Advanced Programming

Testing and Assessment

Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

September 14, 2017

Today's Program

- Quick monad recap
- What is an assessment?
- Testing
- Property based testing

Our Two Favorite Type Classes

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b
   (>>)   :: m a -> m b -> m b
   fail   :: String -> m a
```

Make Functors (and Applicatives) For Your Monads

Whenever you make your type a monad, you should also make it a functor (and an applicative functor).

Here are implementations of `Functor` and `Applicative` for free (you can sometimes/often do better):

```
instance Monad T where
  ...

instance Functor T where
  fmap f xs = xs >>= return . f

instance Applicative T where
  pure = return
  af <*> ax = do f <- af
                 x <- ax
                 return (f x)
```

What Is An Assessment?

From the frontpage of the exam:

> *For each question your report should give an overview of your solution, including **an assessment** of how good you think your solution is and on which grounds you base your assessment (testing, gut feeling, proof of correctness, . . .).*

Assessment

You should document:

- Your assumptions (if any).
- How suitable is your choice of algorithms and data structures, often based on your assumptions
- The correctness/robustness of your code.
- An overall summary of the quality of your code.

And present evidence for your conclusions.

Morse Code

- One of the exercises from exercise set 0 is about decoding morse code.
- That is, write the functions

  ```
  encode :: String -> String
  decode :: String -> [String]
  ```

  For instance, `"...---..-....-"` could be the encoding for both Sofia and Eugenia.

Morse Code, Implementation

```haskell
import qualified Data.List as L

charMap = [('A', ".-"), ('B', "-..."), ... ]

findChar c = fromMaybe "" $ lookup c charMap

encode :: String -> String
encode = concatMap findChar
decode :: String -> [String]
decode ""    = [""]
decode input = [ c : rest | (c, code) <- charMap
                          , code `L.isPrefixOf` input
                          , let clen = length code
                          , rest <- decode $ drop clen input]
```

Assessment

You should document:

- Your assumptions (if any).
- How suitable is your choice of algorithms and data structures, often based on your assumptions
- The correctness/robustness of your code.
- An overall summary of the quality of your code.

And present evidence for your conclusions.

(Bad) Assessment of Morse Code Implementation

I believe that my solution for working with Morse code is quite good. The only weakness is that I use a list as my data structure for mapping chars to the corresponding Morse code. Instead I should probably have used an array or a map (tree or hash based). That would make ‘findChar‘ a constant time operation, which would make ‘encode‘ much faster. Alas, it also make my implementation of ‘decode‘ much uglier, and since the list is short i decided to stay with list.

I tested my functions in the REPL for many examples, and they always gave the correct results.

Correctness Claim by Testing

To claim that your code is correct, you must **as a minimum** do some kind of testing:

- Black-box testing
- White-box testing
- Functional testing
- Unit testing
- ...

Bare minimum, for each test:

- write down what you (think) you test,
- what is the expected outcome of the test,
- what was the outcome of the test.

All in a test schema/table (or as unit test). Summarise if needed.

Unit Testing In Haskell

The standard Unit testing framework for Haskell is HUnit, in the module `Test.HUnit`.

Unit tests for the `Morse` module

```
import Test.HUnit
import qualified Morse

test1 = TestCase $ assertBool "Decode Sofia" $
              "SOFIA" `elem` Morse.decode "...---..-....-"
test2 = TestCase $ assertBool "Decode Eugenia" $
              "EUGENIA" `elem` Morse.decode "..---..-....-"

tests = TestList [TestLabel "Decode" $ TestList [test1, test2],
                  TestLabel "Encode" $
                        "-.-.-." ~=? Morse.encode "KEN"]

main = runTestTT tests
```

Property Based Testing

To say something about the correctness of our code, we should be able to prove what *properties* holds for the code, or at least test that the properties hold for a few instances.

For instance, for the `Morse` module we would expect to be able to decode an encoded string:

$$s = decode(encode(s))$$

Alas, that's too strong a property. Several strings can have the same encoding. Thus the property we are after is

$$s \in decode(encode(s))$$

Using QuickCheck For Property Testing

The standard testing framework for property based testing in Haskell is QuickCheck, in the module `Test.QuickCheck`.

Again, for the we code up our property as a function, which we can then test with the function `quickCheck`

```
import Test.QuickCheck
import qualified Morse

prop_encode_decode s = s `elem` Morse.decode (Morse.encode s)

main = quickCheck prop_encode_decode
```

- QuickCheck generates random values by clever use of the `Arbitrary` type-class:

  ```
  class Arbitrary a where
    arbitrary :: Gen a
  ```

- That uses the type:

  ```
  newtype Gen a = MkGen { unGen :: QCGen -> Int -> a }
  ```

  to generate values of type `a`.
- `Gen` is a monad.

QuickCheck for Morse, Take 2

```haskell
import qualified Test.QuickCheck as QC
import qualified Data.Char as C
import qualified Morse

upper = map C.toUpper

prop_encode_decode (LO s) = upper s `elem`
                              Morse.decode (Morse.encode s)

asciiLetter = QC.elements (['a'..'z'] ++ ['A'..'Z'])

newtype LettersOnly = LO String
                    deriving (Eq, Show)

instance QC.Arbitrary LettersOnly where
  arbitrary = fmap LO (QC.listOf asciiLetter)
```

QuickCheck for Morse, Take 3

```haskell
import Test.QuickCheck
import qualified Data.Char as C
import qualified Morse

upper = map C.toUpper
prop_encode_decode (LO s) = upper s `elem`
                                   Morse.decode (Morse.encode s)

weightedLetters = frequency [(2 ^ (max - length code), return c)
                            | (c,code) <- Morse.charMap]
  where max = 1 + (maximum $ map (length . snd) Morse.charMap)

newtype LettersOnly = LO String deriving (Eq, Show)

instance Arbitrary LettersOnly where
  arbitrary = fmap LO $ do n <- choose (0, 5)
                           vectorOf n weightedLetters
```

Testing Algebraic Data Types

How can be generate random expressions for checking that Add is commutative:

```
data Expr = Con Int
          | Add Expr Expr
     deriving (Eq, Show, Read, Ord)

value :: Expr -> Int
value (Con n) = n
value (Add x y) = value x + value y

prop_com_add x y = value (Add x y) == value (Add y x)
```

Generating Exprs

- Our first attempt

```
expr =  oneof [ fmap Con arbitrary
              , do x <- expr
                   y <- expr
                   return $ Add x y]

instance Arbitrary Expr where
  arbitrary = expr
```

  is correct,
- ... but may generate humongous expressions.
- Instead we should generate a `sized` expression

```
expr = sized exprN

exprN 0 = fmap Con arbitrary
exprN n = oneof [fmap Con arbitrary,
                 liftM2 Add subexpr subexpr]
  where subexpr = exprN (n `div` 2)
```

Test your understanding: Check that minus is commutative

- Add constructor and extend `eval`.
- Extend data generator:

```
expr = sized exprN
exprN 0 = liftM Con arbitrary
exprN n = oneof [liftM Con arbitrary,
                 liftM2 Add subexpr subexpr,
                 liftM2 Minus subexpr subexpr
                ]
  where subexpr = exprN (n `div` 2)
```

- Write a property

```
prop_com_minus x y =
  eval (Minus x y) == eval (Minus y x)
```

Shrinking in Haskell

- The `Arbitrary` type class also specify the function `shrink`

  ```
  shrink :: a -> [a]
  ```

  Which should produces a (possibly) empty list of all the possible immediate shrinks of the given value.
- For `Exprs`

  ```
  instance Arbitrary Expr where
    arbitrary = sized exprN
      where expr N 0 = ...

    shrink (Add e1 e2) = [e1, e2]
    shrink (Minus e1 e2) = [e1, e2]
    shrink _ = []
  ```

Summary

- Practise making an assessments (it will be on the exam)
- To claim correctness you should have some kind of evidence, as a minimum some testing.
- Use HUnit for unit testing
- Use QuickCheck for better testing
- Write better generators by using the `Gen` monad