



Universidade do Minho
Escola de Engenharia

Programação Ciber-Física

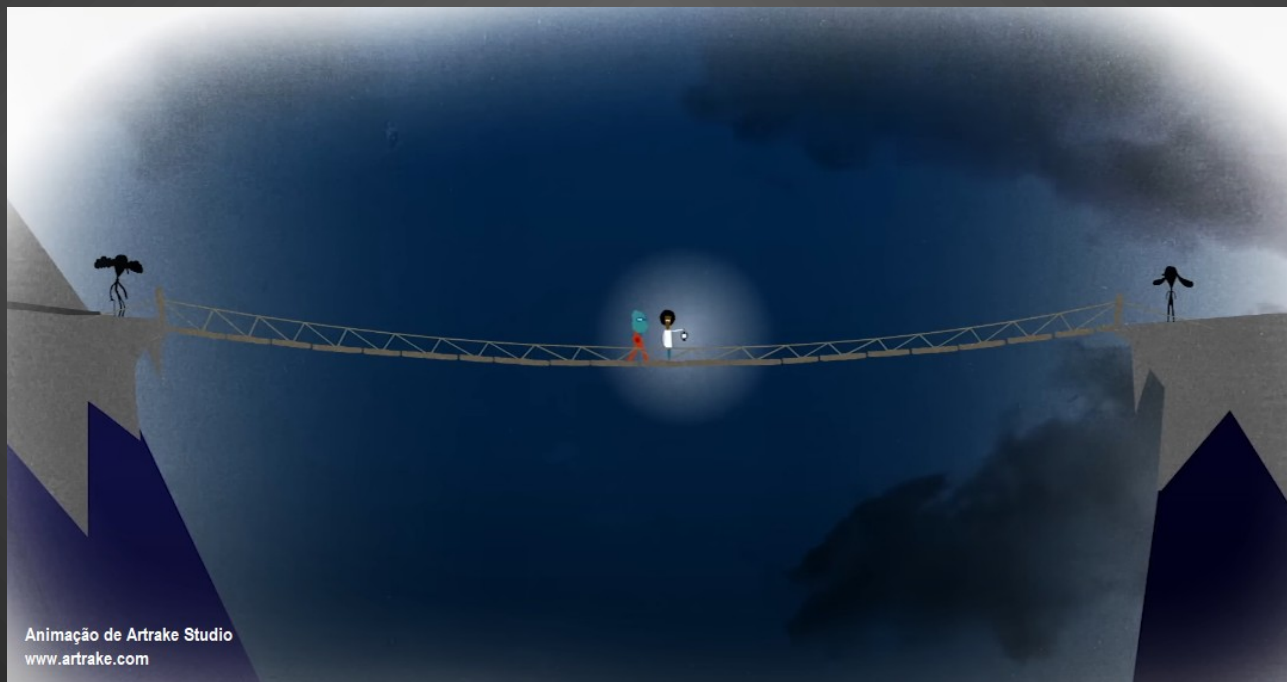
2023/2024

Trabalho Prático 2

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

Contexto

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. 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.



Exercício 1

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

- 1) modelar o cenário 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 ficheiro (Adventurers.hs), ou seja, adicionou-se uma definição às funções que carecem de uma definição.

Utilidades:

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

Declaração de Objects como uma alternativa entre Adventurer ou um valor *void* “()”

```
lamp :: Objects
```

```
lamp = Right ()
```

Declaração da lanterna como lado direito da alternativa Objects

```
moveLamp :: State -> State
```

```
moveLamp = changeState lamp
```

Mover a lanterna

Utilidades:

```
getTimeOAdv :: Objects -> Int
```

```
getTimeOAdv (Left a) = getTimeAdv a
```

```
adventurers :: [Objects]
```

```
adventurers = map Left [P1, P2, P5, P10]
```

Tratamento dos aventureiros como Objects.

Isto é realizado através da criação de uma lista onde a cada dado do tipo Adventurer é aplicada a função Left, criando assim um dado do tipo Objects, facilitando a restante implementação.

getTimeOAdv permite ir buscar o tempo de um aventureiro quando este é representado como Objects.

getTimeAdv:

```
getTimeAdv :: Adventurer -> Int
```

```
getTimeAdv P1 = 1
```

```
getTimeAdv P2 = 2
```

```
getTimeAdv P5 = 5
```

```
getTimeAdv P10 = 10
```

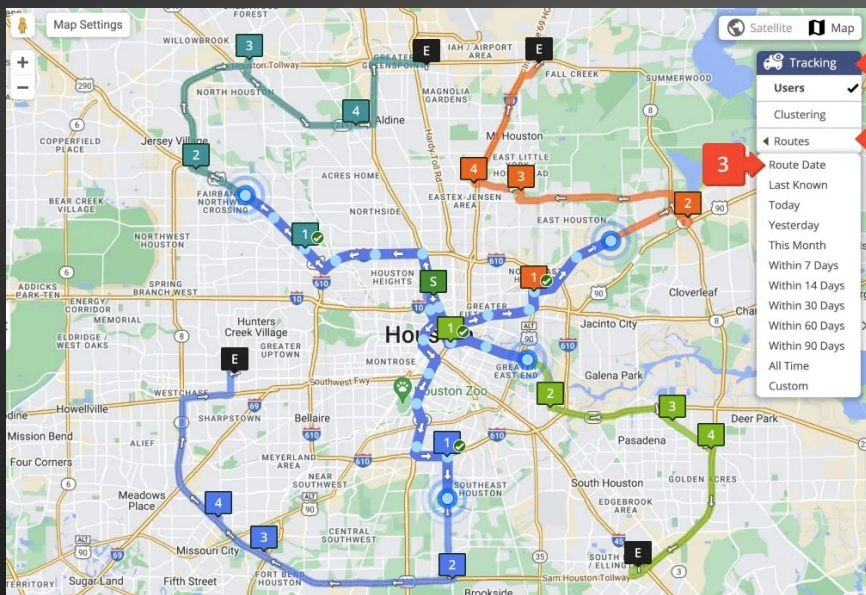
Esta função é de implementação bastante simples.

Serve apenas para indicar o tempo que cada aventureiro demora a passar a ponte.

allValidPlays:

```
allValidPlays :: State -> ListDur State
```

```
allValidPlays s = manyChoice [moveOne s, moveTwo s]
```



A função *allValidPlays* 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.

Resulta da junção, através da função *manyChoice*, de duas funções que devolvem listas de durações de estado: *moveOne* e *moveTwo*.

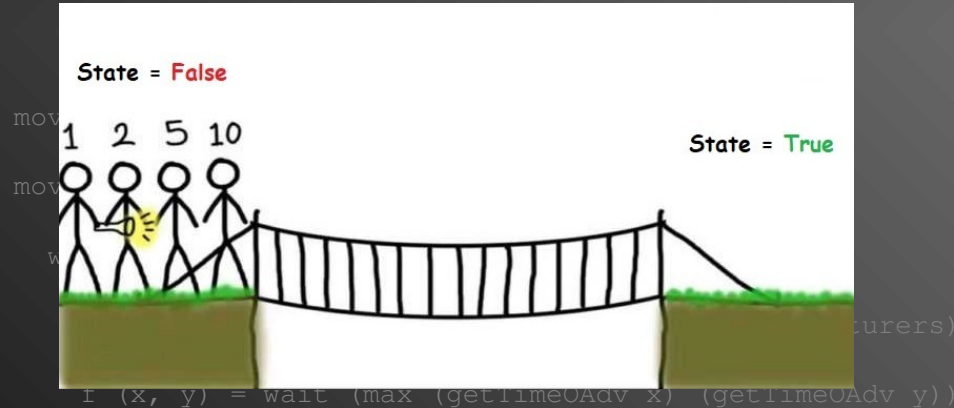

```
moveOne :: State -> ListDur State
```

```
moveOne s = LD (map f canCross)
```

```
where
```

```
canCross = filter ((== s lamp) . s) adventurers
```

```
f x = wait (getTimeOAdv x) (return (changeState x (moveLamp s)))
```



Recebe o estado s como argumento e devolve todos os estados atingíveis a partir de s juntamente com a duração de cada um.

Para esse efeito, precisa de filtrar todos os aventureiros que podem atravessar a ponte, ou seja, todos os que estão no mesmo lado que a lanterna (têm o mesmo estado, True ou False) e de seguida calcula o tempo da travessia para cada um (aplicando em conjunto as mudanças de estado).

```
moveOne :: State -> ListDur State
```

```
moveOne s = LD (map f canCross)
```

```
where
```

```
canCross = filter ((== s lamp) . s) adventurers
```

```
f x = wait (getTimeOAdv x) (return (changeState x (
```

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.

```
moveTwo :: State -> ListDur State
```

```
moveTwo s = LD (map f po)
```

```
where
```

```
po = makePairs (filter ((== s lamp) . s) adventurers)
```

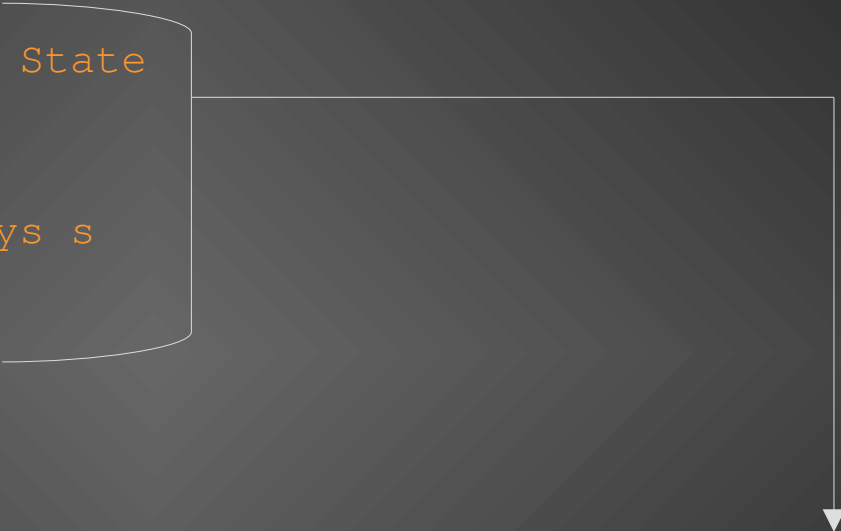
```
f (x, y) = wait (max (getTimeOAdv x) (getTimeOAdv y)) (return (changeState y (changeState x (moveLamp s))))
```

exec:

```
exec :: Int -> State -> ListDur State
```

```
exec 0 s = return s
```

```
exec n s = do s' <- allValidPlays s  
             exec (n-1) s'
```



Dado um valor inteiro n e um estado inicial, calcula todas as n -possíveis sequências de movimentos que os aventureiros podem fazer.

exec:

```
exec :: Int -> State -> ListDur State
```

```
exec 0 s = return s
```



```
exec n s = do s' <- allValidPlays s
```

```
    exec (n-1) s'
```

Caso base:

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

É possível devolver apenas s , por causa do Applicative *ListDur* onde temos $\text{pure } x = \text{LD } [\text{Duration } (0,x)]$.

exec:

```
exec :: Int -> State -> ListDur State
```

```
exec 0 s = return s
```

```
exec n s = do s' <- allValidPlays s
```

```
    exec (n-1) s'
```



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 respectivas 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.

Verificação de propriedades

Duas funções foram declaradas para a verificação das seguintes propriedades:

- 1) mostrar que é realmente possível que todos os aventureiros estejam do outro lado em 17 minutos;
- 2) mostrar que é impossível que todos os aventureiros estejam do outro lado em menos de 17 minutos.

A propriedade 1) é verificada pela função *leq17*, enquanto que a propriedade 2) é analisada pela função *ll7*. Esta verificação permite verificar se o sistema foi bem implementado.

leq17:

```
leq17 :: Bool
```

```
leq17 = any f (remLD (exec 5 gInit))
```

```
    where
```

```
        f (Duration (x, s)) = x<=17 && all s adventurers && s lamp
```

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.

l17:

```
l17 :: Bool
```

```
l17 = any f (remLD (exec 5 gInit)) || any f (remLD (exec 8 gInit)) || any f (remLD (exec 10 gInit))
```

```
  where
```

```
    f (Duration (x, s)) = x < 17 && all s adventurers && s lamp
```

A lógica para a função *l17* 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.

Haskell vs UPPAAL

Haskell:

Vantagens:

- **Especificação Precisa dos Comportamentos:** Haskell exige uma especificação detalhada dos comportamentos do algoritmo em várias circunstâncias, proporcionando uma compreensão mais profunda do problema.
- **Exploração de Todas as Possibilidades:** O não-determinismo dos monads em Haskell permite explorar todas as possibilidades, tornando-o eficaz na modelagem de problemas complexos.
- **Sistema de Tipos Forte e Natureza Funcional:** O forte sistema de tipos e a natureza funcional do Haskell facilitam a criação de modelos claros e concisos, tornando-o uma ferramenta poderosa para a modelagem de problemas complexos.

Desvantagens:

- **Identificação de Pontos de Falha:** Identificar pontos de falha no Haskell pode ser desafiador e envolver tentativa e erro, o que não é a abordagem mais eficiente.
- **Programação de Todas as Iterações:** Haskell requer a programação de todas as iterações necessárias para obter resultados, limitando-se aos casos pré-planeados pelo programador.
- **Número Insuficiente de Casos de Teste:** Devido à necessidade de programar todas as iterações, pode haver um número insuficiente de casos de teste para validar completamente o programa.

UPPAAL:

Vantagens:

- **Análise Minuciosa de Potenciais Falhas:** O UPPAAL é capaz de identificar possíveis deadlocks no programa, ajudando a evitar falhas críticas.
- **Simulações Visuais:** A execução visual das simulações auxilia na compreensão do comportamento modelado, facilitando a identificação e correção de comportamentos inesperados.
- **Ferramentas Simples para Problemas Complexos:** UPPAAL permite simular problemas complexos utilizando ferramentas simples, sem necessidade de algoritmos de programação complexos.
- **Verificação de Propriedades Temporais:** É possível verificar facilmente as propriedades temporais de um modelo utilizando fórmulas CTL.

Desvantagens:

- **Modelagem Desafiante e Demorada:** Modelar sistemas complexos no UPPAAL exige bastante planeamento e familiarização com a ferramenta, sendo uma tarefa desafiante e demorada.
- **Necessidade de Abstração:** É necessário realizar uma abstração do sistema para se adequar às capacidades do UPPAAL, o que pode resultar na perda de detalhes importantes do comportamento do sistema.

Trabalho Futuro

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.