Universidade do Minho
Escola de Engenharia

Modelling and analysis of real-time systems in Haskell

Mestrado integrado em Engenharia Informática
Métodos Formais em Engenharia de Software
Arquitetura e Cálculo
4º Ano, 2º Semestre

A82441 - Alexandre Pinho   A82313 - Pedro Gonçalves

Braga, junho de 2020

Conteúdo

1 Introdução

O Haskell é uma linguagem de programação puramente funcional de propósito geral. Esta linguagem suporta a utilização de mónadas, um conceito matemático que permite a definição e composição de funções que lidam com efeitos implicitamente.

Este relatório é redigido no âmbito da UC de Arquitetura e Cálculo do perfil de mestrado MFES do curso de Engenharia Informático da Universidade do Minho. Ao longo das próximas secções, será descrito o problema proposto, o modelo utilizado para o resolver, e a especificação das propriedades também pedidas.

2 Descrição do problema

O problema discutido neste projeto consiste em modelar a travessia de quatro aventureiros sobre uma ponte. Contudo, a travessia dos aventureiros necessita de respeitar algumas restrições: no máximo só podem atravessar duas pessoas em simultâneo, e é necessário uso de uma lanterna durante as travessias. Neste projeto considerou-se que apenas existe uma única lanterna, e que cada aventureiro demora um certo tempo a fazer a travessia da ponte, sendo que aquando de uma travessia em pares, essa vai possuir a duração do aventureiro mais lento.

3 Solução Haskell

Ao longo deste capítulo, vai ser descrita a forma como o grupo abordou o problema e também a maneira como foram implementadas as soluções em código, tanto para o problema base como para as tarefas opcionais realizadas pelo grupo, especificamente a manipulação de traços em vez de estados finais, e a verificação de propriedades de *safety* sob traços.

3.1 Representação do problema

É definida a estrutura de tipos que representam os conceitos relevantes do problema. Concretamente, o tipo `Objects` é um `Either` habitado por entidades que podem ser um aventureiro (`Adventurer`), ou a lanterna. Existem quatro aventureiros, pelo que o tipo `Adventurer` tem quatro valores possíveis.

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

Cada aventureiro demora um certo período de tempo a atravessar a ponte, dado pela função `getTimeAdv`.

```haskell
getTimeAdv :: Adventurer -> Int
getTimeAdv P1 = 1
getTimeAdv P2 = 2
```

```haskell
getTimeAdv P5 = 5
getTimeAdv P10 = 10
```

Finalmente, é necessário mapear cada objeto ao lado da ponte em que ele se encontra. Este mapeamento pode ser implementado como uma função estado de objetos para booleanos, sendo que, por convenção, o valor `False` representa o lado esquerdo (inicial) e `True` o lado direito (final). Por conveniência, são definidos também os estados inicial `gInit` e final `gFinal`.

```haskell
type State = Objects -> Bool

gInit :: State
gInit = const False

gFinal :: State
gFinal = const True
```

Para manipular o estado são também definidas duas funções auxiliares, a `changeState` que altera o lado de um objeto, e a `mChangeState` que altera o lado de todos os objetos de uma dada lista.

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

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

3.2 Mónadas Duration e ListDur

Uma mónada é um conceito matemático que permite definir e compor funções que lidam com efeitos implicitamente. Concretamente, uma mónada é definida pelo seu construtor de tipo, pela sua função de unidade (`return` em Haskell) e pela operação de *Kleisli lifting* (implementada indiretamente através do operador *bind* em Haskell).

Posto isto, é definida a mónada Duration, que associa a um valor do tipo parametrizado um inteiro que representa uma duração de tempo. Assim, como se pode ver no excerto de código apresentado a seguir, a função de unidade transforma um valor numa computação com duração nula, e o operador de *bind* efetua a composição de duas computações pela soma das suas durações. São omitidas as definições das instâncias das classes `Functor` e `Applicative`; o código completo encontra-se em anexo.

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

remDur :: Duration a -> (Int, a)
remDur (Duration x) = x

getDuration :: Duration a -> Int
```

```
getDuration (Duration (d,x)) = d

getValue :: Duration a -> a
getValue (Duration (d,x)) = x

instance Monad Duration where
    (Duration (i,x)) >>= k = Duration (i + (getDuration (k x)), getValue (k x))
    return x = (Duration (0,x))
```

Pela combinação de mónadas é possível combinar efeitos. Para a modelação do problema proposto, é necessário considerar as várias possibilidades para as transições entre estados. Assim, é implementada a mónada `ListDur`, que utiliza as listas para encapsular o não-determinismo.

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

remLD :: ListDur a -> [Duration a]
remLD (LD x) = x

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

3.3 Implementação para problema base

Utilizando a mónada `ListDur` parametrizada no tipo `State`, cujo valor representa o estado dos objetos num certo instante de tempo, é possível definir funções que produzem todos os estados a que é possível chegar (em n transições) a partir de um dado estado inicial.

A função `allValidPlays` produz o resultado de aplicar cada transição possível ao dado estado, produzindo todos os estados adjacentes e associando uma duração a cada estado, que é o tempo que a transição demora.

Para tal, são selecionados os aventureiros que se encontram do mesmo lado da lanterna e construída a lista de todas as transições possíveis (com recurso à função auxiliar `subsets`), isto é, os objetos que mudam de lado entre os estados inicial e final. Para cada transição, é calculada a duração e aplicada a função `mChangeState`. Por fim, é construído o valor do tipo `ListDur`, para ser possível aplicar transições adicionais no futuro.

```
allValidPlays :: State -> ListDur State
allValidPlays s =
    let lantern = s $ Right ()
```

```
      adventurers = filter (\x -> s x == lantern)
                            [Left P1, Left P2, Left P5, Left P10]
      adventurerMoves = map (\l -> (Right ()) : l)
                                (subsets 1 adventurers ++
                                 subsets 2 adventurers)
  in LD $ map (\moves ->
                  Duration (maximum (map getTimeAdv (lefts moves)),
                            mChangeState moves s)) adventurerMoves

subsets 0 _ = [[]]
subsets _ [] = []
subsets n (x : xs) = map (x :) (subsets (n - 1) xs) ++ subsets n xs
```

Com a função `allValidPlays` definida, é possível de uma forma concisa definir as funções `exec` e `execUpTo` que, dado um número de transições n a percorrer e um estado inicial, produzem, respetivamente, todos os estados ao fim de n transições, e todos os estados ao fim de i transições, para todo o i entre 0 e n.

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

execUpTo :: Int -> State -> ListDur State
execUpTo n s = manyChoice $ map (\i -> exec i s) [0..n]
```

3.3.1 Verificação de propriedades

Para verificar as propriedades propostas, é apenas necessário efetuar um teste por estado obtido através a função `execUpTo`. Este teste verifica se todos os objetos se encontram do lado direito da ponte (pela comparação ao estado `gFinal`), e, dependendo da propriedade, verifica se o tempo que demorou a chegar ao estado foi menor ou igual a 17, ou menor a 17. Apenas a primeira propriedade é verificada, como esperado.

```
leq17 :: Bool
leq17 =
  any reachedAndLeq17 (remLD $ execUpTo 5 gInit)
  where reachedAndLeq17 (Duration (d, s)) = d <= 17 &&
                                             s == gFinal

l17 :: Bool
l17 =
  any reachedAndL17 (remLD $ execUpTo 5 gInit)
  where reachedAndL17 (Duration (d, s)) = d < 17 &&
```

```
                                        s == gFinal
```

3.4 Implementação para extensão do problema

Para possibilitar a verificação de propriedades em traços, a mónada é parametrizada no tipo `[State]`, que representa uma sequência de estados, ou seja, um traço.

Assim, é relativamente simples adaptar o código anterior: na função `allValidPlays'`, os estados calculados formam cada um um traço com apenas um estado; as funções `exec'` e `execUpTo'` passam a receber um traço inicial para facilitar o passo recursivo; e a função `exec'` trata de aplicar a função `allValidPlays'` apenas no ultimo estado de cada traço, e de ir acumulando em argumento o traço atual.

```
allValidPlays' :: State -> ListDur [State]
allValidPlays' s =
    let lantern = s $ Right ()
        adventurers = filter (\x -> s x == lantern)
                          [Left P1, Left P2, Left P5, Left P10]
        adventurerMoves = map (\l -> (Right ()) : l)
                              (subsets 1 adventurers ++
                               subsets 2 adventurers)
    in LD $ map (\moves ->
                    Duration (maximum (map getTimeAdv (lefts moves)),
                              [mChangeState moves s])) adventurerMoves

exec' :: Int -> [State] -> ListDur [State]
exec' 0 s = return s
exec' n s = do s' <- allValidPlays' (last s)
               s'' <- exec' (n - 1) (s ++ s')
               return s''

execUpTo' :: Int -> [State] -> ListDur [State]
execUpTo' n s = manyChoice $ map (\i -> exec' i s) [0..n]
```

3.4.1 Verificação de propriedades

A principal vantagem de lidar com traços é poder inspeccionar o comportamento do sistema para poder saber como e porquê uma propriedade se verifica ou não. No caso das propriedades `leq17'` e `l17'`, obtemos a lista de traços que verificam as condições definidas. Se esta for vazia, significa que a propriedade não é verificada. Na primeira propriedade, são obtidos dois traços que satisfazem as condições, o que significa que existe mais que uma solução ótima ao problema. Na segunda propriedade não obtemos traço algum, indicando, como já visto, que não existem soluções melhores.

```haskell
leq17' :: [(ListDur [State])]
leq17' =
    [LD [trace] | trace <- remLD $
                               execUpTo' 5 [gInit], reachedAndLeq17 trace]
    where reachedAndLeq17 (Duration (d, s)) = d <= 17 && last s == gFinal

l17' :: [(ListDur [State])]
l17' =
    [LD [trace] | trace <- remLD $
                               execUpTo' 5 [gInit], reachedAndL17 trace]
    where reachedAndL17 (Duration (d, s)) = d < 17 && last s == gFinal
```

Em adição às duas propriedades anteriores, são verificadas três propriedades de *safety*:

- `checkLanternSafety`: verifica se a lanterna muda de lado a cada transição, e se apenas mudam de lado aventureiros que estejam no mesmo lado da lanterna.

```haskell
checkLanternSafety :: Int -> Bool
checkLanternSafety n =
    let traces = remLD $ exec' n [gInit]
    in all (checkTrace . getValue) traces
       where checkTrace (h1:h2:t) =
                 (h1 (Right ())) /= (h2 (Right ())) &&
                                    (checkCrossing h1 h2) &&
                                    checkTrace (h2:t)
             checkTrace _ = True
             checkCrossing t1 t2 = all (\a -> t1 a == t2 a ||
                                           t2 a == t2 (Right ()))
                                     [Left P1, Left P2,
                                      Left P5, Left P10]
```

- `checkAdventurersSafety`: verifica se, em cada transição, atravessam a ponte no máximo dois aventureiros.

```haskell
checkAdventurersSafety :: Int -> Bool
checkAdventurersSafety n =
    let traces = remLD $ exec' n [gInit]
    in all (checkTrace . getValue) traces
        where checkTrace (h1:h2:t) = (checkCrossingLeq2 h1 h2) &&
                                     checkTrace (h2:t)
              checkTrace _ = True
              checkCrossingLeq2 t1 t2 = (count (\a -> t1 a /= t2 a)
                                            [Left P1, Left P2,
                                             Left P5, Left P10]) <= 2
```

- `checkTraceDuration`: verifica se a duração dos traços é calculada corretamente.

```haskell
checkTraceDuration :: Int -> Bool
checkTraceDuration n =
    let traces = remLD $ execUpTo' n [gInit]
    in all (\t -> (calculateDuration . getValue) t ==
                  (getDuration t)) traces
        where calculateDuration (h1:h2:t) =
                  (calculateTransitionDuration h1 h2) +
                  calculateDuration (h2:t)
              calculateDuration _ = 0
              calculateTransitionDuration t1 t2 =
                  let adventurers = filter (\x -> t1 x /= t2 x)
                                           [Left P1, Left P2,
                                            Left P5, Left P10]
                  in maximum $ map getTimeAdv $ lefts adventurers
```

Todas as propriedades de *safety* são verificadas, o que profere confiança ao modelo implementado e à solução obtida.

4 UPPAAL vs Haskell

Após concluída a implementação do modelo em Haskell, é possível fazer uma análise crítica às vantagens e desvantagens do uso de Haskell para este tipo de problemas, comparativamente ao UPPAAL usado no trabalho anterior.

Ao contrário do UPPAAL, onde o comportamento das transições e da duração já estão implementados de raiz no sistema, se um utilizador pretender modelar um sistema semelhante ao que foi modelado neste projeto em Haskell, necessitará de implementar o comportamento das transições e da duração manualmente utilizando mónadas, o que poderá ser um contra-tempo caso esse utilizador não possua experiência sobre mónadas.

Outra inconveniência na utilização de Haskell na modelação deste tipo de problemas, é o facto da especificação de invariantes sobre o modelo ser mais difícil comparativamente à especificação de propriedades em UPPAAL, pois nesta última é possível utilizar lógica CTL, que possui operadores próprios que simplificam bastante a especificação de propriedades sobre o modelo.

Outra vantagem relativa à utilização de UPPAAL, é que esta ferramenta possui um simulador que permite uma fácil visualização do comportamento do sistema com o passar do tempo, ao invés do Haskell, que por ser uma ferramenta de um âmbito muito mais geral, não possui nenhum tipo de funcionalidade própria para facilitar a visualização do comportamento do sistema, o que juntamente com a mais difícil especificação de propriedades, leva a que seja mais complicado verificar se um sistema possui o comportamento esperado.

Por fim, a única desvantagem que o grupo considerou existir relativamente à utilização do UPPAAL, é que esta ferramenta requer que os utilizadores possuam conhecimentos sobre autómatos temporais, o que faz com que o Haskell se torne uma opção mais acessível para a maioria dos programadores, pois não necessita de nenhum conhecimento extra para conseguir implementar modelos como o que foi abordado nestes projetos.

5 Conclusão

Com este projeto, o grupo considera que os seus conhecimentos sobre mónadas aumentaram, permitindo perceber a sua utilidade para a modelação de sistemas. Foi possível perceber as principais funcionalidades das mónadas em Haskell, e utiliza-las para construir e validar modelos.

Em suma, este projeto permitiu ao grupo aumentar os seus conhecimentos sobre a utilização de mónadas e ferramentas como o Haskell na modelação de sistemas.

A Modelo Travessia

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

import Data.List
import Data.Either
import Control.Monad

import DurationMonad

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

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

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 ())]

gInit :: State
gInit = const False

gEx1 :: State
gEx1 (Left P1) = True
gEx1 _  = False

gFinal :: State
gFinal = const True
```

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

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

allValidPlays :: State -> ListDur State
allValidPlays s =
    let lantern = s $ Right ()
        adventurers = filter (\x -> s x == lantern)
                          [Left P1, Left P2, Left P5, Left P10]
        adventurerMoves = map (\l -> (Right ()) : l)
                              (subsets 1 adventurers ++
                               subsets 2 adventurers)
    in LD $ map (\moves ->
                   Duration (maximum (map getTimeAdv (lefts moves)),
                             mChangeState moves s)) adventurerMoves

subsets 0 _ = [[]]
subsets _ [] = []
subsets n (x : xs) = map (x :) (subsets (n - 1) xs) ++ subsets n xs


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

execUpTo :: Int -> State -> ListDur State
execUpTo n s = manyChoice $ map (\i -> exec i s) [0..n]


leq17 :: Bool
leq17 = any reachedAndLeq17 (remLD $ execUpTo 5 gInit)
            where reachedAndLeq17 (Duration (d, s)) = d <= 17 && s == gFinal


l17 :: Bool
l17 = any reachedAndL17 (remLD $ execUpTo 5 gInit)
          where reachedAndL17 (Duration (d, s)) = d < 17 && s == gFinal
```

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

remLD :: ListDur a -> [Duration a]
remLD (LD x) = x

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


instance Applicative ListDur where
   pure x = LD [Duration (0, x)]
   l1 <*> l2 = LD $ do x <- remLD l1
                       y <- remLD l2
                       g (remDur x, remDur y) where
                           g ((d1, f), (d2, x)) =
                            return $ Duration (d1 + d2, f x)

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

manyChoice :: [ListDur a] -> ListDur a
manyChoice = LD . concat . (map remLD)
--------------------------------------------------------------------------------

allValidPlays' :: State -> ListDur [State]
allValidPlays' s =
    let lantern = s $ Right ()
        adventurers = filter (\x -> s x == lantern)
                             [Left P1, Left P2, Left P5, Left P10]
        adventurerMoves = map (\l -> (Right ()) : l)
                                 (subsets 1 adventurers ++
                                  subsets 2 adventurers)
    in LD $ map (\moves ->
                    Duration (maximum (map getTimeAdv (lefts moves)),
                              [mChangeState moves s])) adventurerMoves

exec' :: Int -> [State] -> ListDur [State]
```

```haskell
exec' 0 s = return s
exec' n s = do s' <- allValidPlays' (last s)
               s'' <- exec' (n - 1) (s ++ s')
               return s''

execUpTo' :: Int -> [State] -> ListDur [State]
execUpTo' n s = manyChoice $ map (\i -> exec' i s) [0..n]

leq17' :: [(ListDur [State])]
leq17' =
    [LD [trace] | trace <- remLD $
                              execUpTo' 5 [gInit], reachedAndLeq17 trace]
    where reachedAndLeq17 (Duration (d, s)) = d <= 17 && last s == gFinal

l17' :: [(ListDur [State])]
l17' =
    [LD [trace] | trace <- remLD $
                              execUpTo' 5 [gInit], reachedAndL17 trace]
    where reachedAndL17 (Duration (d, s)) = d < 17 && last s == gFinal

checkLanternSafety :: Int -> Bool
checkLanternSafety n =
    let traces = remLD $ exec' n [gInit]
    in all (checkTrace . getValue) traces
       where checkTrace (h1:h2:t) =
                   (h1 (Right ())) /= (h2 (Right ())) &&
                                      (checkCrossing h1 h2) &&
                                      checkTrace (h2:t)
             checkTrace _ = True
             checkCrossing t1 t2 = all (\a -> t1 a == t2 a ||
                                              t2 a == t2 (Right ()))
                                        [Left P1, Left P2,
                                         Left P5, Left P10]

count :: (a -> Bool) -> [a] -> Int
count _ [] = 0
count p (x:xs) | p x       = 1 + count p xs
               | otherwise = count p xs

checkAdventurersSafety :: Int -> Bool
checkAdventurersSafety n =
    let traces = remLD $ exec' n [gInit]
    in all (checkTrace . getValue) traces
        where checkTrace (h1:h2:t) = (checkCrossingLeq2 h1 h2) &&
                                      checkTrace (h2:t)
              checkTrace _ = True
```

```haskell
            checkCrossingLeq2 t1 t2 = (count (\a -> t1 a /= t2 a)
                                             [Left P1, Left P2,
                                              Left P5, Left P10]) <= 2

checkTraceDuration :: Int -> Bool
checkTraceDuration n =
    let traces = remLD $ execUpTo' n [gInit]
    in all (\t -> (calculateDuration . getValue) t ==
                  (getDuration t)) traces
        where calculateDuration (h1:h2:t) =
                  (calculateTransitionDuration h1 h2) +
                  calculateDuration (h2:t)
              calculateDuration _ = 0
              calculateTransitionDuration t1 t2 =
                  let adventurers = filter (\x -> t1 x /= t2 x)
                                           [Left P1, Left P2,
                                            Left P5, Left P10]
                  in maximum $ map getTimeAdv $ lefts adventurers
```

B Mónada Duração

```haskell
module DurationMonad where

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

remDur :: Duration a -> (Int, a)
remDur (Duration x) = x

getDuration :: Duration a -> Int
getDuration (Duration (d,x)) = d

getValue :: Duration a -> a
getValue (Duration (d,x)) = x

instance Functor Duration where
  fmap f (Duration (i,x)) = Duration (i,f x)

instance Applicative Duration where
  pure x = (Duration (0,x))
  (Duration (i,f)) <*> (Duration (j, x)) = (Duration (i+j, f x))

instance Monad Duration where
    (Duration (i,x)) >>= k = Duration (i + (getDuration (k x)), getValue (k x))
    return x = (Duration (0,x))

wait1 :: Duration a -> Duration a
wait1 (Duration (d,x)) = Duration (d+1,x)

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