Escola de Engenharia
Universidade do Minho

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado Integrado em Engenharia Informática
Arquitetura e Cálculo

Trabalho Prático

TP1

Manuel Monteiro - A74036

Tiago Baptista - A75328

Braga, 16 de Junho de 2020

Conteúdo

Lista de excertos de código

1. Introdução

O trabalho descrito no presente documento surge no âmbito da unidade curricular de Arquitetura e Cálculo pertencente ao perfil de Métodos Formais em Engenharia de Software. Este relatório visa expor o problema bem como a abordagem adoptada na resolução do mesmo.

O problema proposto no enunciado refere quatro aventureiros que pretendem passar uma ponte. No entanto, a ponto só aguenta com o peso de dois aventureiros de cada vez. Além disso, é necessário que os aventureiros tenham na sua posse ao longo de toda a passagem. Deve ainda ser acrescentado que os aventureiro não têm a mesma capacidade para atravessarem a ponte sendo que o mais rápido demora 1 minuto e o mais lento 10. Os restantes demoram 2 e 5 minutos. Foi elaborado um sistema de mônadas em Haskell que visa resolver o problema e responder às questões no enunciado.

2. Estruturação de Código

2.1 Definição de Mônadas

Inicialmente, o sistema utiliza um mônada de duração cuja definição se encontra ilustrada no excerto 1, que consegue indicar o tempo total desde o início da resolução do problema.

```
1 data Duration a = Duration (Int, a) deriving Show
```

1: *Definição do mônada de duração.*

Existem também os diferentes aventureiros como um tipo de dados. Utilizaram-se os tipos de dados para definir objectos que são um aventureiro ou uma lanterna. A definição destes objetos encontra-se clarificada no excerto 2.

```
1 data Adventurer = P1 | P2 | P5 | P10 deriving (Show,Eq)
2
3 type Objects = Either Adventurer ()
```

2: *Definição de objetos (Aventureiros + Lanterna).*

Além disso foi criado um tipo estado mais as suas instâncias. Primeiro foi criada a instância **Show** para que fosse possível a visualização do tipo de dados na consola. De seguida criou-se também a instância **Eq** de maneira a que fosse possível comparar os estados de forma direta.

Estas instâncias permitem identificar o lado em que se encontram os aventureiros e a lanterna. O valor **TRUE** representa o lado final da ponte. Já o valor **FALSE** representa o lado inicial da ponte.

O código referente à criação de ambas as instância está apresentado no excerto 3.

```haskell
type State = Objects -> Bool

instance Show State where
  show s = (show . (fmap show)) [s (Left P1),
                                 s (Left P2),
                                 s (Left P5),
                                 s (Left P10),
                                 s (Right ())]

instance Eq State where
  (==) s1 s2 = and [s1 (Left P1) == s2 (Left P1),
                    s1 (Left P2) == s2 (Left P2),
                    s1 (Left P5) == s2 (Left P5),
                    s1 (Left P10) == s2 (Left P10),
                    s1 (Right ()) == s2 (Right ())]
```

3: *Definição do tipo estado.*

Foi também utilizado outro mônada que engloba uma lista de durações, representada no excerto 4, sendo que este vai consistir na lista que contém as soluções e representações dos movimentos válidos para o problema.

```haskell
data ListDur a = LD [Duration a] deriving Show
```

4: *Definição do mônada de lista de durações.*

Em seguida, definiu-se o funtor da lista, 5, para que assim fosse possível a aplicação de funções neste tipo de dados:

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

5: *Funtor do mônada de lista de durações.*

Depois, foi também definido o aplicativo da mônada que aplica uma função embrulhada no contexto a um valor embrulhado nesse mesmo contexto:

```haskell
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(d,f),Duration(d',x)) =
                           ↪  return (Duration(d+d',f x))
```

***6:** Aplicativo do mônada de lista de durações.*

Por fim, foi definido o mônada, ilustrado no excerto 7 que aplica uma função para devolver valores embrulhados a partir de um valor:

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

***7:** Mônada da lista de durações.*

2.2 Mudanças de Estado

Para a resolução do problema é necessária a mudança do estado dos aventureiros bem como da lanterna. Para isso é utilizada uma função, representada no excerto 8, que muda o estado do objeto pretendido (Aventureiro ou lanterna):

```haskell
changeState :: Objects -> State -> State
changeState a s = let v = s a in (\x -> if x == a then not v
 ↪  else s x)
```

***8:** Definição da função de mudança de estado de um objeto.*

A função **changeState** altera o estado de 1 objeto. Para que fosse possível aplicar a alteração do estado a mais do que um objeto foi criada a função **mChangeState**, que recorre à função de ordem superior **foldr**, de maneira a que a função **changeState** seja aplicada a todos os elementos da lista de objetos. A função **mChangeState** está representada no excerto 9.

```haskell
mChangeState :: [Objects] -> State -> State
mChangeState os s = foldr changeState s os
```

9: *Definição da função de mudança de estado de vários objetos.*

Desta forma, quando um aventureiro pegar na lanterna e atravessar a ponte, consegue-se mudar os estados desses dois objetos (lanterna e aventureiro).

2.3 Definição de Movimentos Válidos

Para definir os movimentos válidos a partir de um certo estado, considerou-se necessário verificar quais dos aventureiros se encontram do mesmo lado da lanterna para assim pegar nesta e atravessar a ponte. Para isso, foi criada uma função, representada no excerto 10, que a partir de uma lista de aventureiros e de um estado, devolve aqueles que estão no mesmo lado da lanterna:

```haskell
validAdv :: [Adventurer] -> State -> [Adventurer]
validAdv [] _ = []
validAdv (h:t) s = if s (Left h) == s (Right ()) then h :
 ↪  validAdv t s else validAdv t s
```

10: *Definição da função que verifica os aventureiros que podem atravessar.*

Depois de estarem identificados os aventureiros que podem atravessar a ponte,agruparam-se os mesmos 2 a 2 ou individualmente, pois são as únicas forma de se poder efetuar um movimento. Para tal foi elaborada uma função de combinação, ilustrada no excerto 11, que recebe um inteiro (número de elementos que se pretende ter na combinação) e uma lista de aventureiros (valores a serem combinados).

```haskell
combine :: Int -> [Adventurer] -> [[Adventurer]]
combine 0 _ = [[]]
combine _ [] = []
combine n (x:xs) = map (x :) (combine (n-1) xs) ++ combine n
 ↪  xs
```

11: *Definição da função que agrupa aventureiros.*

Com os aventureiros no mesmo lado da lanterna combinados dois a dois ou individualmente, pode-se então a partir de um estado criar uma lista de mônadas de duração que indicam o tempo de travessia e o estado resultante desse movimento. Para que fosse possível a criação do mônada de duração, foi necessário a criar uma função com recurso à função *wait* para indicar o tempo de travessia de cada um dos aventureiros.

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

12: *Definição da função que indica o tempo de travessia de cada um dos aventureiros.*

```haskell
wait :: Int -> Duration a -> Duration a
wait i (Duration (d,x)) = Duration (i + d, x)
```

13: *Definição da função que aplica o tempo de espera no mônada de duração.*

Deve ainda ser destacado que quando se agrupam dois aventureiros o tempo de travessia dos mesmos, será o maior valor entre os dois.

```haskell
dList :: State -> [[Adventurer]] -> [Duration State]
dList _ [] = []
dList s (x:xs)
   | length x == 1 =
      [wait (getTimeAdv(x !! 0)) $ return $ mChangeState [Left
       ↪  (x !! 0), Right ()] s] ++ dList s xs
   | length x == 2 =
      [wait (max (getTimeAdv (x !! 0)) (getTimeAdv(x !! 1))) $
       ↪  return $ mChangeState [Left (x !! 0), Left (x !! 1),
       ↪  Right ()] s] ++ dList s xs
```

14: *Definição da função que devolve uma lista de mônadas de durações com movimentos válidos.*

Por fim, pegou-se na lista de durações e embrulhou-se a mesma na mônada referida anteriormente obtendo assim os movimentos válidos a partir de um estado. Note-se que foram agrupados os aventureiros válidos de atravessar a ponte 2 a 2 ou sozinhos como explicado anteriormente.

```haskell
allValidPlays :: State -> ListDur State
allValidPlays s = let x = validAdv [P1,P2,P5,P10] s in
    LD (dList s (combine 2 x ++ combine 1 x))
```

15: *Definição da função que devolve o mônada de lista de durações com movimentos válidas.*

Após se conter os movimentos válidos por estado, foi criada uma função que a partir de um estado executa n movimentos a partir do mesmo:

```haskell
exec :: Int -> State -> ListDur State
exec 0 s = return s
exec n s = do s1 <- allValidPlays s
              exec (n-1) s1
```

16: *Definição da função que executa n movimentos a partir de um estado.*

3. Verificação de resultados

Após executar n vezes os movimentos válidos a partir de um estado, é criada uma lista de durações com todas as diferentes possibilidades. Posteriormente,é então utilizado um estado inicial, com todos os objetos do lado inicial da ponte:

```
gInit :: State
gInit = const False
```

17: *Definição do estado inicial.*

Para verificar se é possível atravessar em menos de 17 unidades temporais 5 movimentos basta executar 5 vezes a função de movimentos válidos e verificar se existe algum elemento nessa lista em que a duração seja $<= 17$ e que todos os objetos no estado estejam no lado final (valores todos verdadeiros):

```
leq17 :: Bool
leq17 = length x > 0
        where y = map (\x -> (getDuration x, getValue x)) $
         ↪  remLD (exec 5 gInit)
              x = filter (\x -> fst x <= 17 && snd x == const
               ↪  True) y
```

18: *Definição da propriedade que verifica soluções em 5 movimentos com um tempo* $<= 17$ *unidades temporais.*

Pode também verificar-se se existem soluções que sejam mais rápidas do que 17 unidades temporais:

```
l17 :: Bool
l17 = length x > 0
        where y = map (\x -> (getDuration x, getValue x)) $
         ↪  remLD (exec 5 gInit)
              x = filter (\x -> fst x < 17 && snd x == const
               ↪  True) y
```

19: *Definição da propriedade que verifica soluções em 5 movimentos com um tempo* < 17 *unidades temporais.*

Executando estas duas propriedades verifica-se se existem ou não soluções para as mesmas:

```
*Adventurers> leq17
True
*Adventurers> l17
False
```

20: *Verificação das duas propriedades.*

4. Conclusão

4.1 UPPAL vs Mônadas

Com os dois trabalhos práticos entrou-se em contacto com duas abordagens para resolver o mesmo problema que consiste na modelação de um sistema em tempo real.

Começando pelo *UPPAL*, trata-se de uma ferramenta desenhada especificamente para a modelação de sistemas concorrentes temporais e recorre ao uso de automatos temporais, sendo que permite a verificação de propriedades usando formulas lógicas. Por outro lado, o processo de modelação utilizando mônadas está assente na linguagem funcional *Haskell*. As mônadas permitem a extensão de uma linguagem que de outra maneira seria pura (sem efeitos) para uma linguagem que lida com efeitos, ou seja, recebe um input e gera um output e efeitos. Esta extensão é essencial para o caso pois no mundo real existem muitas poucas operações puras (sem efeitos).

Como principais vantagens do *UPAAL* destacam-se o facto do *UPPAL* ter sido construído para o efeito de modulação de sistemas temporais e de restringir, à partida, a maneira de modular, dando um ambiente mais controlado. A existência dos *clocks* e o facto do seu funcionamento já estar predefinido à partida torna o processo mais simples, assim como a facilidade de controlo de transições entre estados dados pelas guardas e *updates*. Por outro lado, a definição de sistemas maiores e mais complexos pode-se tornar bastante confusa com o recurso a vários *templates* e diferentes ações sincronizantes, que podem ser difíceis de gerir. Isto também requer outro tipo de pensamento para os problemas, pois nestes sistemas podem haver adversidades tais como *deadlocks*, *timelocks*, *zeno paths* mais difíceis de detetar, tornando mais penosa a modelação de sistemas desta forma.

Como principais vantagens da programação monádica destacam-se que a interligação de diferente mônadas permite a modelação de várias componentes do sistema, o facto de existir mais controlo sobre todo o processo de modelação e ainda o fator de o conceito de mônada capturar dois problemas num só, a concorrência e a verificação de programas. Para além disso, o facto de se estar a usar uma linguagem de programação pode tornar o processo de modelação numa aproximação a uma possível implementação. Por outro lado, o *Haskell* obriga a definições mais complexas ao nível de código (de funtores, estruturas de dados e do funcionamento do sistema em geral) e comparativamente com o UPAAL não existe uma abordagem genérica, como a criação de templates, apesar de poderem existir paralelismos na estrutura da resolução dos problemas como constatamos na resolução deste trabalho prático, entre o problema dos *adventurers* e dos *knigth*.