Trabalho Prático 2
Arquitetura e Cálculo - 2019/2020

Bruno Antunes - PG40866
Pedro Moura - PG41094

16 de Junho de 2020

Resumo

Modelação e análise de um sistema de tempo-real em Haskell.

Conteúdo

1 Introdução

No seguimento do primeiro trabalho prático, foi novamente proposta a modelação e análise de um sistema de tempo-real, mas desta vez através da utilização de uma linguagem de programação. O enunciado e os objetivos são os mesmos que no trabalho anterior.

O presente relatório descreve a modelação deste problema através da linguagem HASKELL. Para isto são utilizadas mónadas, que são estruturas matemáticas que nos permitem efetuar computações específicas.

Ainda neste relatório, será também feita uma análise comparativa acerca das duas abordagens que foram seguidas na resolução deste problema: utilizando *timed automata* (TP1) e utilizando uma linguagem de programação.

2 Mónadas

Uma mónada é uma estrutura matemática que é composta por:

- um construtor de tipos T;
- uma função do tipo $\eta : X \to T\ X$ que converte um elemento de um tipo básico num elemento de tipo monádico;
- uma função combinadora $(\_)^*$ (*Kleisli function*), que recebe como argumento uma função do tipo $f : X \to T\ Y$, e devolve como resultado uma função $f^* : T\ X \to T\ Y$.

Para realmente ser uma mónada, as funções acima descritas devem respeitar as seguintes regras: $\eta^* = id$, $f^* \cdot \eta = f$ e $(f^* \cdot g)^* = f^* \cdot g^*$.

3 Resolução

O exercício é resolvido através do uso de *bruteforce*. A estratégia passa por criar uma função que, dado um limite de rondas, gera todos os estados possíveis do sistema no final dessas rondas. Para apresentarmos este conjunto de estados possíveis e conseguirmos realizar operações sobre ele utilizaremos uma mónada chamada **Lista de Durações**.

3.1 Lista de Durações

A mónada da lista de durações tem a seguinte definição:

Set Constructor: $$T_X = X \mapsto (\mathbb{N} \times X)^*$$

Unit: $$\eta_X = singl \cdot \langle \underline{0}, id \rangle$$

Kleisli function: $$\frac{f : X \to (\mathbb{N} \times Y)}{f^* = concat \cdot map\ (\lambda(t, x) \to map\ wait_t\ (f\ x))}$$

Provas:

$$
\begin{array}{ll}
 & \eta^* \\
\equiv & \quad \{\ \text{definition of } (\_)^*\ \} \\
 & concat \cdot map\ (\lambda(t, x) \to map\ wait_t\ (\eta x)) \\
\equiv & \quad \{\ \text{definition of } \eta\ \} \\
 & concat \cdot map\ (\lambda(t, x) \to map\ wait_t\ ((singl \cdot \langle \underline{0}, id \rangle)\ x)) \\
\equiv & \quad \{\ \text{def-comp, def-split, def-const, def-id}\ \} \\
 & concat \cdot map\ (\lambda(t, x) \to map\ wait_t\ (singl\ (0, x))) \\
\equiv & \quad \{\ \text{definition of } singl\ \} \\
 & concat \cdot map\ (\lambda(t, x) \to map\ wait_t\ [(0, x)]) \\
\equiv & \quad \{\ \text{definition of } map\ \} \\
 & concat \cdot map\ (\lambda(t, x) \to [(t, x)]) \\
\equiv & \quad \{\ \text{definition of } singl\ \} \\
 & concat \cdot map\ singl \\
\equiv & \quad \{\ \text{trivial}\ \} \\
 & id \\
\Box &
\end{array}
$$

$$
\begin{array}{ll}
 & (f^* \cdot \eta)\ y \\
\equiv & \quad \{\ \text{definition of } (\_)^*\ \} \\
 & (concat \cdot map\ (\lambda(t,x) \to map\ wait_t\ (f\ x)) \cdot \eta)\ y \\
\equiv & \quad \{\ \text{definition of } \eta\ \} \\
 & (concat \cdot map\ (\lambda(t,x) \to map\ wait_t\ (f\ x)) \cdot (singl \cdot \langle \underline{0}, id \rangle))\ y \\
\equiv & \quad \{\ \text{def-comp } (\times 2), \text{def-split, def-const, def-id, definition of } singl\ \} \\
 & (concat \cdot map\ (\lambda(t,x) \to map\ wait_t\ (f\ x))\ [(0,y)] \\
\equiv & \quad \{\ \text{def-comp, definition of } map\ \} \\
 & concat\ [map\ wait_0\ (f\ y)] \\
\equiv & \quad \{\ \text{definition of } concat\ \} \\
 & map\ wait_0\ (f\ y) \\
\equiv & \quad \{\ map\ wait_0 \equiv map\ id \equiv id, \text{def-id}\ \} \\
 & f\ y \\
\Box &
\end{array}
$$

$$(f^* \cdot g)^* = f^* \cdot g^*$$

3.2 Código

O código aqui descrito encontra-se por completo no ficheiro `Adventurers.hs`.

Comecemos pela definição da mónada da lista de durações em HASKELL:

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

instance Functor ListDur where
fmap f = LD . map f' . remLD
    where f' = \duration -> fmap f duration


instance Applicative ListDur where
```

```
pure = LD . singl . pure
fs <*> xs =
    LD [ f <*> x | f <- remLD fs , x <- remLD xs ]


instance Monad ListDur where
return = LD . singl . return
l >>= f = LD
    . concat
    . map (\(Duration (t,x)) ->
            map (wait t) (remLD (f x)))
    . remLD $ l

-- definicao alternativa
-- l >>= f = LD $ do
--             Duration (d,x) <- remLD l
--             map (wait d) (remLD (f x))
```

Onde η é representada pela função `return` e f^* por `(>>= f)`.

Agora que temos a mónada implementada, podemos passar à resolução do exercício. Na nossa resolução existem 2 funções fundamentais: *exec* e *allValidPlays*.

Começaremos pela função `allValidPlays`, que é definida da seguinte forma:

```
allValidPlays :: State -> ListDur State
allValidPlays s =
    manyChoice [ oneCrossing s, twoCrossing s ]
```

onde `ListDur` é a mónada da lista de durações e as funções `oneCrossing` e `twoCrossing` representam as travessias possíveis com uma e duas pessoas, respetivamente. Esta função gera todas os travessias possíveis a partir de um dado estado.

Cada travessia é gerada pela função `cross`, que tem a seguinte definição:

```
cross :: [Adventurer] -> State -> ListDur State
cross ps s =
    if isCrossValid ps s
```

```
        then LD [
            Duration (maxT,
            mChangeState ((Right ()) : map Left ps) s) ]
        else LD []
    where maxT = maximum (map getTimeAdv ps)
```

No caso da travessia ser válida, i.e. passam no máximo 2 aventureiros e ambos estão do lado da lanterna, a função gera uma lista de durações com apenas um elemento, que contém o estado final da travessia e a sua duração. Caso contrário, gera uma lista vazia.

A função `exec` é definida da seguinte forma:

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

Esta função retorna todos os estados possíveis a partir do estado inicial após n rondas.

Só nos resta agora criar as propriedades de verificação que são pedidas no enunciado:

- É possível que todos os aventureiros estejam do outro lado da ponte em tempo menor ou igual a 17 minutos?

```
leq17 :: Bool
leq17 = any
    (\(Duration (d,s)) ->
        d <= 17 && s == const True)
    (remLD (exec 5 gInit))
```

- É possível que todos os aventureiros estejam do outro lado da ponte em menos de 17 minutos?

```
l17 :: Bool
l17 = any
    (\(Duration (d,s)) ->
        d < 17 && s == const True)
    (remLD (exec 5 gInit))
```

Se corremos estas 2 funções, podemos ver que dão `True` e `False` respetivamente, que é o suposto. Na primeira, a solução final é a seguinte: `Duration`

`(17,["True","True","True","True","True"])`. Podemos ver que todos os aventureiros e a lanterna estão do lado oposto ao inicial, e que a duração foi de 17 minutos. Mas esta resolução apenas nos diz que é possível, e não *como* é possível. Decidimos então fazer uma nova versão, que adiciona um registo de movimentos à lista de durações.

Para isto, precisamos de criar uma nova mónada, como iremos ver a seguir.

4 Resolução - extensão

4.1 Lista de Durações com registo

Esta mónada é muito semelhante à mónada da lista de durações, só que agora acrescentamos um novo campo, que é uma lista onde serão inseridos os registos.

Set Constructor: $$(T_A)_X = X \mapsto (A^* \times (\mathbb{N} \times X))^*$$

Unit: $$\eta_X = singl \cdot \langle \underline{[]}, \langle \underline{0}, id \rangle \rangle$$

Kleisli function: $$\frac{f : X \to (A^* \times (\mathbb{N} \times Y))^*}{f^* = concat \cdot map\ (\lambda(l,(t,x)) \to map\ ((l +\!\!+) \times wait_t)\ (f\ x))}$$

Provas:

$$
\begin{array}{ll}
 & \eta^* \\
\equiv & \quad \{ \text{ definition of } (\_)^* \ \} \\
 & concat \cdot map\ (\lambda(l,(t,x)) \to map\ ((l{+\!\!+}) \times wait_t)\ (\eta\ x)) \\
\equiv & \quad \{ \text{ definition of } \eta \ \} \\
 & concat \cdot map\ (\lambda(l,(t,x)) \to map\ ((l{+\!\!+}) \times wait_t)\ ((singl \cdot \langle \underline{[]}, \langle \underline{0}, id \rangle \rangle)\ x)) \\
\equiv & \quad \{ \text{ def-comp, def-split } (\times 2), \text{ def-const } (\times 2), \text{ def-id } \} \\
 & concat \cdot map\ (\lambda(l,(t,x)) \to map\ ((l{+\!\!+}) \times wait_t)\ (singl\ ([],(0,x))) \\
\equiv & \quad \{ \text{ definition of } singl \ \} \\
 & concat \cdot map\ (\lambda(l,(t,x)) \to map\ ((l{+\!\!+}) \times wait_t)\ [([],(0,x))]) \\
\equiv & \quad \{ \text{ definition of } map \ \} \\
 & concat \cdot map\ (\lambda(l,(t,x)) \to [(l,(t,x))]) \\
\equiv & \quad \{ \text{ definition of } singl \ \} \\
 & concat \cdot map\ singl \\
\equiv & \quad \{ \text{ trivial } \} \\
 & id \\
\Box &
\end{array}
$$

$$
\begin{array}{ll}
 & (f^* \cdot \eta)\ y \\
\equiv & \quad \{\ \text{definition of } (\_)^*\ \} \\
 & (concat \cdot map\ (\lambda(l,(t,x)) \to map\ ((l +\!\!+) \times wait_t)\ (f\ x)) \cdot \eta)\ y \\
\equiv & \quad \{\ \text{definition of } \eta\ \} \\
 & (concat \cdot map\ (\lambda(l,(t,x)) \to map\ ((l +\!\!+) \times wait_t)\ (f\ x)) \cdot (singl \cdot \langle \underline{[]}, \langle \underline{0}, id \rangle \rangle))\ y \\
\equiv & \quad \{\ \text{def-comp } (\times 2), \text{def-split } (\times 2), \text{def-const } (\times 2), \text{def-id, definition of } singl\ \} \\
 & concat \cdot map\ (\lambda(l,(t,x)) \to map\ ((l +\!\!+) \times wait_t)\ (f\ x))\ [([], (0, y))] \\
\equiv & \quad \{\ \text{definition of } map\ \} \\
 & concat\ [map\ (([] +\!\!+) \times wait_0)\ (f\ y)] \\
\equiv & \quad \{\ \text{definition of } concat\ \} \\
 & map\ (([] +\!\!+) \times wait_0)\ (f\ y) \\
\equiv & \quad \{\ map\ (([] +\!\!+) \times wait_0) \equiv map\ id \equiv id, \text{ def-id}\ \} \\
 & f\ y \\
\Box &
\end{array}
$$

$$(f^* \cdot g)^* = f^* \cdot g^*$$

4.2 Código

O código apresentado nesta secção está descrito na totalidade no ficheiro `AdventurersLog.hs`.

A mónada da lista de durações com registo está especificada da seguinte forma em HASKELL:

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

remLLD :: LogListDur t a -> [([t], Duration a)]
remLLD (LLD l) = l
```

```
instance Functor (LogListDur t) where
    fmap f = LLD . map f' . remLLD
        where f' = \(l, d) -> (l, fmap f d)


instance Applicative (LogListDur t) where
    pure = LLD . singl . split nil pure
    fs <*> l = LLD [ (l2 ++ l1, df <*> d )
        | (l1, df) <- remLLD fs, (l2, d) <- remLLD l ]



instance Monad (LogListDur t) where
    return = LLD . singl . split nil return
    l >>= f = LLD
        . concat
        . map (\(l', Duration (t,x)) ->
            map ((l' ++) >< (wait t)) (remLLD (f x)))
        . remLLD $ l

    -- Definicao alternativa
    -- l >>= f = LLD $ do
    --     (l1, Duration (t, x)) <- remLLD l
    --     let u = remLLD (f x) in
    --     map (\(l2, d) -> (l1 ++ l2, wait t d)) u
```

A resolução do exercício com esta mónada segue a mesma estratégia da anterior. Aliás, as funções existentes são exatamente as mesmas, sendo levemente modificadas para suportar esta nova mónada. Por esta razão não iremos descrever o código em detalhe.

O único acrescento que fizemos foi a estrutura de dados `Movement`, que representa uma travessia, e é definida da seguinte forma:

```
data Movement = Mov {
    from :: Bool,
    to :: Bool,
    adventurers :: [Adventurer],
    time :: Int
```

}

Onde `from` e `to` representam a origem e o destino da travessia, respetivamente; `adventurers` indica que aventureiros irão realizar a travessia; e `time` é o tempo que a travessia demora. É tipo de dados que são utilizados como registo na nova mónada.

Desta vez, ao tentarmos encontrar uma solução, podemos obter o seguinte resultado:

```
 ([False ---[P1,P2] t=2--> True,False <--[P1] t=1--- True,False
---[P5,P10] t=10--> True,False <--[P2] t=2--- True,False ---[P1,P2]
t=2--> True],Duration (17,["True","True","True","True","True"]))
```

Como podemos ver, é um resultado muito mais expressivo que o anterior, porque mostra-nos os movimentos que os aventureiros fizeram para chegar à solução.

5 UPPAAL vs HASKELL

Nesta secção fazemos uma análise comparativa acerca das duas abordagens seguidas para a modelação do sistema: UPPAAL e HASKELL.

No que toca à **criação** do modelo, tivemos mais facilidade em realizá-la em HASKELL. Não só porque já temos experiência prévia com a linguagem, mas também por ser precisamente uma linguagem de programação funcional, o que leva a que a modelação do sistema seja feita de uma forma mais "sequencial". Ao contrário do UPPAAL em que temos de pensar em vários processos ao mesmo tempo e na forma que eles interagem entre si. É claro que podíamos ter criado um modelo apenas com um processo, nas isso provavelmente teria uma grande influência no autómato resultante. O que nos leva ao segundo ponto da nossa comparação: **visualização**.

A nível de visualização do modelo, achamos indubitavelmente que o UPPAAL é superior. O facto de utilizarmos autómatos como estrutura da nossa modelação permite-nos uma compreensão simples e direta do comportamento do sistema. Já no HASKELL esta compreensão é feita através da análise das diferentes funções e estruturas que tenhamos. Adicionalmente, a funcionalidade de simulação do UPPAAL fornece uma grande ajuda no caso de precisarmos de resolver quaisquer problemas que surjam no nosso modelo. Enquanto que no HASKELL, esta resolução é feita com recurso a testes.

Por fim, temos a **verificação** do modelo. Ambas as abordagens seguem uma lógica CTL, o UPPAAL explicitamente, o HASKELL implicitamente através do uso de *bruteforce*, por isso ambas as verificações vão consistir em propriedades com formulações semelhantes. Uma das principais diferenças é o facto de o UPPAAL suportar com comportamento contínuo, o que leva à necessidade de provar propriedades adicionais (ex.: *zeno*) para termos a certeza que o modelo funciona corretamente. Já o HASKELL descreve apenas comportamentos discretos, não havendo necessidade de provar estas propriedades. Uma outra diferença que notamos foi a nível de implementação de restrições. Enquanto que no UPPAAL temos de criar um modelo que obedeça a certas restrições, havendo muitas vezes a necessidade de prova destas mesmas no fim, no HASKELL incluímos estas restrições no desenvolvimento do modelo, não havendo necessidade de as verificar porque sabemos que o modelo está correto por construção.

6 Conclusão

Dámos assim por concluído este segundo trabalho prático. De um modo geral estamos quase totalmente satisfeitos com o resultado deste, porque conseguimos fazer tudo o que foi pedido, e foi uma forma de conseguirmos aplicar os conteúdos que aprendemos nesta fase final da UC, como a criação de mónadas. No entanto não conseguimos provar uma das propriedades das mónadas, tanto na lista de durações como na lista de durações com registos. Implementamos uma forma diferente de modelação, o que nos deu uma base de comparação para com a abordagem seguida no primeiro trabalho prático. No geral, vemos vantagens e desvantagens tanto numa como outra, sendo que, na nossa opinião, tudo se resume a uma questão de gosto.