

# My Report

Me

Thursday 13<sup>th</sup> January, 2022

## 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?	2
2	The most basic library	2
3	Wrapping it up in an executable	3
4	Simple Tests	3
5	Profiling	4
6	Conclusion	4
	Bibliography	4

# 1 How to use this?

To generate the PDF, open `report.tex` in your favorite  $\text{\LaTeX}$  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.

```
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 [?].

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

Something to reverse lists.

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

That's it, for now.

### 3 Wrapping it up in an executable

We will now use the library from Section 2 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"
```

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
```

Note that the above `showCode` block is only shown, but it gets ignored by the Haskell compiler.

### 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 Text.Printf (printf)
import Test.QuickCheck
import Test.QuickCheck.Test (isSuccess)
import System.Exit (exitSuccess, exitFailure)
```

The tuples in the following list consist of a name for the test and a `quickCheckResult` applied to a property. For example, we check that the output of `funnyfunction` is one of three possibilities. The second test checks that reversing twice gives the same string back.

```
tests :: [(String, IO Result)]
tests =
  [ ("funnytest1", quickCheckResult (\n -> funnyfunction n `elem` [42, n*100, (n-1)*100]) )
  , ("reversetest", quickCheckResult (\str -> myreverse (myreverse str) == (str::String)) )
  ]

main :: IO ()
main = do
  results <- mapM \(s,a) -> printf "\n%-20s: " s >> a) tests
  if all isSuccess results
  then exitSuccess
  else exitFailure
```

To run this, use `stack clean && stack test --coverage`. In particular this will generate a nice report using `hpc`. Look for “The coverage report for ... is available at ... .html” and open this html file in your browser. See [https://wiki.haskell.org/Haskell\\_program\\_coverage](https://wiki.haskell.org/Haskell_program_coverage) for more examples and information how to read the report.

## 5 Profiling

The GHC compiler comes with a profiling system to keep track of which functions are executed how often and how much time and memory they take. To activate this RTS, we compile and execute our program as follows:

```
stack clean
stack build --executable-profiling --library-profiling \
  --ghc-options="-fprof-auto -rtsopts"
stack exec -- myprogram +RTS -p -h
```

The results are saved in the file `myprogram.prof` which looks like this. Note for example, that `funnyfunction` was called on 14 entries.

COST CENTRE	MODULE	no. entries		individual		inherited	
				%time	%alloc	%time	%alloc
MAIN	MAIN	227	0	0.0	0.8	0.0	100.0
main	Main	455	0	0.0	29.6	0.0	38.6
funnyfunction	Basics	518	14	0.0	0.6	0.0	0.6
randomnumbers	Basics	464	0	0.0	0.3	0.0	8.4
randomRIO	System.Random	468	0	0.0	0.0	0.0	8.0
...							

For many more RTS options, see the GHC documentation online at [https://downloads.haskell.org/~ghc/latest/docs/html/users\\_guide/profiling.html](https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/profiling.html).

To speed up our program, especially in GHCi, we can also add the parameter `-fobject-code` to the `ghc-options` field in the `.cabal` file.

## 6 Conclusion

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