Modelling and Analysis of a Cyber-Physical System with Monads

Cyber-Physical Programming — Practical Assignment 2

Melânia Pereira   Paulo R. Pereira

University of Minho

Table of contents

The *ListDur* and *ListLogDur* monads

ListDur

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

instance Functor ListDur where
fmap f = LD . (map (fmap f)) . remLD
```

ListDur

```haskell
instance Monad ListDur where
return = pure
l >>= k = LD $ do
  x <- remLD l
  g x k where
    g :: Duration t -> (t -> ListDur a) -> [Duration a]
    g (Duration (d, a)) k =
      map (\(Duration (d', a)) ->
      (Duration (d + d', a))) (remLD (k a))
```

ListLogDur

```
data ListLogDur a = LSD [Duration (String, a)] deriving Show

instance Functor ListLogDur where
    fmap f = LSD . (map (fmap (id >< f))) . remLSD
```

ListLogDur

```
instance Monad ListLogDur where
    return = pure
    l >>= k =
      let k' = LD . remLSD . k
      l' = (LD . remLSD) l in
      LSD $ remLD (l' >>= (auxLSDMonad k')) where
        auxLSDMonad k (s, x) =
        (LD . map (Duration . (id >< ((++ s) >< id)) . remDur) .
        ↪  remLD . k) x
        remDur (Duration a) = a
```

Modelling the problem

Modelling the problem

1. Which adventurers can cross?

```haskell
advsWhereLanternIs :: State -> [Adventurer]
advsWhereLanternIs s =
    filter ((== s lantern) . s . Left) [P1, P2, P5, P10]
```

Modelling the problem

2. How the adventurers can cross?

```
combinationsUpTo2 :: Eq a => [a] -> [[a]]
combinationsUpTo2 = conc . (split f g) where
      f t = do {x <- t; return [x]}
      g t = do {x <- t; y <- (remove x t); return [x, y]}
      remove x [] = []
      remove x (h:t) = if x==h then t else remove x t
```

Modelling the problem

3. How long do the different groups take?

```haskell
addTime :: [Adventurer] -> (Int, [Adventurer])
addTime = split (maximum . (map getTimeAdv)) id
```

Modelling the problem

4. No one can cross without the lantern!

```haskell
addLantern :: (Int, [Adventurer]) -> (Int, [Object])
addLantern = id >< ((lantern :) . map Left)
```

Modelling the problem

5. Finally, all valid plays for a given state!

```
allValidPlays :: State -> ListLogDur State
allValidPlays s = LSD $ map Duration $ map (id >< (split (toTrace
↪  s) id) . (mCS s)) t where
  t = (map (addLantern . addTime) . combinationsUpTo2 .
   ↪  advsWhereLanternIs) s
  mCS = flip mChangeState
  toTrace s s' = printTrace (state2List s, state2List s')
```

The trace log

The trace log

As we saw, we keep the trace by calling the function $\mathit{toTrace} :: \mathit{State} \rightarrow \mathit{State} \rightarrow \mathit{String}$.

But what does it do?

The trace log

According to the representation of the state, adventurers can be represented by indexes.

```
index2Adv :: Int -> String
index2Adv 0 = "P1"
index2Adv 1 = "P2"
index2Adv 2 = "P5"
index2Adv 3 = "P10"
```

We take advantage of this to be able to present an elegant trace of the moves.

The trace log

Example

Given both states:

```
st = gInit

st' (Left P1) = True
st' (Left P2) = True
st' (Left P5) = False
st' (Left P10) = False
st' lantern = True
```

Goal: represent them in lists and compare element to element storing the index if they are different.

The trace log

Example

```
stL = [False, False, False, False, False]

stL' = [True, True, False, False, True]

-- result: [0,1,4] means that "P1 and P2 crosses"
```

N.B.: Index 4 represents the lantern, and because we assume that the movements are always valid, we can ignore that.

The trace log

Getting things pretty!

```
printTrace :: ([Bool], [Bool]) -> String
printTrace = prettyLog . (map index2Adv) . init .
↪   indexesWithDifferentValues

prettyLog :: [String] -> String
prettyLog = Cp.cond ((>1) . length) f ((++ " cross\n") . head)
↪   where
    f = (++" crosses\n") . conc . ((concat . map (++" and ")) ><
    ↪   id) . (split init last)
```

The trace log

Back to function *allValidPlays*:

for a given state s and each following state s',

```
toTrace s s' = printTrace (state2List s, state2List s')
```

The trace log

Example (optimal play)

```
> optimalTrace
P1 and P2 crosses
P1 cross
P5 and P10 crosses
P2 cross
P1 and P2 crosses

OR

P1 and P2 crosses
P2 cross
P5 and P10 crosses
P1 cross
P1 and P2 crosses

in 17 minutes.
```

Solving the problem

exec

```
execPred :: (State -> Bool) -> State -> (Int, ListLogDur State)
execPred p s = aux p s 0 where
               aux p s it = let st = exec it s
                                res = filter pred (map remDur
                                ↪   (remLSD st)) in
                                if length (res) > 0 then ((it+1) ,
                                ↪   LSD (map Duration res))
                                else aux p s (it+1) where
                                  remDur (Duration a) = a
                                  pred (_, (_,s)) = p s
```

Answering the questions!

Is it possible for all adventurers to be on the other side in ≤ 17 minutes and not exceeding 5 moves?

```
leqX :: Int -> (Int, Bool)
leqX n = if res then (it,res)
                else (0,res) where
                  res = length (filter p (map remDur (remLSD l)))
                   ↪  > 0
                  (it,l) = execPred (== gEnd) gInit
                  p (d,(_,_)) = d <= n
                  remDur (Duration a) = a

leq17 :: Bool
leq17 = p2 (leqX 17) && p1 (leqX 17) <= 5
```

Answering the questions!

Is it possible for all adventurers to be on the other side in < 17 minutes?

```haskell
lX :: Int -> (Int, Bool)
lX n = if res then (it,res)
             else (0,res) where
               res = length (filter p (map remDur (remLSD l))) >
               ↪  0
               (it,l) = execPred (== gEnd) gInit
               p (d,(_,_)) = d < n
               remDur (Duration a) = a

l17 :: Bool
l17 = p2 (lX 17)
```

Comparative Analysis and Final Comments

Scalability

- In the HASKELL approach, running the *exec* function for a very large *n* slows down the program! Runtime is exponential!
- UPPAAL model checking engine allows efficient and fast timed automata model exploration. So, for expensive executions, UPPAAL may be a better option.

System's security

- Modelling in UPPAAL can be easier because of invariants that can be easily defined.
- The HASKELL approach requires the designer to correctly implement the functions to correctly model the problem, fulfilling pre and post conditions and system invariants. Therefore, it is more susceptible to errors.

Clocks vs Duration Monad

- In UPPAAL is that clocks are logical concepts. The simulator does not allow seeing clock values (only the satisfaction can be checked via constraints). This works if we just want to answer the problem's questions. However, it would be nice to get the time associated to some particular execution (as HASKELL does).
- In every state of every possible sequence of moves, monad gives us the "clock value", i.e. the duration of the moves so far.

Space of solutions

- Reachability queries in UPPAAL doesn’t explore the whole state space and stops when the given expression becomes true. UPPAAL does not give us all the solutions — only one, if possible.
- HASKELL gives us the entire space of solutions!

Getting the trace (or path)

- The fact that UPPAAL produces counterexamples allows us to automatically get the trace (or path) — e.g., only need to negate the property *leq17* in order to obtain the trace log for an optimal solution.
- In HASKELL, there was a need to incorporate the path in the monad definition. Of course, for someone who is modelling with monads in HASKELL, that should not be a problem at all.

Questions?