



UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

MESTRADO EM ENGENHARIA INFORMÁTICA

MÉTODOS FORMAIS DE PROGRAMAÇÃO

Programação Cíber-Física  
Trabalho Prático II

Ariana Lousada (PG47034) Ana Ribeiro (PG47841)

20 de junho de 2022

### **Resumo**

O presente documento descreve todo o raciocínio durante o desenvolvimento do segundo trabalho prático da unidade curricular Programação Cíber-Física inserida no perfil Métodos Formais de Programação do Mestrado de Engenharia Informática da Universidade do Minho.

O trabalho prático encontra-se dividido em duas tarefas: (1) Construção de um Monad e respetiva aplicação em Haskell para o *Adventure's Problem* e a respetiva (2) Discussão e comparação com o modelo do mesmo problema construído em UPPAAL durante o período letivo da unidade curricular.

# 1 Primeira Tarefa

O objetivo da primeira tarefa do presente trabalho prático consiste na construção e implementação do *monad ListDur* em Haskell para um problema já abordado anteriormente na disciplina: *The Adventure's Problem*.

Pretende-se testar várias soluções com a construção do *monad*. Para o seu desenvolvimento, a equipa de trabalho analisou em detalhe alguns monads abordados na unidade curricular, como o `DurationMonad` e o `LogList`.

Com isto, começou-se por definir o *functor*, no qual se aplicou um raciocínio semelhante ao `DurationMonad`, definindo o `fmap` para aplicar uma dada função a uma lista de `Durations`:

```
instance Functor ListDur where
    fmap f = let f' = \(Duration (i,x)) -> Duration (i,f x) in
              LD . (map f') . remLD
```

De seguida, definiu-se o *Applicative Functor*, no qual se aplicou um raciocínio semelhante ao `DurationMonad`, utilizando também parte da implementação do `LogLists` uma vez que neste contexto é necessário percorrer listas de `Durations`:

```
instance Applicative ListDur where
    pure x = LD [Duration (0,x)]
    l1 <*> l2 = LD $ do x <- remLD l1
                       y <- remLD l2
                       g(x,y) where
                           g(Duration (t1, w), Duration (t2, z)) =
                               return $ Duration (t1+t2, w z)
```

Por fim, definiu-se o *Monad* também recorrendo a uma lógica semelhante ao `LogLists` construído para o problema do *Knight's quest*:

```
instance Monad ListDur where
    return = pure
    l >>= k = LD $ do x <- remLD l
                      g x where
                          g(Duration (s,x)) = let u = (remLD (k x)) in
                                                  map \(Duration (s',x)) -> Duration (s + s', x)) u
```

Com o monad inteiramente definido, é então necessário definir a sua implementação para o problema em análise.

Primeiro começamos por definir a função `getTimeAdv` que retorna o tempo(em minutos) que cada `Adventurer` demora a atravessar a ponte.

```
getTimeAdv :: Adventurer -> Int
getTimeAdv P1 = 1
getTimeAdv P2 = 2
getTimeAdv P5 = 5
getTimeAdv P10 = 10
```

De seguida definimos a função `allValidPlays` que retorna todos os movimentos que são possíveis fazer, dado um estado. Para esta implementação são usadas algumas funções definidas previamente.

```

allValidPlays :: State -> ListDur State
allValidPlays s = LD [
  if s (Right()) == s (Left P1)
  then Duration(getTimeAdv P1, mChangeState [(Right()), (Left P1)] s)
  else Duration(0, s),
  if s (Right()) == s (Left P2)
  then Duration(getTimeAdv P2, mChangeState [(Right()), (Left P2)] s)
  else Duration(0, s),
  if s (Right()) == s (Left P5)
  then Duration(getTimeAdv P5, mChangeState [(Right()), (Left P5)] s)
  else Duration(0, s),
  if s (Right()) == s (Left P10)
  then Duration(getTimeAdv P10, mChangeState [(Right()), (Left P10)] s)
  else Duration(0, s),
  if (s (Right()) == s (Left P1)) && (s (Right()) == s (Left P2))
  then Duration(getTimeAdv P2, mChangeState [(Right()), (Left P1), (Left P2)] s)
  else Duration(0, s),
  if (s (Right()) == s (Left P1)) && (s (Right()) == s (Left P5))
  then Duration(getTimeAdv P5, mChangeState [(Right()), (Left P1), (Left P5)] s)
  else Duration(0, s),
  if (s (Right()) == s (Left P1)) && (s (Right()) == s (Left P10))
  then Duration(getTimeAdv P10, mChangeState [(Right()), (Left P1), (Left P10)] s)
  else Duration(0, s),
  if (s (Right()) == s (Left P2)) && (s (Right()) == s (Left P5))
  then Duration(getTimeAdv P5, mChangeState [(Right()), (Left P2), (Left P5)] s)
  else Duration(0, s),
  if (s (Right()) == s (Left P2)) && (s (Right()) == s (Left P10))
  then Duration(getTimeAdv P10, mChangeState [(Right()), (Left P2), (Left P10)] s)
  else Duration(0, s),
  if (s (Right()) == s (Left P5)) && (s (Right()) == s (Left P10))
  then Duration(getTimeAdv P10, mChangeState [(Right()), (Left P5), (Left P10)] s)
  else Duration(0, s)]

```

Por fim definimos uma função `exec` que executa a função anterior o número de vezes definidas. Esta função irá retornar uma `ListDur` com todos os estados que são alcançáveis.

```

exec :: Int -> State -> ListDur State
exec 0 s = LD [return $ s]
exec n s = do s0 <- allValidPlays s
            exec (n-1) s0

```

Após a definição das funções de implementação, passamos à definição das funções de verificação. Apesar de apenas haver restrição de número de movimentos em uma das funções, a equipa de trabalho decidiu utilizar 5 iterações para ambos os casos devido ao tempo de execução destas.

- Conseguirão todos os *Adventurers* atravessar a ponte em 17 minutos ou menos? Em 5 ou menos movimentos?

```

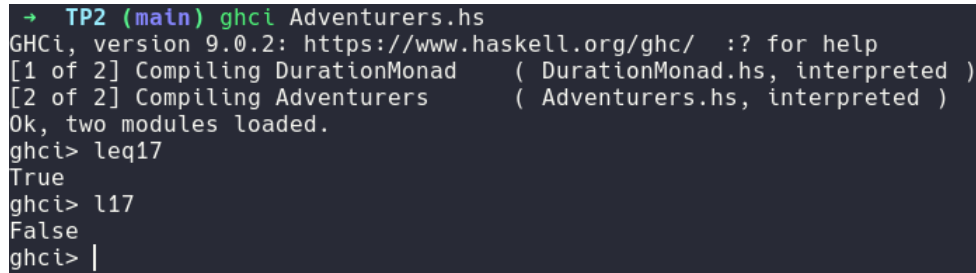
leq17 :: Bool
leq17 = any (\x -> getDuration(x) <= 17
            && getValue(x) == const True) (remLD (exec 5 gInit))

```

- Conseguirão todos os *Adventurers* atravessar a ponte em menos de 17 minutos?

```
l17 :: Bool
l17 = any (\x -> getDuration(x) < 17
           && getValue(x) == const True) (remLD (exec 5 gInit))
```

Testando as funções de verificação anteriormente definidas com o `ghci` obtêm-se os resultados já esperados:



```
→ TP2 (main) ghci Adventurers.hs
GHCi, version 9.0.2: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling DurationMonad    ( DurationMonad.hs, interpreted )
[2 of 2] Compiling Adventurers      ( Adventurers.hs, interpreted )
Ok, two modules loaded.
ghci> leq17
True
ghci> l17
False
ghci> |
```

Figura 1: Resultados obtidos recorrendo ao `ghci`

- É possível todos os aventureiros atravessarem a ponte em 17 minutos e no máximo em 5 passos.
- Não é possível todos os aventureiros atravessarem a ponte em menos de 17 minutos.

## 2 Segunda Tarefa

Para a segunda tarefa do presente trabalho prático, pretende-se discutir e comparar os modelos construídos para o mesmo problema em UPPAAL e em Haskell.

Enquanto que na ferramenta UPPAAL é possível observar mais intuitivamente os vários possíveis estados do sistema, esta apresenta algumas limitações:

- O próprio editor da ferramenta é simples. Não permite a definição de funções muito complexas.
- Não se podem aplicar alguns *data types* em certos contextos (por exemplo: valores do tipo *double* não podem ser utilizados em estruturas).
- Em termos de *model checking*, cada localização convém estar sempre ligada a um determinado valor temporal - o modelo não pode estar numa localização por tempo indefinido.
- Uma *reachability query* na ferramenta não testa extensivamente todos os estados possíveis, parando logo na primeira vez que a *query* seja verdadeira.
- Uma *liveness query*, uma vez que consiste numa implicação ( $p \rightarrow q$ ) define que se 'p' é alcançável, eventualmente 'q' também vai ser. Contudo, podem existir cenários no qual 'q' é alcançável sem 'p'.

A maioria destas limitações da ferramenta UPPAAL são facilmente contornáveis em Haskell, principalmente quando se utilizam Monads. Uma vez que é possível definir um Monad em específico para o problema em questão, é possível definir todas as regras, cláusulas, lógica e cálculos necessários para explorar o contexto do problema.

Com isto em conta, a principal desvantagem da utilização de Monads face à utilização da ferramenta UPPAAL será a dificuldade na sua interpretação e construção. Enquanto que na ferramenta a sua utilização é mais intuitiva, a construção de Monad exige já outro tipo de estudo e pesquisa por parte do implementador.

Após a construção e teste do monad `ListDur`, observou-se que é possível obter estados e resultados do problema mais rapidamente em comparação com a ferramenta UPPAAL. Isto deve-se ao Monad utilizar lógica e cálculo automaticamente sem intervenção do utilizador. Enquanto isto, no UPPAAL

é necessário explorar cada cenário em particular, assim como definir cláusulas para cada requisito do sistema, testando cada uma individualmente.

Em suma, conclui-se que, recorrendo a Monads, obtém-se uma solução mais eficiente.

### **3 Referências**

- (1) A Gentle Introduction to Haskell, Version 98 : Monads
- (2) Applicative Functors : Haskell
- (3) Custom Monad in Haskell : Michael Gilliland
- (4) Current limitations of the Uppaal tool