns (tt or ff, n, l), where 'n' is the number of different solutions to get to a final
285 -- 'Duration State' that satisfies the previous conditions , and 'l' is a list of 'Duration2 State'
286 -- that that satisfies the previous conditions .
287 test3 :: Int -> (Int -> Bool) -> (Bool, Int, [Duration2 State])
288 test3 n f = let r = remLD2 (exec2 n gInit)
289             l = (filter (fun'2 f) r)
290             in (length(l)/=0, length(l), l)
291
292 \begin{lstlisting}[caption={Optimal task 6, novo ambiente de teste.},captionpos=b]]
293 {-- Test functions --}
294 -- Gets a condition , f, to be applied to the time of the 'Duration2 State', checks if the
295 -- condition is True and if all the 'Adventurers' are in the safe state , returns True or
296 -- False if both conditions are satisfied .
297 fun'2 :: (Int -> Bool) -> Duration2 State -> Bool
298 fun'2 f (Duration2 (i, x, l)) = (f i) && (and (map x players))
299
300 -- Gets a n, number of execution , and a condition for the time of a 'Duration State' and checks
301 -- if for all the possible moves in n executions there are at least one 'Duration State' with
302 -- time that respects the condition 'f' and all the 'Adventurers' in the safe state .
303 -- Returns True or False .
304 test1 :: Int -> (Int -> Bool) -> Bool
305 test1 n f = let r = remLD2 (exec2 n gInit)
306             in or (map (fun'2 f) r)
307
308 -- Gets a n, number of execution , and a condition for the time of a 'Duration State' and checks
309 -- if for all the possible moves in n executions there are at least one 'Duration State' with

```



```

310 -- time that respects the contition 'f' and all the 'Adventurers' in the safe state .
311 -- Returns (tt or ff, n), where 'n' is the number of diferrent solutions to get to a final
312 -- 'Duration State' that satisfies the previous conditions .
313 test2 :: Int -> (Int -> Bool) -> (Bool, Int)
314 test2 n f = let r = remLD2 (exec2 n gInit)
315             l = (filter (fun'2 f) r)
316             in (length(l)/=0, length(l))
317
318 -- Gets a n, number of exectution , and a condition for the time of a 'Duration State' and checks
319 -- if for all the possible moves in n executions there are at least one 'Duration State' with
320 -- time that respects the contition 'f' and all the 'Adventurers' in the safe state .
321 -- Returns (tt or ff, n, l), where 'n' is the number of diferrent solutions to get to a final
322 -- 'Duration State' that satisfies the previous conditions , and 'l' is a list of 'Duration2 State'
323 -- that that satisfies the previous conditions .
324 test3 :: Int -> (Int -> Bool) -> (Bool, Int, [Duration2 State])
325 test3 n f = let r = remLD2 (exec2 n gInit)
326             l = (filter (fun'2 f) r)
327             in (length(l)/=0, length(l), l)

```

Listing 13: Optinal task 6, novo ambiente de teste.

```

> test1 5 (<=17)
True
> test2 5 (<=17)
(True,2)
> test3 5 (<=17)
(True,2,
 [Duration2 (17,["True", "True", "True", "True", "True"],
 [Duration (2,["False", "False", "False", "False", "False"]),
 ,Duration (1,["True", "True", "False", "False", "True"]),
 ,Duration (10,["False", "True", "False", "False", "False"]),
 ,Duration (2,["False", "True", "True", "True", "True"]),
 ,Duration (2,["False", "False", "True", "True", "False"]),
 ,Duration (0,["True", "True", "True", "True", "True"])]),
 [Duration2 (17,["True", "True", "True", "True", "True"],
 [Duration (2,["False", "False", "False", "False", "False"]),
 ,Duration (2,["True", "True", "False", "False", "True"]),
 ,Duration (10,["True", "False", "False", "False", "False"]),
 ,Duration (1,["True", "False", "True", "True", "True"]),
 ,Duration (2,["False", "False", "True", "True", "False"]),
 ,Duration (0,["True", "True", "True", "True", "True"])]])

```

Figura 14: Exemplo da utilização das novas funções de teste, *test*

8 Conclusion

Concluída assim a modelização do sistema em Haskell, obtivemos não só a solução esperada como uma nova implementação que permite analisar o problema dos aventureiros.

O desenvolvimento desta nova perspectiva envolveu algumas dificuldades, sendo a principal a otimização do modelo sem criar restrições de decisão. Ao implementar decisões que apesar de possíveis não são as mais eficientes como nenhum aventureiro se mover ou não obrigar a passarem dois aventureiros para a direita quando possível, estamos a dar mais liberdade ao modelo mas também aumentar a árvore de possíveis caminhos. De modo que o maior desafio foi como criar o modelo mais permissivo possível mas que seja suficientemente otimizado para ter tempos de execução razoáveis para muitas iterações.

Apesar de grupo considerar que o trabalho foi concluído com sucesso, seria ainda pertinente aplicar algumas melhorias. A mais pertinente mas também mais desafiante seria implementar uma memória ao programa. Esta *feature* permitiria não só uma melhor análise do sistema mas também saber qual a combinação de movimentos associada à solução do problema. A nossa implementação permite apenas verificar a sua existência.

Outra melhoria seria uma cláusula na mudança de estado onde caso esta implicasse mudar para um estado maior a 17 minutos ou sair da posição final pretendida, não se aplicaria mais nenhuma mudança. A implementação adota permite adicionar uma lista de objetos nula de modo a tentar controlar esta situação. Isto é, se existir um estado 's' na iteração que corresponde à solução, então a iteração seguinte poderá ser através da lista de objetos

[], mantendo assim o estado. Deste modo, casos em que o sistema passa pela solução mas como é obrigado a efetuar mais uma iteração volta a sair do estado pretendido não acontecem. Claramente esta situação não é de todo a ideal uma vez que apesar de não "perdermos" a solução estamos a aumentar significativamente o tamanho das nossas propagações.

De forma a tentar contrariar a complexidade crescente do problema devia ainda existir uma melhoria geral na eficiência das nossas funções. Não só recorrendo a métodos mais otimizados como a reduzir as passagens de tipos abundantes no nosso código.

De qualquer modo consideramos que este trabalho foi concluído com sucesso apesar das dificuldades encontradas e permitiu aplicar, consolidar e comparar os conhecimentos lecionados ao longo do semestre, o que o grupo considerou interessante.

9 Attachments

```
328 {-# LANGUAGE FlexibleInstances #-}
329 module Adventurers where
330
331 import DurationMonad
332
333 -- List of adventurers
334 data Adventurers = P1 | P2 | P5 | P10 deriving (Show, Eq)
335
336 -- Objects of the game (Adventurers + Lantern)
337 type Object = Either Adventurers ()
338
339 -- Time that each adventurer needs to cross the bridge
340 getTimeAdv :: Adventurers -> Int
341 getTimeAdv P1 = 1
342 getTimeAdv P2 = 2
343 getTimeAdv P5 = 5
344 getTimeAdv P10 = 10
345
346 -----
347 {- Game State MEMORY:
348  - The state of the game, i.e the current position of each object (Adventurers and Lantern).
349  - The function (const False) represents the initial state of the game, with all Adventurers
350    and the Lantern on the left side of the bridge.
351  - Similarly, the function (const True) represents the end state of the game, with all
352    adventurers and the lantern on the right side of the bridge. --}
353
354 -- Game Memory
355 type State = Object -> Bool
356
357 instance Show State where
358     show s = (show . (fmap show)) [s (Left P1),
359                                     s (Left P2),
360                                     s (Left P5),
361                                     s (Left P10),
362                                     s (Right ())]
363
364 instance Eq State where
365     (==) s1 s2 = and [s1 (Left P1) == s2 (Left P1),
366                       s1 (Left P2) == s2 (Left P2),
367                       s1 (Left P5) == s2 (Left P5),
368                       s1 (Left P10) == s2 (Left P10),
369                       s1 (Right ()) == s2 (Right ())]
370
371 -----
372 -- Initial state of the Game
373 gInit :: State
374 gInit = const False
375
376 -- Desired final state of the Game
377 gEnd :: State
378 gEnd = const True
379
380 -- Changes the 'State' of the game for a given 'Object'
381 changeState :: Object -> State -> State
382 changeState a s = let v = s a
383                   in (\x -> if x == a then not v else s x)
384
385 -- Changes the 'State' of the Game of a list of 'Objects'
386 mChangeState :: [Object] -> State -> State
```

```

387 mChangeState os s = foldr changeState s os
388
389 -- List of all moves
390 possibleMoves :: [[Objetc]]
391 possibleMoves = [
392   [], -- No 'Object' changes state / position {-- Optional Task 1 --}
393   [(Left P1)], -- 'Adventure P1' changes state / position
394   [(Left P2)], -- 'Adventure P2' changes state / position
395   [(Left P5)], -- 'Adventure P5' changes state / position
396   [(Left P10)], -- 'Adventure P10' changes state / position
397   [(Left P2), (Left P1)], -- 'Adventure P2 & P1' changes state / position
398   [(Left P5), (Left P1)], -- 'Adventure P5 & P1' changes state / position
399   [(Left P10), (Left P1)], -- 'Adventure P10 & P1' changes state / position
400   [(Left P5), (Left P2)], -- 'Adventure P5 & P2' changes state / position
401   [(Left P10), (Left P2)], -- 'Adventure P10 & P2' changes state / position
402   [(Left P10), (Left P5)] -- 'Adventure P10 & P5' changes state / position
403
404 -- Given a 'State' calculates the list of possible moves with respect to the 'Latern' position
405 moves :: State -> [[Objetc]]
406 moves s = filter (\list -> and (map (\a -> (s a) == (s (Right ()))) list)) possibleMoves
407
408 -- Changes the 'State' of the 'Objects' inside a list and gives a 'Duration State' with the
409 -- time needed to perform the move and the resulting 'State'
410 fun :: State -> [Objetc] -> Duration State
411 fun s [] = Duration (0, s)
412 fun s (h:t) = Duration (getTimeAdv((\ (Left a) -> a) h), mChangeState ((h:t)++[(Right ())] s)
413
414 {-- For a given state of the game, the function presents all the
415 possible moves that the adventurers can make. --}
416 allValidPlays :: State -> ListDur State
417 allValidPlays s = LD (map (fun s) (moves s))
418
419 -----
420 -- Receives the number n (number of individual moves), a function 'allValidPlays' and a 'State',
421 -- and returns a 'List Duration' with all the moves that the adventures can make.
422 propagate :: Int -> (State -> ListDur State) -> State -> ListDur State
423 propagate 0 _ s = pure s
424 propagate 1 s_l s = s_l s
425 propagate n s_l s = do
426   r <- (s_l s)
427   propagate (n-1) s_l r
428
429 {-- For a given number n and initial state, the function calculates all possible n-sequences
430 of moves that the adventures can make. --}
431 exec :: Int -> State -> ListDur State
432 exec n s = propagate n (allValidPlays) s
433
434 -----
435 -- List of all 'Objects' that are 'Adventurers'
436 players :: [Objetc]
437 players = [(Left P1), (Left P2), (Left P5), (Left P10)]
438
439 -- For a given 'Duration State' checks if the time is <=17 and if all players are at the 'State'
440 -- True, right side of the bridge (Safe!)
441 fun' :: Duration State -> Bool
442 fun' (Duration (i, x)) = ((<= 17) i) && (and (map x players))
443
444 {-- Is it possible for all adventurers to be on the other side in <=17 min and not exceeding
445 5 moves ? --}
446 leq17 :: Bool
447 leq17 = let r = remLD (exec 5 gInit)

```

```
448         in or (map (fun') r)
449
450 -----
451 -- 1st approach
452 -----
453 -- For a given 'Duration State' checks if the time is <17 and if all players are at the 'State'
454 -- True, right side of the bridge (Safe!)
455 funn :: Duration State -> Bool
456 funn (Duration (i, x)) = ((< 17) i) && (and (map x players))
457
458 {-- Is it possible for all adventurers to be on the other side in < 17 min ? --}
459 l17 :: Bool
460 l17 = let r = remLD (exec 17 gInit)
461       in or (map (funn) r)
462
463 -----
464 {-- Optional Task 2 --}
465 -----
466 -- Another way to see if "it is possible for all adventurers to be on the other side in < 17 min"
467 -- is to star with n=1 and see if there are at least one 'State' with time <17 and all the
468 -- 'Adventurers' are in the safe state, if False we increase n to n+1 until we can conclude if
469 -- it is possible for all adventurers to be on the other side in < 17 min.
470
471 -- Gets a condition, p.e (<17), and a 'Duration State' and checks if the time satisfies the
472 -- condition 'f' and all adventures are in the safe state.
473 fun2'' :: (Int -> Bool) -> Duration State -> Bool
474 fun2'' f (Duration (i, x)) = (f i) && (and (map x players))
475
476 -- Gets a n, number of execution, and a condition for the time of a 'Duration State' and checks
477 -- if for all the possible moves in n executions, there are at least one 'Duration State' with
478 -- time that respects the condition 'f' and all the 'Adventurers' in the safe state.
479 test :: Int -> (Int -> Bool) -> Bool
480 test n f = let r = remLD (exec n gInit)
481           in or (map (fun2'' f) r)
482
483 -----
484 -- Example:
485 -- > test 1 (<17)      > test 1 (<=17)      > test 1 (<19)
486 -- False              False              False
487 -- > test 2 (<17)      > test 2 (<=17)      > test 2 (<19)
488 -- False              False              False
489 -- > test 3 (<17)      > test 3 (<=17)      > test 3 (<19)
490 -- False              False              False
491 -- > test 4 (<17)      > test 4 (<=17)      > test 4 (<19)
492 -- False              False              False
493 -- > test 5 (<17)      > test 5 (<=17)      > test 5 (<17)
494 -- False              True               True
495 -- > test 6 (<17)      > test 6 (<=17)      > test 6 (<17)
496 -- False              True               True
497
498 -----
499 {-- Implementation of the monad used for the problem of the adventurers. Recall the Knight's
500 quest --}
501 data ListDur a = LD [Duration a] deriving Show
502
503 -- Transforms a "ListDur a" to a list of "Duration a"
504 remLD :: ListDur a -> [Duration a]
505 remLD (LD x) = x
506
507 -- Functor ListDur
508 instance Functor ListDur where
```

```
509   fmap f = let r = \(Duration (i, a)) -> Duration (i, f a)
510             in LD . ((map r).remLD)
511
512 -- Applicative ListDur
513 instance Applicative ListDur where
514   pure x = LD [Duration (0, x)]
515   l1 <*> l2 = LD $ do r1 <- remLD l1
516                     r2 <- remLD l2
517                     m (r1, r2) where
518                       m (Duration (i1, f), Duration(i2, a))
519                         = return (Duration(i1 + i2, f a))
520
521 -- Monad ListDur
522 instance Monad ListDur where
523   return = pure
524   l >>= k = LD $ do r <- remLD l
525                   m r where
526                     m (Duration (il, x)) =
527                       let v = remLD (k x)
528                       in map \(Duration (im, x)) -> Duration (il + im, x)) v
529
530 manyChoice :: [ListDur a] -> ListDur a
531 manyChoice = LD . concat . (map remLD)
532
533 -----
534 {-- Optional Task 3 --}
535 -----
536 -- New type of data, 'Duration2', same as 'Duration' but this one has a list of the final and
537 -- previous states, [States].
538 data Duration2 a = Duration2 (Int, a, [State]) deriving Show
539
540 getDuration2 :: Duration2 a -> Int
541 getDuration2 (Duration2 (d,_,_)) = d
542
543 getValue2 :: Duration2 a -> a
544 getValue2 (Duration2 (_,x,_)) = x
545
546 getStates2 :: Duration2 a -> [State]
547 getStates2 (Duration2 (_,_,l)) = l
548
549 -- Functor Duration2
550 instance Functor Duration2 where
551   fmap f (Duration2 (i,x,l)) = Duration2 (i, f x, l)
552
553 -- Applicative Duration2
554 instance Applicative Duration2 where
555   pure x = (Duration2 (0,x,[]))
556   (Duration2 (i,f,l)) <*> (Duration2 (j,x,m)) = (Duration2 (i+j, f x, m))
557
558 -- Monad Duration2
559 instance Monad Duration2 where
560   (Duration2 (i,x,l)) >>= k =
561     Duration2 (i + (getDuration2 (k x)), getValue2 (k x), l)
562   return = pure
563
564 -----
565 -- New type of data, 'ListDur2', same as 'ListDur' but this one has a list 'Duration2'.
566 data ListDur2 a = LD2 [Duration2 a] deriving Show
567
568 -- Transforms a "ListDur2 a" to a list of "Duration2 a"
569 remLD2 :: ListDur2 a -> [Duration2 a]
```

```
570 remLD2 (LD2 x) = x
571
572 -- Functor ListDur2
573 instance Functor ListDur2 where
574   fmap f = let r = \(Duration2 (i, a, l)) -> Duration2 (i, f a, l)
575             in LD2 . ((map r).remLD2)
576
577 -- Applicative ListDur2
578 instance Applicative ListDur2 where
579   pure x = LD2 [Duration2 (0, x, [])]
580   l1 <*> l2 = LD2 $ do r1 <- remLD2 l1
581                       r2 <- remLD2 l2
582                       m (r1, r2) where
583                         m (Duration2 (i1, f, l), Duration2 (i2, a, m))
584                           = return (Duration2 (i1 + i2, f a, m))
585
586 -- Monad ListDur2
587 instance Monad ListDur2 where
588   return = pure
589   l >>= k = LD2 $ do r <- remLD2 l
590                     m r where
591                       m (Duration2 (il, x', l)) =
592                         let v = remLD2 (k x')
593                         in map \(Duration2 (im, x, n')) -> Duration2 (il + im, x, l++n') v
594
595 -----
596 -- Changes the 'State' of the 'Objects' inside a list and gives a 'Duration2 State' with the
597 -- time needed to perform the move, the resulting 'State' and puts the state in the list of
598 -- states.
599 fun4 :: State -> [Objetc] -> Duration2 State
600 fun4 s [] = Duration2 (0, s, [])
601 fun4 s (h:t) = let s' = mChangeState ((h:t)++[(Right ())]) s
602               in Duration2 (getTimeAdv(\(Left a) -> a) h), s', [s])
603
604 {-- For a given state of the game, the function presents all the possible moves that the
605 adventurers can make. --}
606 allValidPlays2 :: State -> ListDur2 State
607 allValidPlays2 s = LD2 (map (fun4 s) (moves s))
608
609 -----
610 -- Receives the number n (number of individual moves), a function 'allValidPlays' and a 'State',
611 -- and returns a 'List Duration' with all the moves that the adventures can make.
612 propagate2 :: Int -> (State -> ListDur2 State) -> State -> ListDur2 State
613 propagate2 0 _ s = LD2 [Duration2 (0, s, [s])]
614 propagate2 n s_l s = do
615   r <- (s_l s)
616   propagate2 (n-1) s_l r
617
618 {-- For a given number n and initial state, the function calculates all possible n-sequences
619 of moves that the adventures can make and the path to get there. --}
620 exec2 :: Int -> State -> ListDur2 State
621 exec2 n s = propagate2 n (allValidPlays2) s
622
623 -----
624 {-- Optional Task 4 --}
625 -----
626 {-- Test functions --}
627 -- Gets a conditon, f, to be applied to the time of the 'Duration2 State', checks if the
628 -- condition is True and if all the 'Adventurers' are in the safe state, returns True or
629 -- False if both conditions are satisfied.
630 fun'2 :: (Int -> Bool) -> Duration2 State -> Bool
```

```

631 fun'2 f (Duration2 (i, x, l)) = (f i) && (and (map x players))
632
633 -- Gets a n, number of execution, and a condition for the time of a 'Duration State' and checks
634 -- if for all the possible moves in n executions there are at least one 'Duration State' with
635 -- time that respects the condition 'f' and all the 'Adventurers' in the safe state.
636 -- Returns True or False.
637 test1 :: Int -> (Int -> Bool) -> Bool
638 test1 n f = let r = remLD2 (exec2 n gInit)
639             in or (map (fun'2 f) r)
640
641 -- Gets a n, number of execution, and a condition for the time of a 'Duration State' and checks
642 -- if for all the possible moves in n executions there are at least one 'Duration State' with
643 -- time that respects the condition 'f' and all the 'Adventurers' in the safe state.
644 -- Returns (tt or ff, n), where 'n' is the number of different solutions to get to a final
645 -- 'Duration State' that satisfies the previous conditions.
646 test2 :: Int -> (Int -> Bool) -> (Bool, Int)
647 test2 n f = let r = remLD2 (exec2 n gInit)
648             l = (filter (fun'2 f) r)
649             in (length(l)/=0, length(l))
650
651 -- Gets a n, number of execution, and a condition for the time of a 'Duration State' and checks
652 -- if for all the possible moves in n executions there are at least one 'Duration State' with
653 -- time that respects the condition 'f' and all the 'Adventurers' in the safe state.
654 -- Returns (tt or ff, n, l), where 'n' is the number of different solutions to get to a final
655 -- 'Duration State' that satisfies the previous conditions, and 'l' is a list of 'Duration State'
656 -- that that satisfies the previous conditions.
657 test3 :: Int -> (Int -> Bool) -> (Bool, Int, [Duration2 State])
658 test3 n f = let r = remLD2 (exec2 n gInit)
659             l = (filter (fun'2 f) r)
660             in (length(l)/=0, length(l), l)
661
662 -----
663 -- Example:
664
665 -- > test1 5 (<=17)
666 -- True
667
668 -- > test2 5 (<=17)
669 -- (True,2)
670
671 -- > test3 5 (<=17)
672 -- (True,2,
673 -- [Duration2 (17,[ " True "," True "," True "," True "," True "],
674 -- [" False "," False "," False "," False "," False "],
675 -- [" True "," True "," False "," False "," True "],
676 -- [" False "," True "," False "," False "," False "],
677 -- [" False "," True "," True "," True "," True "],
678 -- [" False "," False "," True "," True "," False "],
679 -- [" True "," True "," True "," True "," True "])])
680 -- ,Duration2 (17,[ " True "," True "," True "," True "," True "],
681 -- [" False "," False "," False "," False "," False "],
682 -- [" True "," True "," False "," False "," True "],
683 -- [" True "," False "," False "," False "," False "],
684 -- [" True "," False "," True "," True "," True "],
685 -- [" False "," False "," True "," True "," False "],
686 -- [" True "," True "," True "," True "," True "])]) ])
```

Listing 14: Adventurers.hs

```

687 {-# LANGUAGE FlexibleInstances #-}
688 module Adventurers where
```

```
689
690 import DurationMonad
691
692 -- List of adventurers
693 data Adventurers = P1 | P2 | P5 | P10 deriving (Show, Eq)
694
695 -- Objects of the game (Adventurers + Lantern)
696 type Object = Either Adventurers ()
697
698 -- Time that each adventurer needs to cross the bridge
699 getTimeAdv :: Adventurers -> Int
700 getTimeAdv P1 = 1
701 getTimeAdv P2 = 2
702 getTimeAdv P5 = 5
703 getTimeAdv P10 = 10
704
705 -----
706 {-- Game State MEMORY:
707  - The state of the game, i.e the current position of each object (Adventurers and Lantern).
708  - The function (const False) represents the initial state of the game, with all Adventurers
709    and the Lantern on the left side of the bridge.
710  - Similarly, the function (const True) represents the end state of the game, with all
711    adventurers and the lantern on the right side of the bridge. --}
712
713 -- Game Memory
714 type State = Object -> Bool
715
716 instance Show State where
717     show s = (show . (fmap show)) [s (Left P1),
718                                     s (Left P2),
719                                     s (Left P5),
720                                     s (Left P10),
721                                     s (Right ())]
722
723 instance Eq State where
724     (==) s1 s2 = and [s1 (Left P1) == s2 (Left P1),
725                       s1 (Left P2) == s2 (Left P2),
726                       s1 (Left P5) == s2 (Left P5),
727                       s1 (Left P10) == s2 (Left P10),
728                       s1 (Right ()) == s2 (Right ())]
729
730 -----
731 -- Initial state of the Game
732 gInit :: State
733 gInit = const False
734
735 -- Desired final state of the Game
736 gEnd :: State
737 gEnd = const True
738
739 -- Changes the 'State' of the game for a given 'Object'
740 changeState :: Object -> State -> State
741 changeState a s = let v = s a
742                   in (\x -> if x == a then not v else s x)
743
744 -- Changes the 'State' of the Game of a list of 'Objects'
745 mChangeState :: [Object] -> State -> State
746 mChangeState os s = foldr changeState s os
747
748 -- List of all moves
749 possibleMoves :: [[Object]]
```



```

750 possibleMoves = [
751   [],                -- No 'Object' changes state / position {-- Optional Task 1 --}
752   [(Left P1)],       -- 'Adventure P1' changes state / position
753   [(Left P2)],       -- 'Adventure P2' changes state / position
754   [(Left P5)],       -- 'Adventure P5' changes state / position
755   [(Left P10)],      -- 'Adventure P10' changes state / position
756   [(Left P2), (Left P1)], -- 'Adventure P2 & P1' changes state / position
757   [(Left P5), (Left P1)], -- 'Adventure P5 & P1' changes state / position
758   [(Left P10), (Left P1)], -- 'Adventure P10 & P1' changes state / position
759   [(Left P5), (Left P2)], -- 'Adventure P5 & P2' changes state / position
760   [(Left P10), (Left P2)], -- 'Adventure P10 & P2' changes state / position
761   [(Left P10), (Left P5)] -- 'Adventure P10 & P5' changes state / position
762
763 -- Given a 'State' calculates the list of possible moves with respect to the 'Latern' position
764 moves :: State -> [[Objetc]]
765 moves s = filter (\list -> and (map (\a -> (s a) == (s (Right ()))) list)) possibleMoves
766
767 -- Changes the 'State' of the 'Objects' inside a list and gives a 'Duration State' with the
768 -- time needed to perform the move and the resulting 'State'
769 fun :: State -> [Objetc] -> Duration State
770 fun s [] = Duration (0, s)
771 fun s (h:t) = Duration (getTimeAdv((\ (Left a) -> a) h), mChangeState ((h:t)++[(Right ())]) s)
772
773 {-- For a given state of the game, the function presents all the
774 possible moves that the adventurers can make. --}
775 allValidPlays :: State -> ListDur State
776 allValidPlays s = LD (map (fun s) (moves s))
777
778 -----
779 -- Receives the number n (number of individual moves), a function 'allValidPlays' and a 'State',
780 -- and returns a 'List Duration' with all the moves that the adventures can make.
781 propagate :: Int -> (State -> ListDur State) -> State -> ListDur State
782 propagate 0 _ s = pure s
783 propagate 1 s_l s = s_l s
784 propagate n s_l s = do
785     r <- (s_l s)
786     propagate (n-1) s_l r
787
788 {-- For a given number n and initial state, the function calculates all possible n-sequences
789 of moves that the adventures can make. --}
790 exec :: Int -> State -> ListDur State
791 exec n s = propagate n (allValidPlays) s
792
793 -----
794 -- List of all 'Objects' that are 'Adventurers'
795 players :: [Objetc]
796 players = [(Left P1), (Left P2), (Left P5), (Left P10)]
797
798 -- For a given 'Duration State' checks if the time is <=17 and if all players are at the 'State'
799 -- True, right side of the bridge (Safe!)
800 fun' :: (Int -> Bool) -> Duration State -> Bool
801 fun' f (Duration (i, x)) = (f i) && (and (map x players))
802
803 {-- Is it possible for all adventurers to be on the other side in <=17 min and not exceeding
804 5 moves ? --}
805 leq17 :: Bool
806 leq17 = let r = remLD (exec 5 gInit)
807         in or (map (fun' (<= 17)) r)
808
809 -----
810 -- 1st approach

```

```

811 -----
812 -- For a given 'Duration State' checks if the time is <17 and if all players are at the 'State'
813 -- True, right side of the bridge (Safe!)
814 funn :: Duration State -> Bool
815 funn (Duration (i, x)) = ((< 17) i) && (and (map x players))
816
817 {-- Is it possible for all adventurers to be on the other side in < 17 min ? --}
818 l17 :: Bool
819 l17 = let r = remLD (exec 17 gInit)
820       in or (map (funn) r)
821
822 -----
823 {-- Optional Task 2 --}
824 -----
825 -- Another way to see if "it is possible for all adventurers to be on the other side in < 17 min"
826 -- is to star with n=1 and see if there are at least one 'State' with time <17 and all the
827 -- 'Adventurers' are in the safe state, if False we increase n to n+1 until we can conclude if
828 -- it is possible for all adventurers to be on the other side in < 17 min.
829
830 -- Gets a condition, p.e (<17), and a 'Duration State' and checks if the time satisfies the
831 -- condition 'f' and all adventures are in the safe state.
832 fun2'' :: (Int -> Bool) -> Duration State -> Bool
833 fun2'' f (Duration (i, x)) = (f i) && (and (map x players))
834
835 -- Gets a n, number of execution, and a condition for the time of a 'Duration State' and checks
836 -- if for all the possible moves in n executions, there are at least one 'Duration State' with
837 -- time that respects the condition 'f' and all the 'Adventurers' in the safe state.
838 test :: Int -> (Int -> Bool) -> Bool
839 test n f = let r = remLD (exec n gInit)
840           in or (map (fun2'' f) r)
841
842 -----
843 -- Example:
844 -- > test 1 (<17)      > test 1 (<=17)      > test 1 (<19)
845 -- False              False              False
846 -- > test 2 (<17)      > test 2 (<=17)      > test 2 (<19)
847 -- False              False              False
848 -- > test 3 (<17)      > test 3 (<=17)      > test 3 (<19)
849 -- False              False              False
850 -- > test 4 (<17)      > test 4 (<=17)      > test 4 (<19)
851 -- False              False              False
852 -- > test 5 (<17)      > test 5 (<=17)      > test 5 (<17)
853 -- False              True               True
854 -- > test 6 (<17)      > test 6 (<=17)      > test 6 (<17)
855 -- False              True               True
856
857 -----
858 {-- Implementation of the monad used for the problem of the adventurers. Recall the Knight's
859 quest --}
860 data ListDur a = LD [Duration a] deriving Show
861
862 -- Transforms a "ListDur a" to a list of "Duration a"
863 remLD :: ListDur a -> [Duration a]
864 remLD (LD x) = x
865
866 -- Functor ListDur
867 instance Functor ListDur where
868   fmap f = let r = \(Duration (i, a)) -> Duration (i, f a)
869           in LD . ((map r).remLD)
870
871 -- Applicative ListDur

```

```
872 instance Applicative ListDur where
873   pure x = LD [Duration (0, x)]
874   l1 <*> l2 = LD $ do r1 <- remLD l1
875                     r2 <- remLD l2
876                     m (r1, r2) where
877                       m (Duration (i1, f), Duration(i2, a))
878                         = return (Duration(i1 + i2, f a))
879
880 -- Monad ListDur
881 instance Monad ListDur where
882   return = pure
883   l >=> k = LD $ do r <- remLD l
884                   m r where
885                     m (Duration (il, x)) =
886                       let v = remLD (k x)
887                       in map (\(Duration (im, x)) -> Duration (il + im, x)) v
888
889 manyChoice :: [ListDur a] -> ListDur a
890 manyChoice = LD . concat . (map remLD)
891
892 -----
893 {-- Optional Task 3 v2 --}
894 -----
895 -- New type of data, 'Duration2', same as 'Duration' but this one has a list of the final and
896 -- previous duration states, [Duration State].
897 data Duration2 a = Duration2 (Int, a, [Duration State]) deriving Show
898
899 getDuration2 :: Duration2 a -> Int
900 getDuration2 (Duration2 (d,_,_)) = d
901
902 getValue2 :: Duration2 a -> a
903 getValue2 (Duration2 (_,x,_)) = x
904
905 getStates2 :: Duration2 a -> [Duration State]
906 getStates2 (Duration2 (_,_,l)) = l
907
908 -- Functor Duration2
909 instance Functor Duration2 where
910   fmap f (Duration2 (i,x,l)) = Duration2 (i, f x, l)
911
912 -- Applicative Duration2
913 instance Applicative Duration2 where
914   pure x = (Duration2 (0,x,[]))
915   (Duration2 (i,f,l)) <*> (Duration2 (j,x,m)) = (Duration2 (i+j, f x, m))
916
917 -- Monad Duration2
918 instance Monad Duration2 where
919   (Duration2 (i,x,l)) >=> k =
920     Duration2 (i + (getDuration2 (k x)), getValue2 (k x), l)
921   return = pure
922
923 -----
924 -- New type of data, 'ListDur2', same as 'ListDur' but this one has a list 'Duration2'.
925 data ListDur2 a = LD2 [Duration2 a] deriving Show
926
927 -- Transforms a "ListDur2 a" to a list of "Duration2 a"
928 remLD2 :: ListDur2 a -> [Duration2 a]
929 remLD2 (LD2 x) = x
930
931 -- Functor ListDur2
932 instance Functor ListDur2 where
```

```
933   fmap f = let r = \(Duration2 (i, a, l)) -> Duration2 (i, f a, l)
934             in LD2 . ((map r).remLD2)
935
936 -- Applicative ListDur2
937 instance Applicative ListDur2 where
938   pure x = LD2 [Duration2 (0, x, [])]
939   l1 <*> l2 = LD2 $ do r1 <- remLD2 l1
940                       r2 <- remLD2 l2
941                       m (r1, r2) where
942                         m (Duration2 (i1, f, l), Duration2(i2, a, m))
943                           = return (Duration2(i1 + i2, f a, m))
944
945 -- Monad ListDur2
946 instance Monad ListDur2 where
947   return = pure
948   l >>= k = LD2 $ do r <- remLD2 l
949                     m r where
950                       m (Duration2 (il, x', l)) =
951                         let v = remLD2 (k x')
952                         in map (\(Duration2 (im, x, n')) -> Duration2 (il + im, x, l++n')) v
953
954 -----
955 -- Changes the 'State' of the 'Objects' inside a list and gives a 'Duration2 State' with the
956 -- time needed to perform the move, the resulting 'State' and puts the state in the list of
957 -- states .
958 fun4 :: State -> [Objetc] -> Duration2 State
959 fun4 s [] = Duration2 (0, s, [])
960 fun4 s (h:t) = let s' = mChangeState ((h:t)++[(Right ())]) s
961                i = getTimeAdv(\(Left a) -> a) h
962                in Duration2 (i, s', [Duration (i,s)])
963
964 allValidPlays2 :: State -> ListDur2 State
965 allValidPlays2 s = LD2 (map (fun4 s) (moves s))
966
967 -----
968 -- Receives the number n (number of individual moves), a function 'allValidPlays' and a 'State',
969 -- and returns a 'List Duration' with all the moves that the adventures can make.
970 propagate2 :: Int -> (State -> ListDur2 State) -> State -> ListDur2 State
971 propagate2 0 _ s = LD2 [Duration2 (0, s, [Duration (0, s)])]
972 propagate2 n s_l s = do
973   r <- (s_l s)
974   propagate2 (n-1) s_l r
975
976 {-- For a given number n and initial state, the function calculates all possible n-sequences
977 of moves that the adventures can make and the path to get there. --}
978 exec2 :: Int -> State -> ListDur2 State
979 exec2 n s = propagate2 n (allValidPlays2) s
980
981 -----
982 {-- Optional Task 4 --}
983
984 {-- Test functions --}
985 -- Gets a conditon, f, to be applied to the time of the 'Duration2 State', checks if the
986 -- condition is True and if all the 'Adventurers' are in the safe state, returns True or
987 -- False if both conditions are satisfied .
988 fun'2 :: (Int -> Bool) -> Duration2 State -> Bool
989 fun'2 f (Duration2 (i, x, l)) = (f i) && (and (map x players))
990
991 -- Gets a n, number of execution, and a condition for the time of a 'Duration State' and checks
992 -- if for all the possible moves in n executions there are at least one 'Duration State' with
993 -- time that respects the contition 'f' and all the 'Adventurers' in the safe state .
```

```

994 -- Returns True or False .
995 test1 :: Int -> (Int -> Bool) -> Bool
996 test1 n f = let r = remLD2 (exec2 n gInit)
997             in or (map (fun'2 f) r)
998
999 -- Gets a n, number of execution , and a condition for the time of a 'Duration State' and checks
1000 -- if for all the possible moves in n executions there are at least one 'Duration State' with
1001 -- time that respects the condition 'f' and all the 'Adventurers' in the safe state .
1002 -- Returns (tt or ff, n), where 'n' is the number of different solutions to get to a final
1003 -- 'Duration State' that satisfies the previous conditions .
1004 test2 :: Int -> (Int -> Bool) -> (Bool, Int)
1005 test2 n f = let r = remLD2 (exec2 n gInit)
1006             l = (filter (fun'2 f) r)
1007             in (length(l)/=0, length(l))
1008
1009 -- Gets a n, number of execution , and a condition for the time of a 'Duration State' and checks
1010 -- if for all the possible moves in n executions there are at least one 'Duration State' with
1011 -- time that respects the condition 'f' and all the 'Adventurers' in the safe state .
1012 -- Returns (tt or ff, n, l), where 'n' is the number of different solutions to get to a final
1013 -- 'Duration State' that satisfies the previous conditions , and 'l' is a list of 'Duration2 State'
1014 -- that that satisfies the previous conditions .
1015 test3 :: Int -> (Int -> Bool) -> (Bool, Int, [Duration2 State])
1016 test3 n f = let r = remLD2 (exec2 n gInit)
1017             l = (filter (fun'2 f) r)
1018             in (length(l)/=0, length(l), l)
1019
1020 -----
1021 -- Example:
1022
1023 -- > test1 5 (<=17)
1024 -- True
1025
1026 -- > test2 5 (<=17)
1027 -- (True,2)
1028
1029 -- > test3 5 (<=17)
1030 -- (True,2,
1031 -- (True,2,
1032 -- [Duration2 (17,[ " True "," True "," True "," True "," True "],
1033 -- [Duration (2,[ " False "," False "," False "," False "," False "])
1034 -- ,Duration (1,[ " True "," True "," False "," False "," True "])
1035 -- ,Duration (10,[ " False "," True "," False "," False "," False "])
1036 -- ,Duration (2,[ " False "," True "," True "," True "," True "])
1037 -- ,Duration (2,[ " False "," False "," True "," True "," False "])
1038 -- ,Duration (0,[ " True "," True "," True "," True "," True "]) ] ) ,
1039 -- Duration2 (17,[ " True "," True "," True "," True "," True "],
1040 -- [Duration (2,[ " False "," False "," False "," False "," False "])
1041 -- ,Duration (2,[ " True "," True "," False "," False "," True "])
1042 -- ,Duration (10,[ " True "," False "," False "," False "," False "])
1043 -- ,Duration (1,[ " True "," False "," True "," True "," True "])
1044 -- ,Duration (2,[ " False "," False "," True "," True "," False "])
1045 -- ,Duration (0,[ " True "," True "," True "," True "," True "]) ] ) ]

```

Listing 15: Adventurers2.hs

```

1046 module DurationMonad where
1047
1048 -- Defining a monad (the duration monad) --
1049 data Duration a = Duration (Int, a) deriving Show
1050
1051 getDuration :: Duration a -> Int

```

```
1052 getDuration (Duration (d,_)) = d
1053
1054 getValue :: Duration a -> a
1055 getValue (Duration (_,x)) = x
1056
1057 -- Functor Duration
1058 instance Functor Duration where
1059     fmap f (Duration (i,x)) = Duration (i,f x)
1060
1061 -- Applicative Duration
1062 instance Applicative Duration where
1063     pure x = (Duration (0,x))
1064     (Duration (i,f)) <*> (Duration (j, x)) = (Duration (i+j, f x))
1065
1066 -- Monad Duration
1067 instance Monad Duration where
1068     (Duration (i,x)) >=> k = Duration (i + (getDuration (k x)), getValue (k x))
1069     return = pure
1070
1071 wait1 :: Duration a -> Duration a
1072 wait1 (Duration (d,x)) = Duration (d+1,x)
1073
1074 wait :: Int -> Duration a -> Duration a
1075 wait i (Duration (d,x)) = Duration (i + d, x)
```

Listing 16: DurationMonad.hs

10 Referências

<https://wiki.haskell.org/Monad>
<https://www.youtube.com/watch?v=xCut-QT2cpI&ab>
<https://www.youtube.com/watch?v=CNOff5LPKQI&ab>
<https://www.youtube.com/watch?v=f1Y7vLakykk&ab>
<http://learnyouahaskell.com/chapters>