hspec: Testing framework

Hspec is a testing framework. It provides a friendly DSL for defining tests. It supports writing various kind of tests including unit, integration and property testing. tasty is another popular testing framework for Haskell.

Get started

```
$ stack new mylib rio --resolver lts-13.28
$ cd mylib
$ stack test
...
mylib> Test suite mylib-test passed
```

Woohoo, let's kick this off

Test driven development

Create an `src/Reverse.hs`:

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
module Reverse
  ( myReverse
  ) where
import RIO

myReverse :: [a] -> [a]
myReverse = undefined
```

Unit Testing

Create `test/ReverseSpec.hs`

```haskell
module ReverseSpec where

import Test.Hspec
import Test.Hspec.QuickCheck
import Reverse

spec :: Spec
spec = do
  describe "myReverse" $ do
    it "handles empty lists" $ myReverse [] `shouldBe` ([] :: [Int])
```

And now run in a terminal:

```
$ stack test --file-watch --fast
```

It fails, of course:

```
Failures:
  src/Reverse.hs:10:13:
  1) Reverse.myReverse handles empty lists
       uncaught exception: ErrorCall
       Prelude.undefined
       CallStack (from HasCallStack):
         error, called at libraries/base/GHC/Err.hs:78:14 in base:GHC.Err
         undefined, called at src/Reverse.hs:10:13 in mylib-0.1.0.0-FCBjIqB4mhCEtNQa5Mn13e:Reverse
```

Fix it...

```
myReverse :: [a] -> [a]
myReverse [] = []
```

Tests pass. Add another test...

```
    it "reverses hello" $ myReverse "hello" `shouldBe` "olleh"
```

It breaks, add more code:

```
myReverse (x:xs) = myReverse xs ++ [x]
```

Property testing

Yay! Let's add a property:

```
prop "double reverse is id" $ \list ->
    myReverse (myReverse list) `shouldBe` (list :: [Int])
```

That's inefficient, let's create a better reverse function. Again, TDD:

```
-- code
betterReverse :: [a] -> [a]
betterReverse = undefined

-- test
describe "betterReverse" $ do
  prop "behaves the same as myReverse" $ \list ->
    betterReverse list `shouldBe` myReverse (list :: [Int])
```

Notice Remember to export the `betterReverse` function at top of the code file.

Exercise Why is `myReverse` slow, and how can you make it faster?

Real buggy code I wrote by mistake...

```
betterReverse :: [a] -> [a]
betterReverse =
    loop []
  where
    loop res [] = []
    loop res (x:xs) = loop (x:res) xs
```

Test suite catches me!

```
  test/ReverseSpec.hs:16:7:
  1) Reverse.betterReverse behaves the same as myReverse
       Falsifiable (after 4 tests and 3 shrinks):
         [0]
       expected: [0]
        but got: []
```

OK, let's fix the code:

```
betterReverse :: [a] -> [a]
betterReverse =
    loop []
  where
    loop res [] = res
    loop res (x:xs) = loop (x:res) xs
```

Hurrah!

Let's play with vector

First let's add a `vector` dependency to `package.yaml`:

```
- vector >= 0.12.0.3
```

New `Reverse.hs` will now look like that:

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
module Reverse
  ( myReverse
  , betterReverse
  , vectorReverse
  , uvectorReverse
  , svectorReverse
  ) where

import RIO
import qualified RIO.Vector as V
import qualified RIO.Vector.Boxed as VB
import qualified RIO.Vector.Storable as VS
import qualified RIO.Vector.Unboxed as VU
import qualified Data.Vector.Generic.Mutable as VM

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

betterReverse :: [a] -> [a]
betterReverse =
    loop []
  where
    loop res [] = res
    loop res (x:xs) = loop (x:res) xs

vectorReverseGeneric
  :: V.Vector v a
  => [a]
  -> v a
vectorReverseGeneric input = V.create $ do
  let len = length input
  v <- VM.new len
  let loop [] idx = assert (idx == -1) (return v)
      loop (x:xs) idx = do
        VM.unsafeWrite v idx x
        loop xs (idx - 1)
  loop input (len - 1)
{-# INLINEABLE vectorReverseGeneric #-}

vectorReverse :: [a] -> [a]
vectorReverse = VB.toList . vectorReverseGeneric
{-# INLINE vectorReverse #-}

svectorReverse :: VS.Storable a => [a] -> [a]
svectorReverse = VS.toList . vectorReverseGeneric
{-# INLINE svectorReverse #-}

uvectorReverse :: VU.Unbox a => [a] -> [a]
uvectorReverse = VU.toList . vectorReverseGeneric
{-# INLINE uvectorReverse #-}
```

Awesome, but are they working as advertised ? Let's find out!

Add few properties which compares the result with the implemention of `reverse` function in the standard libray:

```
describe "compare with Data.List reverse" $ do
  prop "vectorReverse" $ \list ->
    vectorReverse list `shouldBe` (reverse (list :: [Int]))
  prop "svectorReverse" $ \list ->
    svectorReverse list `shouldBe` (reverse (list :: [Int]))
  prop "uvectorReverse" $ \list ->
    uvectorReverse list `shouldBe` (reverse (list :: [Int]))
```

And yeah, all the tests passes fine!

```
  compare with Data.List reverse
    vectorReverse
      +++ OK, passed 100 tests.
    svectorReverse
      +++ OK, passed 100 tests.
    uvectorReverse
      +++ OK, passed 100 tests.
```