Modelação e análise de um sistema ciber-físico com Monads

Carlos Ferreira

Henrique Neto

Primeira Parte

Instância do Monad ListDur

- η – devolve um *singleton* contendo um objeto "Duration" com o valor $\eta(valor)$ (que neste caso devolve um Duration com o valor de duração nulo)
- k^* – para cada objeto x to tipo "Duration" da lista dada, compõem x com todos os elementos y provenientes da lista $k(x)$.

```
instance Applicative ListDur where
   pure x = LD [return x]
   l1 <*> l2 =  LD [f <*> x | f <- remLD l1, x <- remLD l2 ]

instance Monad ListDur where
   return = pure
   l >>= k =  LD $ do x <- remLD l
                      map (\d -> x >>= const d ) (remLD (k (getValue x)))
```

Primeira Parte

Utilidades do Monad ListDur

- *isEmpty* – predicado que indica se a coleção está vazia;
- filterDuration - filtra os elementos da coleção com base num predicado dado;
- anyDuration - indica se existe um elemento da coleção que satisfaz o predicado dado;
- allDuration - indica se todos os elementos da coleção satisfazem o predicado dado;
- composeDurations - compõem cada elemento da coleção com a função dada;

```
isEmpty :: ListDur a -> Bool
filterDuration :: (Duration a -> Bool) -> ListDur a -> ListDur a
anyDuration    :: (Duration a -> Bool) -> ListDur a -> Bool
allDuration    :: (Duration a -> Bool) -> ListDur a -> Bool

composeDurations :: (a -> Duration  b) -> ListDur a -> ListDur b
composeDurations f = LD . map (>>= f) . remLD
```

Primeira Parte

Modelação do Problema - Básico

Aventureiro

- allAdventurers
- getTimeAdv

Objects

- lantern
- allAdventurersObjects
- allObjects
- getTimeObject

State

- gInit e gEnd
- changeState
- mChangeState

```
data Adventurer = P1 | P2 | P5 | P10 deriving (Show, Eq, Ord)

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

type Objects = Either Adventurer ()

lantern :: Objects
lantern = Right ()

getTimeObject :: Objects -> Int
getTimeObject = either getTimeAdv (const 0)

type State = Objects -> Bool
```

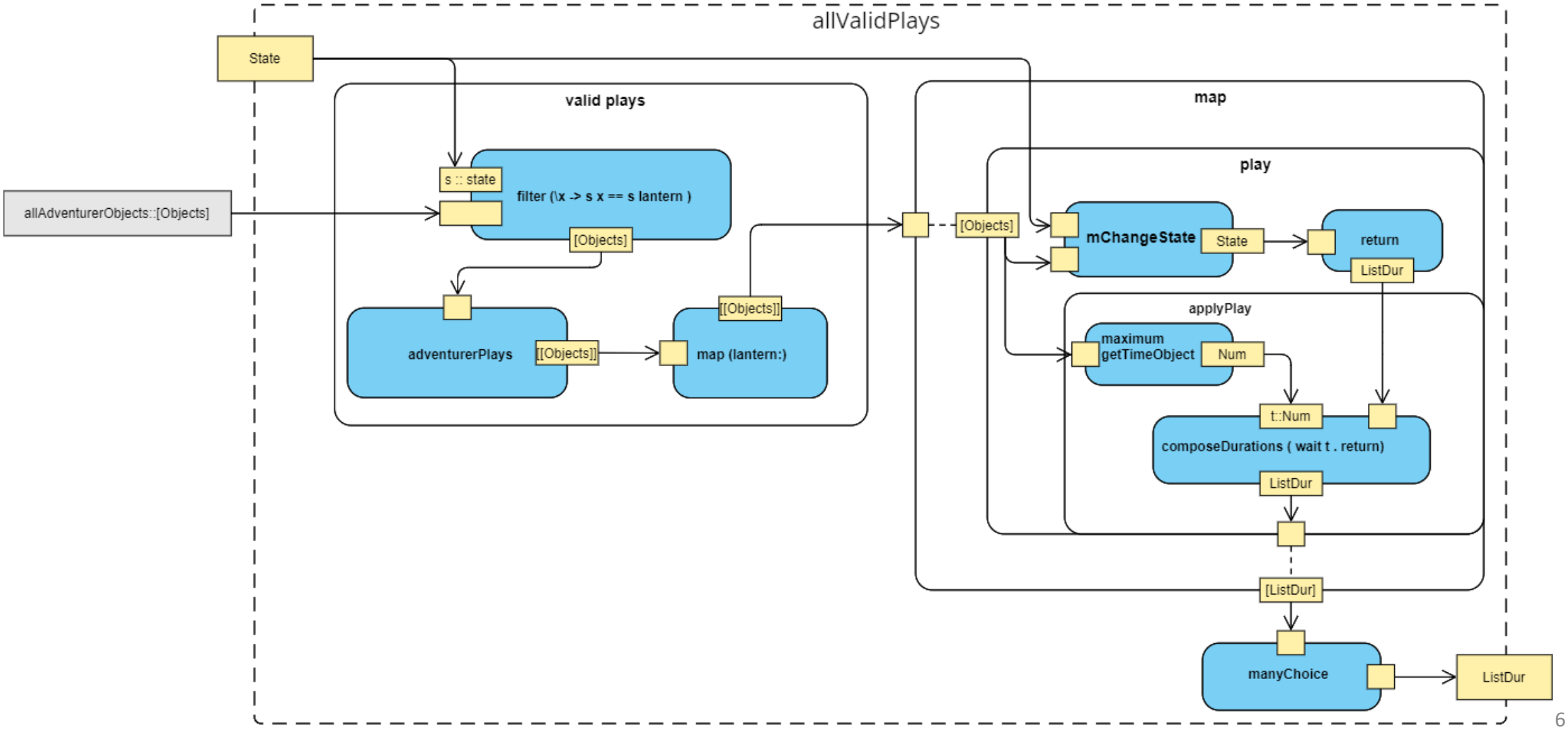
Primeira Parte

Modelação do problema - Transações

1. Filtrar Aventureiros
2. Combinar Aventureiros
3. Aplicar Transações de estados
4. Aplicar passagens de tempo
5. Agregar estados

Primeira Parte

Modelação do problema -Transações



Primeira Parte

Modelação do problema -Transações

- AllValidPlays permite calcular os estados alcançáveis a partir de um estado inicial numa única travessia.
- Para estender a mais travessias usa-se as funcionalidades de uma monad para definir exec.

```
exec :: Int -> State -> ListDur State
exec n s= if n <= 0
          then return s
          else do newSs <- allValidPlays s
                  exec (n - 1) newSs
```

Primeira Parte

Analise do Modelo – Estratégia 1

- Utiliza a função *exec* definida anteriormente.
- Permite procurar uma solução num intervalo de iterações, provando facilmente uma possibilidade.

```
endedLeq :: Int -> ListDur State -> Bool
endedLeq i = anyDuration (\x -> (getDuration x <= i) && (getValue x == gEnd))

leq17 :: Bool
leq17 =  any (endedLeq 17 . \x -> exec x gInit) [1..5]
```

- No entanto, torna difícil provar impossibilidade.

Primeira Parte

Analise do Modelo – Estratégia 2

- Nova definição de *exec, que a cada iteração realiza cálculos com o explicito objetivo de encontrar uma dada situação.*
- *Termina se encontrar a solução ou descobrir que tal é impossível*

```
l17 :: Bool
l17 =  execd (gEnd ==) (\d -> 17 > getDuration d) (return gInit)

execd :: (State -> Bool) -> (Duration State -> Bool) -> ListDur State -> Bool
execd f g ls = let exec = filterDuration g (ls >>= allValidPlays)
               in not(isEmpty ls) && (anyDuration (f . getValue) exec || execd f g exec)
```

Segunda Parte

UPPAAL vs Haskell

- Aprendizagem
- Foco
- Implementação
- Escalabilidade
- Analise e verificação
- Rapidez de implementação
- Interpretação

Segunda Parte

Analise do Modelo – Estratégia 2

	UPPAAL	*Haskell*
Facilidade de aprendizagem	Melhor	Pior
Fatores Físicos do problema	Temporal	Implementável
Escalabilidade	Problemática	Sistemática
Velocidade de implementação.	Rápida	Depende da Experiência
Debbug e deteção de erros	Trivial	Implementável
Foco em segurança e correção	Própria	Implementável
Sistemas Distribuídos	Própria	Implementável
Transações entre Estados	Explicitas	Sistemáticas
Interpretação do modelo	Visual	Dedutível
Poder expressivo de analise	Restrito e Fácil	Implementável

Extra

Adição de *Logs* na Mónada

```
data ListDur a = LD [(String, Duration a)]
```

- η – o *singleton* agora é composto por um par com uma string nula e a Duration com valor $\eta(valor)$.
- k^* – para além de compor as durações, este agora também concatena as strings dos elementos

```
return x = LD [([], return x)]
l >>= k =  LD $ do sx <- remLD l
                   g sx
                   where g (s,x) = map ((s++) >< (\d -> x >>= const d))
                                       (remLD (k (getValue x)))
```

Extra

Mudanças nas utilidades nas Mónada

Todos os tipos das funções anteriores manterem-se, havendo unicamente a necessidade de alterar as suas implementações.

- Para as funções com predicados (filterDuration, anyDuration, allDuration) isto consistia em fazer *snd* antes do respetivo teste.
- Para a função (composeDurations) isto consistiu antes em adicionar $id \times$ ao mapeamento

```
composeDurations f = LD . map (id >< (>>= f)) . remLD
```

- Adicionalmente foi criada uma função *logLD* que permita adicionar conteúdo aos Logs presentes numa lista

```
logLD:: String -> ListDur a -> ListDur a
logLD s = LD . map ((++s) >< id) . remLD
```

Extra

Adições ao modelo do problema

- A mónada implementada é compatível com o código original, mas este não tiraria partido dos logs
- Para contabilizamos esta utilidade foi adicionado uma chamada a *logLD* no método applyPlay

```
applyPlay :: [Objects] -> ListDur a -> ListDur a
applyPlay os = let t = maximum (map getTimeObject os)
                   s = playFormat os
                   in logLD s . composeDurations (wait t . return)
```

- Onde *playFormat* consiste na função que converte uma jogada numa string

```
playFormat :: [Objects] -> String
playFormat = ('>':) . concatMap (either ((' ':) . show) (const ""))
```

Modelação e análise de um sistema ciber-físico com Monads

Carlos Ferreira

Henrique Neto