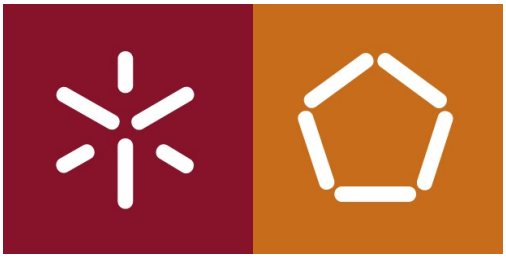

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Arquitetura e Cálculo - 2019/20

Modelação e análise de um sistema de tempo real com Monads

Carla Cruz a80564

Jorge Cruz a78895

16 de Junho de 2020

1 Introdução

No âmbito da unidade curricular de *Arquitetura e Cálculo*, foi desenvolvido um modelo do sistema, bem como verificação e teste, com base no problema proposto. Para alcançar os objetivos propostos, utilizamos a linguagem **Haskell** e os conhecimentos adquiridos sobre *Monads*, em particular, Listas e Durações com *Monads*.

Iremos apresentar, por fim, uma comparação sobre as abordagens em **UPPAAL** e **Haskell**, referindo os pontos positivos e negativos de cada uma e as principais dificuldades.

Neste sentido, iremos apresentar toda a linha de pensamento e estratégias adotadas para a concretização dos objetivos, desde a descrição do problema apresentado no enunciado, passando pela apresentação da forma como implementamos a solução e, por fim, as conclusões obtidas.

2 Descrição do problema

O contexto do problema consiste num grupo de quatro aventureiros que, durante a noite, encontraram uma ponte frágil sob um precipício profundo que necessitam de atravessar. Por razões de segurança, decidiram que não devem atravessar a ponte mais do que 2 pessoas ao mesmo tempo e que um deles deve transportar a lanterna para iluminar o caminho, sendo que apenas existe uma lanterna com eles.

Apresentando habilidades diferentes, os mesmos demoram tempos diferentes a percorrer a ponte, demorando **1, 2, 5 e 10 minutos**, respectivamente. Cada par de aventureiros que atravesse a ponte simultaneamente demora o período de tempo igual ao do mais lento dos dois aventureiros.

Os aventureiros têm opiniões diferentes relativamente ao tempo que demoram todos a atravessar esta ponte. Um afirma que eles não podem estar do outro lado em menos de 19 minutos. Outro discorda e afirma que isso pode ser feito em 17 minutos. Como nossa tarefa, iremos verificar através do **Haskell**, os seguintes aspetos:

- Modelar o sistema com base na situação usando o que aprendemos sobre Monads.
- Mostrar que é realmente possível que todos os aventureiros estejam do outro lado em 17 minutos.
- Mostrar que é impossível para todos os aventureiros estar do outro lado em menos de 17 minutos.

3 Modelo do Sistema

3.1 *Monad* de Listas de Durações

Para a modelação do nosso sistema, o ponto de partida consistiu em implementar o *Functor*, o *Applicative* e, por fim, a *Monad* correspondente às listas de durações. Para a definição destas, tivemos em conta o que já teria sido praticado e utilizado anteriormente no exemplo do *Knight's Quest*, pelo que, desta forma, foi possível a implementação da Monad para o problema dos aventureiros em específico.

A definição do *Functor* do tipo `ListDur` consistiu em aplicar a função recebida em cada `Duration` presente na lista, através do *fmap* (ou do *Functor*) definido para o tipo `Duration`.

```
instance Functor ListDur where
   fmap f = let f' = fmap f in
      LD . (map f') . remLD
```

Novamente, para a definição do *Applicative*, recorremos ao tipo `Duration` para nos ajudar. Ou seja, a função `pure` apenas devolve a lista com um único elemento devolvido pela função `pure` já definida no *Applicative* do tipo `Duration`. Da mesma maneira, a função `<*>` limita-se a pegar em cada elemento `x` da primeira lista e em cada elemento `y` da segunda lista, e devolve uma nova lista com todos os resultados de aplicar `x <*> y` que, por sua vez, recorre à função `<*>` definida no tipo `Duration`.

```
instance Applicative ListDur where
   pure x = LD [pure x]
   l1 <*> l2 = LD $ do x <- remLD l1
                       y <- remLD l2
                       g(x,y) where
                         g(x,y) = return $ x <*> y
```

Por fim, para a definição da *Monad*, a estratégia consistiu em: pegar em cada `Duration(a,b)`, aplicar a função `k` ao valor `b` (que devolve uma lista de durações) e, a cada elemento do resultado dessa operação, somar ao inteiro (correspondente ao tempo) o valor `a`. Todas as listas obtidas são agrupadas numa só, sendo este o resultado da operação `>>=`.

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

3.2 Funções auxiliares

Foi necessária a implementação de uma função que, dado um aventureiro, nos diz quanto tempo este demora a atravessar a ponte e, devido à forma como abordamos o problema, foi necessária também a implementação da função *getAdv* que, dado um valor do tipo `Objects`, nos retorna o aventureiro em específico.

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

getAdv :: Objects -> Adventurer
getAdv (Left a) = a
```

Posto isto, foi possível definir a função *time2cross* em que é possível concluir quanto tempo um grupo de aventureiros demora a percorrer a ponte, dado que este podem atravessá-la acompanhados. O resultado corresponderá sempre ao tempo que o aventureiro mais lento da lista toma a atravessar a ponte.

```
time2cross :: [Objects] -> Int
time2cross l = maximum $ map (getTimeAdv . getAdv) l
```

3.3 allValidPlays

Para dar resposta a uma implementação pedida pelo professor - *allValidPlays* - começamos por repartir o problema. Para começar, percebemos que é necessário saber quais os aventureiros que podem ultrapassar a ponte num dado estado. Desta forma, definimos a função *whoCanCross* que devolve os aventureiros que se encontram do mesmo lado da lanterna uma vez que, para a ponte ser percorrida, é necessário o seu transporte.

```
whoCanCross :: State -> [Objects]
whoCanCross s = filter (\x -> (s x) == (s (Right ())))
[Left P1, Left P2, Left P5, Left P10]
```

Tendo todos os aventureiros que reúnem as condições para ultrapassar a ponte, estes podem passar sozinhos ou acompanhados. Assim, é necessário criar todas as combinações possíveis quer este passe sozinho, quer este vá acompanhado, tendo em conta o grupo de aventureiros que pode passar.

Desta forma, é implementada a função *allChangeState* em que, dado um estado, esta apresenta todas as alterações que podem ser aplicadas ao estado em conjunto com o tempo que os aventureiros levam para atravessar a ponte nessa mudança. Por outras palavras, estamos a criar uma nova lista com todas as mudanças possíveis no estado recebido e a duração que estas demoram a acontecer.

```
allComb :: [a] -> [[a]]
allComb [] = []
allComb (x:xs) = [[x]] ++ map (\y -> [x,y]) xs ++ allComb xs

allChangeState :: State -> ListDur (State -> State)
allChangeState s = LD $ map (\l -> Duration(time2cross l, mChangeState
(l ++ [Right()]))) (allComb . whoCanCross $ s)
```

Com todas as definições apresentadas acima, foi possível, por fim, chegar ao principal objetivo: a implementação da função que, para um dado estado, retorna todos os movimentos possíveis que os aventureiros podem realizar - *allValidPlays*. Através da função `<*>` definida no *Applicative* do tipo `ListDur`, foi fácil aplicar todas as mudanças possíveis ao estado recebido e, assim, obter uma lista com todas as durações e estados possíveis a partir do estado recebido pela função *allValidPlays*.

```
allValidPlays :: State -> ListDur State
allValidPlays s = (allChangeState s) <*> (return s)
```

3.4 exec

De seguida, foi necessário realizar a implementação da função que, dado um valor `n` correspondente ao número de movimentos que se podem fazer e um dado estado inicial, calcula todos os possíveis estados em conjunto com o tempo decorrido para chegar a cada um. É nesta tarefa que realmente podemos observar o poder da *monad* das listas de durações. Aplicando a função *allValidPlays* a uma lista de durações de estados, através da função `>>=` definida para o tipo `ListDur`, estamos a criar uma lista com todos os estados possíveis seguintes e com o tempo decorrido para cada um. Ou seja, cada vez que aplicamos `>>= allValidPlays`, estamos a fazer uma nova iteração de jogadas possíveis. Por exemplo, fazer `return gInit >>= allValidPlays >>= allValidPlays`, corresponde a realizar todas as possíveis duas jogadas a partir do estado inicial.

```
exec :: Int -> State -> ListDur State
exec 0 s = return s
exec n s = let l = exec (n-1) s in
   l >>= allValidPlays
```

3.5 Verificação das propriedades

Para a verificação de propriedades, o desafio encontrou-se no modo como seriam aplicadas as várias jogadas possíveis. A primeira solução consistiu, obviamente, em chamar a função *exec* quantas iterações fossem necessárias ou permitidas até encontrar o estado desejado. Porém, para além de estar a repetir sucessivamente as mesmas operações desnecessariamente, o tempo para encontrar a solução escalava rapidamente à medida que novas iterações eram aplicadas (o que se verificava especialmente para a segunda propriedade). Assim, duas medidas importantes foram tomadas. Primeiro, para evitar chamadas sucessivas à função *exec*, optou-se por aplicar diretamente `>>= allValidPlays` em cada iteração e verificar se o estado desejado fora

atingido. Segundo, observou-se que alguns dos estados, que nunca atingiriam as condições desejadas, estavam a ser arrastados pelas iterações. Para isto, a solução consistiu em, sempre que `>>= allValidPlays` era aplicado, caso o estado desejado não fosse encontrado, apenas os estados com condições para atingir o desejado eram mantidos para a próxima iteração.

As únicas diferenças entre as funções *check* e *check2*, criadas para a verificação das propriedades, encontram-se no número de iterações permitidas e na condição imposta sobre o tempo. Na função *check*, existe um inteiro a acompanhar cada iteração que, cada vez que a função é chamada recursivamente, é decrementado para que, quando o seu valor chegar a 0, a função pare e devolva `False` como resultado pois todas as iterações permitidas foram executadas.

```
check :: Int -> Int -> ListDur State -> Bool
check 0 t l = False
check n t (LD []) = False
check n t l = let l' = remLD $ l >>= allValidPlays in
   if any (\x -> (getDuration x <= t) && (getValue x == gEnd)) l'
      then True
      else check (n-1) t $ LD $ filter (\x -> (getDuration x <= t)) l'

leq17 :: Bool
leq17 = check 5 17 $ return gInit

check2 :: Int -> ListDur State -> Bool
check2 t (LD []) = False
check2 t l = let l' = remLD $ l >>= allValidPlays in
   if any (\x -> (getDuration x < t) && (getValue x == gEnd)) l'
      then True
      else check2 t $ LD $ filter (\x -> (getDuration x < t)) l'

l17 :: Bool
l17 = check2 17 $ return gInit
```

Na figura seguinte, podemos observar o teste destas propriedades e, como seria de esperar, o resultado das mesmas. Ou seja, é possível todos os aventureiros chegarem ao outro lado da ponte em 17 minutos e sem exceder os 5 movimentos (`True` no resultado do `leq17`), mas é impossível acontecer o mesmo em menos de 17 minutos e sem quaisquer restrições no número de movimentos (`False` no resultado do `l17`).

```
(master)⚡ % ghci Adventurers.hs
GHCi, version 8.6.3: http://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.
*Adventurers> leq17
True
*Adventurers> l17
False
```

Figura 2: Teste das propriedades `leq17` e `l17`

4 Tarefa Opcional

Para este trabalho, foi dada a opção de realizar uma tarefa opcional à escolha que estendesse a modelação realizada para o problema proposto. A nossa opção consistiu em apresentar o conjunto de passos realizados para atingir o objetivo do quebra-cabeças, ou seja, mostrar que movimentos os aventureiros tomaram para chegarem todos ao outro lado da ponte em apenas 17 minutos. Para adicionar esta funcionalidade, a solução passou por transformar a *Monad* das listas de durações para uma *Monad* de listas de pares de *logs* e durações.

Ou seja, cada duração passa a ser acompanhada por uma `String`.

```
data LogListDur a = LLD [(String, Duration a)] deriving Show

instance Functor LogListDur where
   fmap f = let f' = \(s,x) -> (s, fmap f x) in
      LLD . (map f') . remLLD

instance Applicative LogListDur where
   pure x = LLD [([], pure x)]
   l1 <*> l2 = LLD $ do x <- remLLD l1
                        y <- remLLD l2
                        g(x,y) where
                           g((s1, d1),(s2, d2)) = return (s1 ++ s2, d1 <*> d2)

instance Monad LogListDur where
   return = pure
   l >>= k = LLD $ do x <- remLLD l
                      g x where
                        g((s,Duration(a,b))) = let u = (remLLD (k b)) in
                            map (\(s',x) -> (s ++ s', wait a x)) u
```

Sempre que uma mudança fosse aplicada num dado estado, foi necessário adicionar um *log* criado pela função *getMoveStr* correspondente ao movimento realizado pelo grupo de aventureiros que atravessa a ponte.

```
getMoveStr :: State -> [Objects] -> String
getMoveStr s l
   | s $ Right () = "<-" ++ (show $ map (\o -> getAdv o) l) ++ "\r\n"
   | otherwise    = "->" ++ (show $ map (\o -> getAdv o) l) ++ "\r\n"
```

De resto, para verificar a solução do problema, definiu-se a função *find* que, com a mesma estratégia das funções criadas para a verificação de propriedades, iterava sobre as jogadas possíveis até atingir o estado pretendido e devolvia o *log* da primeira solução encontrada.

```
find :: LogListDur State -> String
find l = let l' = remLLD $ l >>= lallValidPlays in
   if any (\(s,x) -> (getDuration x <= 17) && (getValue x == gEnd)) l'
      then getMovesLog $ head $ filter (\(s,x) -> (getDuration x <= 17)
                                                  && (getValue x == gEnd)) l'
      else find $ LLD $ filter (\(s,x) -> (getDuration x < 17)) l'

test :: IO ()
test = putStr $ find $ return gInit
```

Assim, para testar, basta apenas executar `test` e verificar os movimentos necessários da solução.

```
[(master)⚡ % ghci Adventurers.hs
GHCi, version 8.6.3: http://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.
[*Adventurers> teste
->[P1,P2]
<-[P1]
->[P5,P10]
<-[P2]
->[P1,P2]
```

Figura 3: Teste da funcionalidade extra adicionada

5 UPPAAL VS MONADS

A utilização do UPPAAL permite uma visualização esquematizada do problema sendo, por vezes, mais simples a sua compreensão e deteção de erros, conseguindo, manualmente, desenhar a linha de pensamento e visualizar diretamente se está de acordo com o que pretendemos. Este também, por exemplo, na situação dos aventureiros chegarem ao outro lado em 17 minutos apresentados num esquema a sua solução.

Um aspeto negativo do UPPAAL é que este, dependendo do pretendido, pode não ser de fácil manipulação, podemos sofrer várias alterações ou até mesmo, quando pretendemos retratar outras situações no nosso autómato teremos de definir novas expressões CTL.

Monads são functores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Com a utilização de *Monads* é possível descrever efeitos computacionais tão díspares quanto entrada / saída, notação de compreensão, atualização de variáveis de estado, comportamento probabilístico, dependência de contexto, comportamento parcial, de uma maneira elegante e uniforme. Além disso, os operadores das *Monads* exibem propriedades notáveis que tornam possível a causa desses efeitos computacionais.

Porém, quanto à validação e verificação de propriedades sobre o sistema modelado, a ferramenta UPPAAL encontra-se equipada com um *model-checker* eficiente que, em conjunto com as expressões CTL definidas pelo utilizador, permite verificar vários tipos de comportamentos que, na linguagem Haskell, podem ser difíceis de explicitar e mesmo impossíveis de testar. Por exemplo, para propriedades de *safety*, ou seja, propriedades nas quais uma condição específica mantém-se verdadeira durante todos os estados, é inexecutável verificar para todas as iterações a sua veracidade pois o tempo seria interminável. Para melhorar este sentido, seria necessário uma mecânica diferente quanto à modelação do sistema que permitisse uma maior eficiência e facilidade na validação de propriedades.

6 Conclusão

Com a elaboração deste projeto, concluímos que utilizar **MONADS** pode ser muito útil para a representação de sistemas de tempo real. Porém, o desenvolvimento do sistema pode revelar-se laborioso devido à sua complexidade mas assim que conseguimos a implementação monádica, tronou-se muito simples chegar a nossa solução do problema proposto. Durante a elaboração deste trabalho, passamos por várias fases em que estruturamos a nossa linha de pensamento e com o auxílio da implementação monádica foi fácil colocá-lo em prática.

Achamos importante referir que há sempre espaço para melhorar, desde tornar, por vezes, otimizar o código desenvolvido com certas modificações cirúrgicas ou até mesmo implementar novas definições que deêm resposta a mais questões colocadas pois a solução desenvolvida é bastante adaptável.

Contudo, foi com a utilização do **Haskell** e a a realização de "monadificações"que conseguimos, com sucesso, dar resposta aos problemas do enunciado, reconhecendo que esta etapa foi concluída com sucesso e que tivemos a possibilidade de aplicar na prática todos os conhecimentos adquiridos nesta área de estudo.