My Report

Me

Friday 10th May, 2024

Abstract

We give a toy example of a report in *literate programming* style. The main advantage of this is that source code and documentation can be written and presented next to each other. We use the listings package to typeset Haskell source code nicely.

Contents

1 How to use this?

To generate the PDF, open `report.tex` in your favorite LaTeX editor and compile. Alternatively, you can manually do `pdflatex report; bibtex report; pdflatex report; pdflatex report` in a terminal.

You should have stack installed (see `https://haskellstack.org/`) and open a terminal in the same folder.

- To compile everything: `stack build`.
- To open ghci and play with your code: `stack ghci`
- To run the executable from Section 3: `stack build && stack exec myprogram`
- To run the tests from Section 4: `stack clean && stack test --coverage`

2 The most basic library

This section describes a module which we will import later on.

```haskell
module Basics where

import Control.Monad.State
import qualified Control.Category as Cat

type AState  = Int -- maybe wrap in constructor
type Output = Bool -- maybe wrap in constructor
type Letter = Int  -- maybe wrap in constructor

type DTrans = State AState Output -- StateT with Identity monad
type NTrans = StateT AState [] Output

-- easy composition of deterministic transitions
-- LY: this is the same as (>>)
-- (<.>) :: DTrans -> DTrans -> DTrans
-- (<.>) a b = a >>= const b -- ignore the output of the intermediate state

-- want to make the above work with this typeclass
-- LY: Category in haskell is defined to accept a kind (* -> * -> *), while DTrans has kind
    *
-- instance Category DTrans where
--  id = StateT $ \s -> (val s, s)
--  (.) = (>>)

-- easy shorthand for defining transitions
-- LY: Please change this name to something else. Better: This really sounds like (<$>).
l :: (AState -> AState) -> DTrans
l f = StateT $ \s -> return (val (f s), (f s))

-- good for testing
showRun :: AState -> DTrans -> String
showRun s t = show $ runState t s

type Valuation = AState -> Output -- could also do set of AState

-- for the moment our state set is N, and all states > 10 are accepting

val :: Valuation
val x = x > 10
```

```haskell
-- make a plus transition
plus :: Int -> DTrans
plus = \x -> l (+x)
-- pTrans = l $ (+)

-- make a subtraction transition
-- LY: VERY VERY VERY BAD name for this, this shadows ALL the s appearing in this file!
    Haskell does not evaluate from top to bottom! I also change the previous short names...
-- s :: Int -> DTrans
-- s = \x -> l (subtract x)
subt :: Int -> DTrans
subt = \x -> l (subtract x)

-- big list of transitions
tlist = [(subt 3), (plus 2), (plus 3), (subt 10), (plus 19)]

-- here's how we can compose em all together
composed = foldr (>>) (l id) tlist

test :: String
test = showRun 9 composed
-- initial state is 9

data DetAut = DA { states :: [AState]
                 , delta :: Letter -> DTrans }

type AutData = ( [AState]                    -- all states
               , [AState]                    -- accepting states
               , [(AState, Letter, AState)]) -- transitions

-- can it define a deterministic automaton? (totality of transition)
safetyCheck :: AutData -> Bool
safetyCheck = undefined

-- contingent on passing safetyCheck, make it into a DA
encodeDA :: AutData -> Maybe DetAut
encodeDA d | safetyCheck d = Just undefined
           | otherwise = Nothing

-- make it into an NA
-- encodeNA :: AutData -> NDetAut
-- encodeNA = undefined

newtype NDetState s a = NST { run :: s -> [(a, s)] }

-- s -> [(a,s)]
-- NTrans b = ST $ s ->

--
-- ~> for transitioning states?
```

3 Wrapping it up in an exectuable

We will now use the library form Section 2 in a program.

```haskell
module Main where

import Basics

main :: IO ()
main = do
  putStrLn "Hello!"
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

4 Simple Tests

We now use the library QuickCheck to randomly generate input for our functions and test some properties.

```
module Main where

import Basics

import Test.Hspec
import Test.QuickCheck
```

The following uses the HSpec library to define different tests. Note that the first test is a specific test with fixed inputs. The second and third test use QuickCheck.

```
main :: IO ()
main = hspec $ do
  describe "Basics" $ do
    it "somenumbers should be the same as [1..10]" $
      somenumbers `shouldBe` [1..10]
    it "if n > - then funnyfunction n > 0" $
      property (\n -> n > 0 ==> funnyfunction n > 0)
    it "myreverse: using it twice gives back the same list" $
      property $ \str -> myreverse (myreverse str) == (str::String)
```

To run the tests, use `stack test`.

To also find out which part of your program is actually used for these tests, run `stack clean && stack test` Then look for "The coverage report for ... is available at ... .html" and open this file in your browser. See also: `https://wiki.haskell.org/Haskell_program_coverage`.

5 Conclusion

Finally, we can see that [LW13] is a nice paper.

References

[LW13] Fenrong Liu and Yanjing Wang. Reasoning about agent types and the hardest logic puzzle ever. *Minds and Machines*, 23(1):123–161, 2013.