My Report

Me

Saturday 12th July, 2025

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 Syntax of

The language of $K\Box$ is defined over a set of propositional variables Prop:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Box\varphi \mid K\varphi \ ,$$

where $in\mathsf{Prop}$. We call $\Box$ and K the *safe belief* and *knowledge* operator respectively.

Below is the implementation of the syntax of $K\Box$, where we index the propositional variables with integers:

```
module Syntax where

data KSBForm = P Int | Neg KSBForm | Con KSBForm KSBForm | Box KSBForm | K KSBForm
```

The other boolean operators $\vee, \rightarrow, \top, \bot$ can be defined in the usual way:

```
dis, implies :: KSBForm -> KSBForm -> KSBForm
dis f g = Neg $ Con (Neg f) (Neg g)
implies f = dis (Neg f)
top, bot :: KSBForm
top = dis (P 0) (Neg $ P 0)
bot = Neg top
```

The *conditional belief* and *belief* operators can be defined as follows:

$$\begin{aligned} B^{\varphi}\psi &:= \tilde{K}\varphi \rightarrow \tilde{K}(\varphi \wedge \Box(\varphi \rightarrow \psi)) \ ; \\ B\varphi &:= B^{\top}\varphi \ , \end{aligned}$$

where $\tilde{K}\varphi := \neg K \neg\varphi$ is the dual of the knowledge operator.

```
cBel :: KSBForm -> KSBForm -> KSBForm
cBel f g = (Neg . K . Neg) f ‘implies‘ (Neg . K . Neg) (f ‘Con‘ Box (f ‘implies‘ g))

bel :: KSBForm -> KSBForm
bel = cBel top
```

2 How to use this?

To generate the PDF, open `report.tex` in your favorite LaTeXeditor 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 4: `stack build && stack exec myprogram`
- To run the tests from Section 5: `stack clean && stack test --coverage`

3 The most basic library

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

```
module Basics where

import Control.Monad
import System.Random

thenumbers :: [Integer]
thenumbers = [1..]

somenumbers :: [Integer]
somenumbers = take 10 thenumbers

randomnumbers :: IO [Integer]
randomnumbers = replicateM 10 $ randomRIO (0,10)
```

We can interrupt the code anywhere we want.

```
funnyfunction :: Integer -> Integer
funnyfunction 0 = 42
```

Even in between cases, like here. It's always good to cite something [Knu11].

```
funnyfunction n | even n     = funnyfunction (n-1)
                | otherwise = n*100
```

Something to reverse lists.

```
myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = myreverse xs ++ [x]
```

If you look at the `.lhs` file then below this line you can find some Haskell code.

But it does not show up in the PDF document. Please only use this for boring or repetitive parts of your code. Do not hide too much from your reader.

That's it, for now.

4 Wrapping it up in an exectuable

We will now use the library form Section 3 in a program.

```
module Main where

import Basics

main :: IO ()
main = do
  putStrLn "Hello!"
  print somenumbers
  print (map funnyfunction somenumbers)
  myrandomnumbers <- randomnumbers
  print myrandomnumbers
  print (map funnyfunction myrandomnumbers)
  putStrLn "GoodBye"
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

The output of the program is something like this:

```
Hello!
[1,2,3,4,5,6,7,8,9,10]
[100,100,300,300,500,500,700,700,900,900]
[1,3,0,1,1,2,8,0,6,4]
[100,300,42,100,100,100,700,42,500,300]
GoodBye
```

5 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`.

6 Conclusion

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

References

[Knu11] Donald E. Knuth. *The Art of Computer Programming. Combinatorial Algorithms, Part 1*, volume 4A. Addison-Wesley Professional, 2011.

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