Modelling and Analysis of a Cyber-Physical System with Monads

Cyber-Physical Programming — Practical Assignment 2

Melânia Pereira Paulo R. Pereira
{pg47520, pg47554}@alunos.uminho.pt

June 20, 2022

Abstract

This assignment aims to model and to analyze a system with a powerful weapon of functional programming — monads! The system to model envolves 4 adventurers, one lantern and a bridge! This 4 adventurers need to cross the bridge, but, for safety reasons, only two people can cross at the same time and one of them needs to carry the lantern, also, each adventurer takes a different time than the others to cross the bridge. The first task of the assignment is to model this system using HASKELL and to verify some claims made by the adventurers, namely, that they can be all on the other side in 17 minutes, and also to show that it is impossible for them all to be on the other side in less then 17 minutes. This system could be modeled using different approaches and modules, as it was done in classes via UPPAAL, so the second task of the assignment focuses on the comparison of both UPPAAL and HASKELL approaches.

1 The Adventurers' Problem

In the middle of the night, four adventurers encounter a shabby rope-bridge spanning a deep ravine. For safety reasons, they decide that no more than 2 people should cross the bridge at the same time and that a flashlight needs to be carried by one of them in every crossing. They have only one flashlight. The 4 adventurers are not equally skilled: crossing the bridge takes them 1, 2, 5, and 10 minutes, respectively. A pair of adventurers crosses the bridge in an amount of time equal to that of the slowest of the two adventurers.

One of the adventurers claims that they cannot be all on the other side in less than 19 minutes. One companion disagrees and claims that it can be done in 17 minutes.

Who is right? That's what we're going to find out.

2 Monadic Approach via HASKELL for Modelling the Problem

The solution is to take advantage of the non-deterministic monad (monad List) to use brute force and calculate all possible moves until we reach the final state. To deal with the time adventurers need to cross, we'll use the duration monad (already implemented by prof. Renato Neves) whose implementation adds the time each adventurer takes in a given move. This duration monad will be "encapsulated" in our final monad $ListLogDur$. This one will offer, for a certain state, a list of following states with the respective duration needed to get it and will also offer the path traveled at the moment. This path will be in a *string* and, as we'll see, it is going to be very elegant. For now, let's analyze the construction of our monad!

2.1 The *ListLogDur* monad

As said before, we'll use our monad to have a list of states with the respective duration needed to get it and the path traveled at the moment. However, we want our monad to be parametric. So,

$$\textbf{data}\ \mathit{ListLogDur}\ a = \mathit{LSD}\ [\mathit{Duration}\ (\mathit{String}, a)]\ \textbf{deriving}\ \mathit{Show}$$

We now have our set-constructor, so we need to define the η function and the $(-)^\star$ operator. To define the η function, when need to understand what means a effect-free representation in this monad — it means that we have no duration and an empty path traveled. So,

$$\begin{aligned}&\eta : X \to \mathit{ListLogDur}\ X\\&\eta\ x = \mathit{LSD}\ [\mathit{Duration}\ (0, ([\,], x))]\end{aligned}$$

For the $(-)^\star$ operator, assuming a function $f : X \to \mathit{ListLogDur}\ Y$, we'll have to define the function, $f^\star : X \to \mathit{ListLogDur}\ Y$, which in HASKELL corresponds to the $(\gg\!=)$ operation. This implementation follows along with the **instance** *Functor* and the **instance** *Applicative* (required by HASKELL itself).

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ \mathit{ListLogDur}\ \textbf{where}\\&\quad \mathsf{fmap}\ f = \mathit{LSD} \cdot (\mathsf{map}\ \ (\mathsf{fmap}\ (\mathit{id} \times f))) \cdot \mathit{remLSD}\\&\textbf{instance}\ \mathit{Applicative}\ \mathit{ListLogDur}\ \textbf{where}\\&\quad \mathit{pure} = \mathit{LSD} \cdot \mathit{pure} \cdot \mathit{pure} \cdot (\lambda x \to ([\,], x))\\&\quad l_1 <\!*\!> l_2 = \mathit{LSD}\ \$\ \textbf{do}\\&\qquad x \leftarrow \mathit{remLSD}\ l_1\\&\qquad y \leftarrow \mathit{remLSD}\ l_2\\&\qquad g\ (x, y)\ \textbf{where}\\&\qquad\quad g\ (\mathit{Duration}\ (d_1, (s, f)), \mathit{Duration}\ (d_2, (s', x))) = \mathit{return}\ (\mathit{Duration}\ (d_1 + d_2, (s +\!\!+ s', f\ x)))\\&\textbf{instance}\ \mathit{Monad}\ \mathit{ListLogDur}\ \textbf{where}\\&\quad \mathit{return} = \mathit{pure}\\&\quad l \gg\!= k = \mathit{LSD}\ \$\ \textbf{do}\\&\qquad x \leftarrow \mathit{remLSD}\ l\\&\qquad g\ x\ k\ \textbf{where}\\&\qquad\quad g\ (\mathit{Duration}\ (d, (s, a)))\ k =\\&\qquad\qquad \mathsf{map}\ \ (\lambda(\mathit{Duration}\ (d', (s', a))) \to (\mathit{Duration}\ (d + d', (s +\!\!+ s', a))))\ (\mathit{remLSD}\ (k\ a))\\&\mathit{remLSD} :: \mathit{ListLogDur}\ a \to [\mathit{Duration}\ (\mathit{String}, a)]\\&\mathit{remLSD}\ (\mathit{LSD}\ x) = x\end{aligned}$$

2.2 Modelling the problem

Adventurers are represented by the following data type:

$$\textbf{data}\ \mathit{Adventurer} = P_1 \mid P_2 \mid P_5 \mid P_{10}\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})$$

Lantern is represented by the $()$ element, so we can represent all the entities by using the coproduct and defining the following data type:

$$\begin{aligned}&\textbf{type}\ \mathit{Object} = \mathit{Adventurer} + ()\\&\mathit{lantern} = i_2\ ()\end{aligned}$$

The names for the adventurers are quite suggestive as they are identified by the time they take to cross. However, it will be very useful to have a function that returns, for each adventurer, the time it takes to cross the bridge.

$$
\begin{aligned}
&\mathit{getTimeAdv} :: \mathit{Adventurer} \to \mathit{Int}\\
&\mathit{getTimeAdv}\ P_1 = 1\\
&\mathit{getTimeAdv}\ P_2 = 2\\
&\mathit{getTimeAdv}\ P_5 = 5\\
&\mathit{getTimeAdv}\ P_{10} = 10
\end{aligned}
$$

Now, we need to define the state of the game, i.e. the current position of each object (adventurers + the lantern). The function *False* represents the initial state of the game, with all adventurers and the lantern on the left side of the bridge. Similarly, the function *True* represents the end state of the game, with all adventurers and the lantern on the right side of the bridge. We also need to define the instances Show and Eq to visualize and compare, respectively, the states of the game.

$$
\begin{aligned}
&\textbf{type}\ \mathit{State} = \mathit{Object} \to \mathit{Bool}\\
&\textbf{instance}\ \mathit{Show}\ \mathit{State}\ \textbf{where}\\
&\quad \mathit{show}\ s = \mathit{show} \cdot \mathit{show}\ \$\ [s\ (i_1\ P_1),\\
&\qquad s\ (i_1\ P_2),\\
&\qquad s\ (i_1\ P_5),\\
&\qquad s\ (i_1\ P_{10}),\\
&\qquad s\ (i_2\ ())]\\
&\textbf{instance}\ \mathit{Eq}\ \mathit{State}\ \textbf{where}\\
&\quad (\equiv)\ \mathit{s1}\ \mathit{s2} = \mathit{and}\ [\mathit{s1}\ (i_1\ P_1) \equiv \mathit{s2}\ (i_1\ P_1),\\
&\qquad \mathit{s1}\ (i_1\ P_2) \equiv \mathit{s2}\ (i_1\ P_2),\\
&\qquad \mathit{s1}\ (i_1\ P_5) \equiv \mathit{s2}\ (i_1\ P_5),\\
&\qquad \mathit{s1}\ (i_1\ P_{10}) \equiv \mathit{s2}\ (i_1\ P_{10}),\\
&\qquad \mathit{s1}\ (i_2\ ()) \equiv \mathit{s2}\ (i_2\ ())]\\
&\mathit{gInit} :: \mathit{State}\\
&\mathit{gInit} = \underline{\mathit{False}}\\
&\mathit{gEnd} :: \mathit{State}\\
&\mathit{gEnd} = \underline{\mathit{True}}\\
&\mathit{state2List} :: \mathit{State} \to [\mathit{Bool}]\\
&\mathit{state2List}\ s = [s\ (i_1\ P_1),\\
&\quad s\ (i_1\ P_2),\\
&\quad s\ (i_1\ P_5),\\
&\quad s\ (i_1\ P_{10}),\\
&\quad s\ (i_2\ ())]
\end{aligned}
$$

Obviously, it is useful a function that changes the state of the game for a given object:

$$
\begin{aligned}
&\mathit{changeState} :: \mathit{Object} \to \mathit{State} \to \mathit{State}\\
&\mathit{changeState}\ a\ s = \textbf{let}\ v = s\ a\ \textbf{in}\ (\lambda x \to \textbf{if}\ x \equiv a\ \textbf{then}\ \neg\ v\ \textbf{else}\ s\ x)
\end{aligned}
$$

Even more useful is a function that changes the state of the game of a list of objects:

$$
\begin{aligned}
&\mathit{mChangeState} :: [\mathit{Object}] \to \mathit{State} \to \mathit{State}\\
&\mathit{mChangeState}\ \mathit{os}\ s = \mathit{foldr}\ \mathit{changeState}\ s\ \mathit{os}
\end{aligned}
$$

With this, we are now ready to define all the valids plays the adventurers can make for a given state of the game, storing, obviously, the respective duration required and the move made. So, for a given $s :: \mathit{State}$, we'll compute $\mathit{allValidPlays} :: \mathit{ListLogDur}\ \mathit{State} \sim \mathit{LSD}\ [\mathit{Duration}\ (\mathit{String}, \mathit{State})]$. For that, let's think:

1. We need to move adventurers — but only adventurers who can pick up the lantern. So, for that given state, we first need to calculate the adventurers who are where the lantern is.

$$advsWhereLanternIs :: State \to [Adventurer]$$
$$advsWhereLanternIs\ s = filter\ ((\equiv s\ lantern) \cdot s \cdot i_1)\ [P_1, P_2, P_5, P_{10}]$$

2. Now, since we got the adventurers who can cross, we need to group them into all possible combinations. As we know, a maximum of 2 adventurers can cross. This parametric function

$$combinationsUpTo2 :: Eq\ a \Rightarrow [a] \to [[a]]$$
$$combinationsUpTo2 = \mathsf{conc} \cdot \langle f, g \rangle\ \mathbf{where}$$
$$f\ t = \mathbf{do}\ \{x \leftarrow t; return\ [x]\}$$
$$g\ t = \mathbf{do}\ \{x \leftarrow t; y \leftarrow (remove\ x\ t); return\ [x, y]\}$$
$$remove\ x\ [\,] = [\,]$$
$$remove\ x\ (h : t) = \mathbf{if}\ x \equiv h\ \mathbf{then}\ t\ \mathbf{else}\ remove\ x\ t$$

   applied to the list of all possivle adventurers will return all possible groups in sublists.

3. Now, we have to get the time both group need to cross — we just need to map the function $getTimeAdv$ and return the maximum value. We may also produce the pair with this result and the initial list of adventurers.

$$addTime :: [Adventurer] \to (Int, [Adventurer])$$
$$addTime = \langle maximum \cdot (\mathsf{map}\ \ getTimeAdv), id \rangle$$

4. We also need to add the lantern to the group that is going to cross — they need the lantern to cross. This returns the list of objects that are going to cross and the time needed to do it.

$$addLantern :: (Int, [Adventurer]) \to (Int, [Object])$$
$$addLantern = id \times ((lantern:) \cdot \mathsf{map}\ \ i_1)$$

5. Finally, we need to use the function $\mathsf{map}\ \ mChangeState$ to change the state of that list of lists of objects (which are our possible moves) and encapsulate it in the monad using the composition $LSD \cdot \mathsf{map}\ \ Duration$. Yes, we are missing something — the path (or the trace)!!! For now, let's just appreciate the final function. The next subsection will explain how we get the path!

$$allValidPlays :: State \to ListLogDur\ State$$
$$allValidPlays\ s = LSD\ \$\ \mathsf{map}\ \ Duration\ \$\ \mathsf{map}\ \ (id \times \langle toTrace\ s, id \rangle \cdot (mCS\ s))\ t\ \mathbf{where}$$
$$t = (\mathsf{map}\ \ (addLantern \cdot addTime) \cdot combinationsUpTo2 \cdot advsWhereLanternIs)\ s$$
$$mCS = flip\ mChangeState$$
$$toTrace\ s\ s' = printTrace\ (state2List\ s, state2List\ s')$$

2.3 The trace log

As we saw, our monad $ListLogDur$ keeps the trace by calling the function $toTrace :: State \to State \to String$. But what does it do?

First, we can see that, according to the representation of the state, adventurers can be represented by indexes. We take advantage of this to be able to present an elegant trace of the moves. For example, if the previous state is $[False, False, False, False, False]$ and the current state is $[True, True, False, False, True]$, we know that P_1 and P_2 have crossed (because the first two and the last elements and diferent). So, we can simply compare element to element and, if they are different, we keep the index. In the previous example, it would return $[0, 1, 4]$ — index 4 represents the lantern, and because we assume that the movements are always valid, we can ignore that.

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

indexesWithDifferentValues :: Eq a ⇒ ([a],[a]) → [Int]
indexesWithDifferentValues (l1,l2) = aux l1 l2 0 where
  aux :: Eq a ⇒ [a] → [a] → Int → [Int]
  aux [] l _ = []
  aux l [] _ = []
  aux (h1 : t1) (h2 : t2) index = if h1 ≢ h2 then index : aux t1 t2 (index + 1)
    else aux t1 t2 (index + 1)
```

The result $[0, 1, 4]$ means that "P_1 and P_2 crosses". We now have to automate this (pretty) print. We only need to ignore the lantern index (4), convert the indexes to the respective adventurers and define a print function for them.

```
printTrace :: ([Bool],[Bool]) → String
printTrace = prettyLog · (map index2Adv) · init · indexesWithDifferentValues

prettyLog :: [String] → String
prettyLog = (>1) · length → f , (++" cross\n") · head where
  f = (++" crosses\n") · conc · ((concat · map (++" and ")) × id) · ⟨init, last⟩
```

Let's see the result of applying the function *printTrace* with the previous example.

```
> t = ([False,False,False,False,False],[True,True,False,False,True])
> printTrace t
"P1 and P2 crosses\n"
```

Finnaly, using the function *putStr*, we get a pretty nice log:

```
> putStr $ printTrace t
P1 and P2 crosses
```

Back to function *allValidPlays*, we do, for a given state s and each following state s',

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

So, this representation is done right in the calculation of the possible moves. At the end, we just need to get that already prepared trace. In the next subsection, we'll see the trace of the optimal play which shows how elegant the log is.

2.4 Solving the problem

First, we defined a function that, for a given number n and an initial state, calculates all possible n-sequences of moves that the adventures can make. For that, we took advantage of the **do** notation — let the monad do the work for us!

```
exec :: Int → State → ListLogDur State
exec 0 s = allValidPlays s
exec n s = do ps ← exec (n − 1) s
  allValidPlays ps
```

The previous functions is nice, but not so much — we don't know how many sequences are needed to reach the end state. It would be much better if we could execute all possible sequences of moves that the adventures can make for a given state untill it fulfills a predicate over a state (passed as a parameter). Additionally, it also returns the number of moves needed to fulfill that predicate.

$$
\begin{aligned}
&execPred :: (State \to Bool) \to State \to (Int, ListLogDur\ State)\\
&execPred\ p\ s = aux\ p\ s\ 0\ \textbf{where}\\
&\quad aux\ p\ s\ it = \textbf{let}\ st = exec\ it\ s\\
&\qquad res = filter\ pred\ (\mathsf{map}\ \ remDur\ (remLSD\ st))\ \textbf{in}\\
&\qquad \textbf{if}\ length\ (res) > 0\ \textbf{then}\ ((it+1), LSD\ (\mathsf{map}\ \ Duration\ res))\\
&\qquad \textbf{else}\ aux\ p\ s\ (it+1)\ \textbf{where}\\
&\qquad\quad remDur\ (Duration\ a) = a\\
&\qquad\quad pred\ (\_, (\_, s)) = p\ s
\end{aligned}
$$

We may use this version to solve the problem and see who's right. For that, 2 more functions were defined to see if it is possible for all adventurers to be on the other side in $\leqslant n$ (and $<n$) minutes and how many moves are needed for that.

$$
\begin{aligned}
&leqX :: Int \to (Int, Bool)\\
&leqX\ n = \textbf{if}\ res\ \textbf{then}\ (it, res)\\
&\qquad \textbf{else}\ (0, res)\ \textbf{where}\\
&\qquad\quad res = length\ (filter\ p\ (\mathsf{map}\ \ remDur\ (remLSD\ l))) > 0\\
&\qquad\quad (it, l) = execPred\ (\equiv gEnd)\ gInit\\
&\qquad\quad p\ (d, (\_, \_)) = d \leqslant n\\
&\qquad\quad remDur\ (Duration\ a) = a\\
&lX :: Int \to (Int, Bool)\\
&lX\ n = \textbf{if}\ res\ \textbf{then}\ (it, res)\\
&\quad \textbf{else}\ (0, res)\ \textbf{where}\\
&\qquad res = length\ (filter\ p\ (\mathsf{map}\ \ remDur\ (remLSD\ l))) > 0\\
&\qquad (it, l) = execPred\ (\equiv gEnd)\ gInit\\
&\qquad p\ (d, (\_, \_)) = d < n\\
&\qquad remDur\ (Duration\ a) = a
\end{aligned}
$$

So let's see who was right!

- *Is it possible for all adventurers to be on the other side in $\leqslant 17$ minutes and not exceeding 5 moves?*

$$
\begin{aligned}
&leq17 :: Bool\\
&leq17 = \pi_2\ (leqX\ 17) \wedge \pi_1\ (leqX\ 17) \leqslant 5
\end{aligned}
$$

```
> leq17
True
```

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

$$
\begin{aligned}
&l17 :: Bool\\
&l17 = \pi_2\ (lX\ 17)
\end{aligned}
$$

```
> l17
False
```

As we saw, it is possible for all adventurers to be on the other side in $\leqslant 17$ minutes and not exceeding 5 moves. Actually, since *l17* returns *False*, 17 is the the optimal time for solving the problem (with exactly 5 moves). One could also get that information by executing the following function *optimalTrace*, which shows how beautiful our trace log is!

$$
\begin{aligned}
&\mathit{optimalTrace} :: \mathsf{IO}\ () \\
&\mathit{optimalTrace} = \\
&\quad \mathit{putStrLn} \cdot t \cdot \mathsf{map}\ \ \mathit{remDur} \cdot \mathit{remLSD} \cdot \pi_2\ \$\ \mathit{execPred}\ (\equiv \mathit{gEnd})\ \mathit{gInit}\ \mathbf{where} \\
&\quad\quad t = \mathit{prt} \cdot \langle \mathit{head} \cdot \mathsf{map}\ \ \pi_1, \mathsf{map}\ \ (\pi_1 \cdot \pi_2) \rangle \cdot \mathit{pairFilter} \cdot \langle \mathit{minimum} \cdot \mathsf{map}\ \ \pi_1, \mathit{id} \rangle \\
&\quad\quad \mathit{remDur}\ (\mathit{Duration}\ a) = a \\
&\quad\quad \mathit{pairFilter}\ (d, l) = \mathit{filter}\ (\lambda(d', (\_, \_)) \rightarrow d \equiv d')\ l \\
&\quad\quad p = (>1) \cdot \mathit{length} \rightarrow p'\ ,\ \mathit{head} \\
&\quad\quad p' = \mathsf{conc} \cdot \langle \mathit{concat} \cdot \mathsf{map}\ \ ((+\!\!+(\texttt{"\textbackslash nOR\textbackslash n\textbackslash n"}))) \cdot \mathit{init}, \mathit{last} \rangle \\
&\quad\quad \mathit{prt}\ (d, l) = (p\ l) +\!\!+ \texttt{"\textbackslash nin "} +\!\!+ (\mathit{show}\ d) +\!\!+ \texttt{" minutes."}
\end{aligned}
$$

```
> 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.
```

3 Comparative Analysis and Final Comments

Let's begin to analyze in terms of scalability. It's easy to see (for the HASKELL approach), by running the *exec* function for a very large n, that the program slows down! In fact, the runtime is exponential, which one would expect given that it's a brute force implementation. However, UPPAAL model checking engine allows efficient and fast timed automata model exploration. So, for expensive executions, UPPAAL may be a better option. Also, in terms of systems 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. Syntax correctness is not enough. Therefore, it is more susceptible to errors.

Even so, UPPAAL have some considerable disadvantages. One of the main disadvantages 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), e.g., the optimal trace duration. In every state of every possible sequence of moves, monad gives us the "clock value", i.e. the duration of the moves so far. Also, concerning the space of solutions and taking into account the goal of reaching the final state with the aforementioned criteria, in the HASKELL approach, we saw that there were

two optimal solutions. However, 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!

On the other hand, the fact that UPPAAL produces counterexamples allows us to automatically get the trace (or path). After we verify some property (e.g. that is possible for all adventurers to be on the other side in $\leqslant 17$ minutes and not exceeding 5 moves), one could negate the property in order to obtain the trace log for that property. 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.

Considering all this, there's no way to say which approach is better. It all depends on the designer. A proof of this is that Melânia prefers Uppaal and Paulo prefers HASKELL.