Universidade do Minho
Escola de Engenharia

Programação Ciber-Física 2023/2024

Trabalho Prático 2

Ricardo Lopes Santos Silva :: pg54188; Afonso Xavier Cardoso Marques :: pg53601

I. Introdução

Neste trabalho, abordaremos a modelação e análise de um sistema ciber-físico utilizando a linguagem de programação Haskell, com ênfase no uso de monads. Monads são uma poderosa abstração em Haskell que permitem tratar efeitos como estado, I/O, e não-determinismo de uma forma funcional e elegante. A tarefa proposta envolve a modelação de um cenário específico onde quatro aventureiros precisam de atravessar uma ponte de corda, respeitando restrições de segurança e tempo. Através desta modelação, pretendemos demonstrar a viabilidade de cumprir os requisitos temporais impostos, além de garantir que todas as regras de segurança sejam respeitadas.

O relatório elaborado explica o código desenvolvido e as conclusões obtidas durante a análise. Este exercício não só ilustra a aplicação prática das monads em Haskell, mas também destaca a importância da modelação precisa em sistemas ciber-físicos complexos.

A. *Contextualização*

No meio da noite, quatro aventureiros encontram uma ponte de corda velha que atravessa um desfiladeiro profundo. Por razões de segurança, decidem que não mais do que duas pessoas devem atravessar a ponte ao mesmo tempo e que uma lanterna precisa de ser carregada por um deles em cada travessia. Eles têm apenas uma lanterna. Os quatro aventureiros não são igualmente habilidosos: atravessar a ponte leva-lhes 1, 2, 5 e 10 minutos, respetivamente. Um par de aventureiros atravessa a ponte em um tempo igual ao do mais lento dos dois aventureiros.

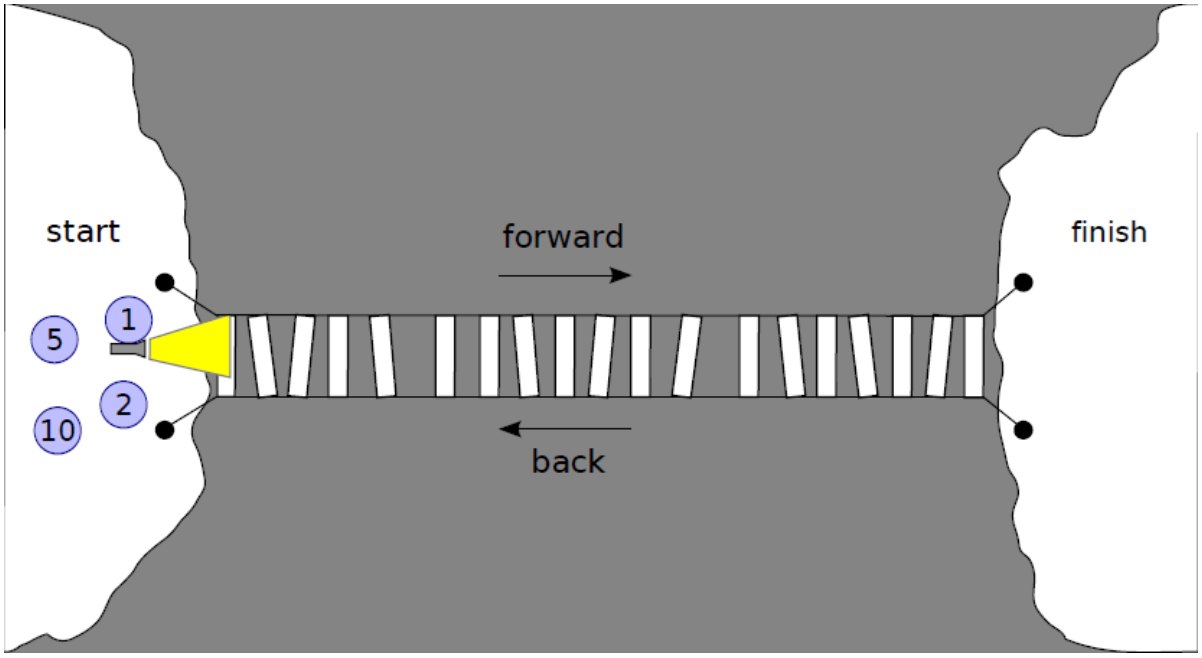

Fig. 1: Cenário em estudo

Um dos aventureiros afirma que eles não podem todos estar do outro lado em menos de 19 minutos. Um companheiro discorda e afirma que isso pode ser feito em 17 minutos.

II. Exercício 1

A primeira tarefa é verificar as afirmações seguintes utilizando Haskell. Especificamente, será preciso:

1) **modelar** o sistema acima usando **monads**, em particular as de duração e não-determinísticas;
2) mostrar que é realmente **possível** que todos os aventureiros estejam do outro lado em 17 minutos;
3) mostrar que é **impossível** que todos os aventureiros estejam do outro lado em menos de 17 minutos.

Para cumprir esta tarefa, completou-se o código no anexo (Adventurers.hs), ou seja, adicionou-se uma definição às funções que carecem de uma definição, seguindo os comentários presentes no código. Para auxilio à resolução, usou-se como inspiração o monad de duração dos slides e do código Haskell que foi previamente fornecido pelo docente. Também se analisou detalhadamente o código referente ao *Knight's quest*, mais particularmente, o monad *ListDur*.

*1) **Código desenvolvido**:*
Analisando o código fornecido, notamos que o tipo Objects é declarado da seguinte forma:

```
1
2 type Objects = Either Adventurer ()
```

Tendo em conta o contexto do problema, onde existem quatro aventureiros e uma lanterna, e os cinco têm de atravessar, faz sentido declarar a lanterna como sendo um dado do tipo Objects. Fizeram-se as seguintes declarações sobre a lanterna:

```
1
2 -- lanterna
3 lamp :: Objects
4 lamp = Right ()
5
6 -- mover a lanterna
7 moveLamp :: State -> State
8 moveLamp = changeState lamp
```

Onde *lamp* é definido como sendo o lado direito de uma alternativa Objects, e a movimentação da lanterna envolve a mudança do seu estado. De forma a tornar a resolução mais adequada ao código fornecido, criaram-se as definições *getTimeOAdv* e *adventurers* onde os aventureiros em formato Adventurer são tratados como um dado Objects.

```
-- igual a getTimeAdv só que para os objetos
getTimeOAdv :: Objects -> Int
getTimeOAdv (Left a) = getTimeAdv a

-- lista dos aventureiros como objetos
adventurers :: [Objects]
adventurers = map Left [P1, P2, P5, P10]
```

getTimeOAdv apenas devolve o resultado de *getTimeAdv* para o lado esquerdo de Objects e *adventurers* devolve uma lista de Objects, onde aplica a função Left a cada elemento da lista [P1, P2, P5, P10], isto é, cria uma lista de valores do tipo Either Adventurer (), onde cada elemento é do tipo Left Adventurer.

Quanto às funções fornecidas para completar, vai-se agora abordar a estratégia implementada. Comecemos então pela função **getTimeAdv**:

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

Esta função é de implementação bastante simples. Serve apenas para indicar o tempo que cada aventureiro demora a passar a ponte. De seguida, também é requisitada a definição da função **allValidPlays**, que para um certo estado devolve todas as jogadas possíveis dos aventureiros. Analogicamente, pode ser vista como um GPS (Global Positioning System), onde para um determinado ponto de partida, existem vários caminhos com tempos de viagem diferentes.

```
allValidPlays :: State -> ListDur State
allValidPlays s = manyChoice [moveOne s, moveTwo s]
```

Tendo em conta que a função recebe um estado *s* como argumento, e quer-se saber quais são todas as jogadas possíveis para *s*, temos de ter em conta se apenas se vai mover um aventureiro ou dois ao mesmo tempo. Para isso foram criadas as funções auxiliares *moveOne* e *moveTwo*.

```
moveOne :: State -> ListDur State
moveOne s = LD (map f canCross)
  where
    -- canCross :: [Objects]
    canCross = filter ((== s lamp) . s) adventurers
    -- f :: State -> Duration State
    f x = wait (getTimeOAdv x) (return (changeState x (moveLamp s)))


moveTwo :: State -> ListDur State
moveTwo s = LD (map f po)
  where
    -- pares dos aventureiros que podem atravessar a ponte
    -- po :: [(Objects, Objects)]
    po = makePairs (filter ((== s lamp) . s) adventurers)

    -- duracões dos pares após atravessarem
    -- f :: (Objects, Objects) -> State
    f (x, y) = wait (max (getTimeOAdv x) (getTimeOAdv y)) (return (changeState y (changeState x (moveLamp s
    ))))
```

- *moveOne* recebe o estado como argumento e devolve todas os estados atingíveis a partir de *s* juntamente com a duração de cada um. Para esse efeito, precisa de filtrar todos aventureiros que podem atravessar a ponte, ou seja, todos os que estão no mesmo lado que a lanterna (têm o mesmo estado) e de seguida calcula o tempo da travessia para cada um (aplicando em conjunto as mudanças de estado).

- *moveTwo* faz exatamente a mesma coisa, com a particularidade de ter em conta que são dois aventureiros em vez de um, logo, após a filtragem dos aventureiros com estado igual à lanterna, criam-se todos os pares de aventureiros possíveis para a travessia, sendo que a duração da travessia vai ser o valor do aventureiro que demora mais tempo.

Como as duas funções, *moveOne* e *moveTwo*, devolvem uma lista de durações de estados (ListDur), é necessário juntar essas duas listas numa só, ficando apenas uma única lista com todas os movimentos possíveis. Para isso, recorreu-se à função *manyChoice*, disponibilizada no código do docente, que constrói uma lista de durações a partir da concatenação de listas de durações de menor dimensão.

Além destas funções, também foi pedido para implementar as funções **exec**, **leq17** e **l17**. A função **exec**, dado um valor inteiro *n* e um estado inicial, calcula todas as *n* possíveis sequências de movimentos que os aventureiros podem fazer.

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

Sendo que ListDur State é igual a [Duration State], ou seja, uma lista de durações para cada estado, a implementação desta função lida com dois casos diferentes:

1) Caso base:
   Quando o número de passos é 0, a função simplesmente retorna o estado atual *s*, encapsulado em um monad ListDur.

2) Caso recursivo:
   Para um número de passos *n* maior que 0, a função **allValidPlays s** gera todos os estados válidos que podem ser alcançados a partir do estado atual *s*. O resultado é um monad que encapsula uma lista de novos estados com as respetivas durações. Em seguida, usando a notação *do*, extraí-se cada novo estado *s'* da lista gerada por allValidPlays s. Após isso, a função **exec (n-1) s'** é chamada recursivamente. O processo mantém-se até que *n* se torne 0. A recursividade permite explorar todos os possíveis caminhos dentro do número de passos especificado.

A função **leq17** verifica a possibilidade de todos os aventureiros passarem para o lado direito em 17 ou menos segundos, executando apenas um total de cinco passos. Para essa finalidade, calculam-se as cinco possíveis sequências de movimentos que os aventureiros podem fazer para o estado inicial pré-definido como *gInit*. A seguir, procura-se na lista devolvida se existe pelo menos uma ocorrência de um estado, *s*, onde o tempo de duração *x* seja menor ou igual a 17 e os estados de todos os aventureiros sejam iguais a *s*. Tendo em conta que a definição de State devolve um valor booleano para um objeto (aventureiro ou lanterna), o lado direito da conjunção (&&) só irá devolver True se *s* for True e todos os objetos tiverem um valor de estado igual a True, ou seja, se estiverem todos no lado direito da ponte.

```
1
2 leq17 :: Bool
3 leq17 = any f (remLD (exec 5 gInit))
4           where
5             f (Duration (x, s)) = x<=17 && all s adventurers && s lamp
```

A lógica para a função **le17** é semelhante no entanto verifica apenas a possibilidade de todos atravessarem a ponte em menos de 17 segundos caso a execução seja de cinco passos, oito passos ou dez passos. Vale a pena referir que para mais de dez passos, o processo começava a ficar demasiado pesado e lento. Como se vai ver mais tarde, esta função devolve sempre False.

```
1
2 l17 :: Bool
3 l17 = any f (remLD (exec 5 gInit)) || any f (remLD (exec 8 gInit)) || any f (remLD (exec 10 gInit))
4       where
5         f (Duration (x, s)) = x < 17 && all s adventurers && s lamp
```

*2) **Testes e Resultados Obtidos**:*

De forma a garantir que a resolução apresentada é de alguma forma válida, criou-se um conjunto de testes onde se pretende obter os resultados de quatro testes diferentes. Primeiro testa-se quantas soluções válidas a função **leq17** consegue encontrar. Isto é, para cinco passos, quantos soluções existem que demorem 17 ou menos minutos. Para isso, obteve-se o valor booleano de **leq17**, e de seguida calculou-se quantas soluções válidas existem dentro do total encontrado. O output é feito num ficheiro e será no formato abaixo. É possível verificar que houve sucesso no teste, e que foram encontradas duas soluções válidas.

```
2  Testing leq17:
3  Is it possible for all adventurers to be on the other side in <=17 min and not exceeding 5 moves? True
4  Valid solution(s) for leq17 found: 2 out of 3484
```

Fig. 2: Output *testLeq17*

O seguinte teste é semelhante, desta vez para a função **l17**. Aqui o resultado permite comprovar que de facto é impossível que todos os aventureiros atravessem a ponte em menos do que 17 minutos.

```
6  Testing l17:
7  Is it possible for all adventurers to be on the other side in < 17 min? False
8  No valid solution(s) for l17 found within the step limit.
```

Fig. 3: Output *testL17*

A imagem em baixo permite ver todas as soluções válidas encontradas para uma execução de cinco passos.

main.hs | output.txt

```
11 Total found (valid or not valid) = 3484
12 All valid final solutions within 5 steps:
13 Time: 19
14 ["True","True","True","True","True"]
15 Time: 20
16 ["True","True","True","True","True"]
17 Time: 23
18 ["True","True","True","True","True"]
19 Time: 19
20 ["True","True","True","True","True"]
21 Time: 20
22 ["True","True","True","True","True"]
23 Time: 33
24 ["True","True","True","True","True"]
25 Time: 17
26 ["True","True","True","True","True"]
27 Time: 23
28 ["True","True","True","True","True"]
29 Time: 33
30 ["True","True","True","True","True"]
31 Time: 20
32 ["True","True","True","True","True"]
33 Time: 21
34 ["True","True","True","True","True"]
35 Time: 24
36 ["True","True","True","True","True"]
37 Time: 20
38 ["True","True","True","True","True"]
39 Time: 21
40 ["True","True","True","True","True"]
41 Time: 34
42 ["True","True","True","True","True"]
43 Time: 17
44 ["True","True","True","True","True"]
45 Time: 24
```

Fig. 4: Output *testPrintFinalStates_All*

Finalmente, as duas soluções obtidas cujo tempo é exatamente 17 minutos numa execução de cinco passos.

```
231 Valid final solutions within 5 steps and exactly 17 minutes:
232 Time: 17
233 ["True","True","True","True","True"]
234 Time: 17
235 ["True","True","True","True","True"]
236
```

Fig. 5: Output *testPrintFinalStates_17*

III. Exercício 2

A segunda tarefa consiste em comparar ambas as abordagens (via UPPAAL e Haskell) para o problema dos aventureiros. Especificamente, o objetivo é fornecer os pontos fortes e fracos das duas abordagens: quais são as (des)vantagens do UPPAAL para este problema? E quanto ao Haskell?

De um modo geral, cada abordagem tem as suas vantagens e desvantagens, como é de esperar.

Na abordagem via **UPPAAL**, conseguimos verificar e validar o modelo, bem como obter uma análise minuciosa dos seus potenciais pontos de falha. Isto porque ao executar simulações, o UPPAAL analisa aleatoriamente todos os casos possíveis e identifica possíveis *deadlocks* no programa. As simulações também são um ponto forte do UPPAAL em relação ao Haskell, pois a sua execução é visual e auxilia no melhor entendimento do verdadeiro comportamento modelado, passo a passo. Isso resulta numa correção melhor e mais rápida de quaisquer comportamentos inesperados. O uso do UPPAAL também não implica diretamente algoritmos de programação, pois é possível simular problemas complexos utilizando as ferramentas simples fornecidas e aproveitar essas ferramentas utilizando formas lógicas para verificar quaisquer requisitos do sistema. Podemos verificar facilmente as propriedades temporais de um modelo a partir de fórmulas CTL.

No entanto, também há algumas desvantagens ao usar o UPPAAL. Modelar sistemas complexos é desafiante e demorado, já que exige bastante planeamento e familiarização com a ferramenta. Requer também alguma abstração do sistema de modo a se adequar às capacidades do UPPAAL o que pode não capturar todos os detalhes do comportamento deste sistema.

Ao trabalhar com **Haskell**, é essencial especificar com precisão os comportamentos do algoritmo em várias circunstâncias. Essa abordagem proporciona uma compreensão mais profunda do problema, uma vez que a programação resulta do estudo da estratégia modelada. O não-determinismo dos monads permite a exploração de todas as possibilidades. Além disso, o forte sistema de tipos do Haskell e sua natureza funcional tornam-no uma ferramenta poderosa para modelar problemas complexos de forma clara e concisa.

Por outro lado, identificar pontos de falha no Haskell às vezes pode envolver tentativa e erro, o que não é a abordagem mais eficiente. Além disso, o Haskell requer a programação de todas as iterações necessárias para obter resultados, uma vez que estes são apenas os casos pré-planeados pelo programador. Isso pode resultar em um número insuficiente de casos de teste para validar o programa.

Em suma, no *UPPAAL*, a verificação de modelos e a análise visual ajudam a identificar e corrigir problemas rapidamente, mas podem ser limitadas em termos de complexidade e escalabilidade. Em *Haskell*, a programação detalhada proporciona um entendimento profundo do problema, mas pode ser menos eficiente na identificação de falhas e exigir um esforço considerável para modelar e validar todos os casos possíveis.

IV. Conclusão

Em suma, consideramos que o trabalho desenvolvido e os resultados apurados são positivos. Destacamos que todas as propriedades pedidas foram implementadas conseguindo provar as mesmas, demonstrando uma clara capacidade em aplicar os conceitos das aulas de Programação Ciber-Física.

Como trabalho futuro, seria também interessante introduzir a visualização dos passos para os casos que se inserissem no ponto ”mostrar que é realmente possível que todos os aventureiros estejam do outro lado em 17 minutos” podendo perceber o melhor o cenário.

REFERENCES

[1] https://uppaal.org/features/
[2] https://haslab.github.io/MFP/PCF/2324/index
[3] https://www.youtube.com/watch?v=7yDmGnA8Hw0
[4] https://www.haskell.org/
[5] https://en.wikipedia.org/wiki/Monad_(functional_programming)