Behavior-Driven Development in Haskell

a tutorial introduction

Simon Hengel
Wimdu GmbH
simon.hengel@wimdu.com

Abstract

Several widely used Haskell libraries do not come with automated tests. Test code assists the developer in maintaining correctness when extending software, and shows the user how an API is supposed to be used. This tutorial gives a short introduction to behavior-driven development in Haskell with Hspec. The illustrated approach can be used for design, documentation and automated testing. It aims at motivating the Haskell programmer to write more and better tests.

About the accompanying tutorial session

There will be a tutorial session on Hspec at the HaL8 Haskell Workshop[1]. This paper is provided as supplementary material. The tutorial session covers additional topics that are not discussed in this paper, in particular:

- There will be a live demonstration of some of Hspec's more advanced features.
- There will be exercises: Participants have the opportunity to use Hspec while developing a small but fully functional JSON web service.

A basic understanding of `Monad` and `Applicative` is required to follow the tutorial session. Prior experience with QuickCheck is useful but not strictly required.

Participants should install the latest version of the Haskell Platform[2] on their computers. In addition, there is a Git repository that contains a skeleton project as a starting point for the exercises. Participants can "`clone`" it and install all required dependencies with:

```
$ git clone https://github.com/sol/hspec-tutorial
$ cd hspec-tutorial
$ cabal install --enable-tests --only-dependencies
```

1. Introduction

Haskell has a strong type system which ensures that many common programming errors are detected at compile time rather than runtime. Thanks to this, we assume that programs written in Haskell have less bugs than programs written in other popular programming languages.

Unfortunately, the type system of Haskell does not always guarantee correctness. For example, it does not ensure that division by zero does not occur. A similar issue can be observed when looking at instances of the `Monoid` type class. They should be monoids in the mathematical sense, but the type system does not enforce this. It is the programmers responsibility to ensure that instance definitions are well-behaved. Hence automated testing is still necessary and useful.

However, several widely used Haskell libraries do not come with automated tests. Examples are `QuickCheck`, `parsec`, `OpenGL`, `transformers` and `mtl`. A lack of tests is not only a quality and maintainability issue. It is also a lack of documentation. Tests document how a library is supposed to be used, and tests document the external behavior of software.

In other programming language communities, *behavior-driven development (BDD)* is getting much attention and is slowly replacing *test-driven development (TDD)*. TDD is an iterative approach to software development. The developer first writes a test case, and only then implements the software in the simplest possible way that makes the test case pass. That could mean yielding a constant value at first. Subsequently, more test cases are written to require more details and drive the implementation closer to the desired goal.

BDD includes TDD, amongst other software development methodologies, but with a terminology and mindset shifted towards specification rather than testing. BDD is meant to be an enjoyable experience for the developer. Compared to TDD it provides a much nicer syntax that makes tests very easy to read. And like TDD it puts emphasis on test automation to take workload off the shoulders of developers.

In BDD, all behavior should be documented, together with an *example* for that behavior. Most existing BDD tools use unit tests as examples. Hspec[3], a BDD tool for Haskell inspired by RSpec [2], supports both unit tests and QuickCheck properties. This tutorial gives a short introduction to behavior-driven development in Haskell with Hspec. The illustrated approach can be used for design, documentation and automated testing. It aims at motivating the Haskell programmer to write more and better tests.

Section 2 gives a brief overview of Hspec. Section 3 uses a small example to show how BDD works in practice. Sections 4 and 5 discuss possible future extensions and related libraries. Section 6 concludes.

2. An overview of Hspec

Hspec can be used to document the external behavior of a system in a way that is similar to a specification. All facets of the system can and should be described with examples, including every possible corner case. The following subsections briefly introduce Hspec.

2.1 Defining specs

Hspec provides an EDSL for defining specs. A *spec* organizes examples in a tree structure (or a forest to be accurate) and is defined with `it` and `describe`. Here is a spec that describes the `reverse` function:

[1] http://www.bioinf.uni-leipzig.de/conference-registration/13haskell/

[2] http://www.haskell.org/platform/

[3] http://hspec.github.io/

```
import Test.Hspec
import Test.QuickCheck

main :: IO ()
main = hspec spec

spec :: Spec
spec = do
  describe "reverse" $ do
    it "reverses a list" $
      reverse [1, 2, 3] `shouldBe` [3, 2, 1]

    it "gives the original list if applied twice" $
      property $ \xs ->
        (reverse . reverse) xs == (xs :: [Int])
```

The `it` function combines a text description for a behavior and a corresponding example into a spec. Examples can be arbitrary HUnit assertions or QuickCheck properties.

The `describe` function combines a list of specs into a larger spec. A spec can be run with the `hspec` function. When a spec is run, all examples are tested and a report is generated. Examples that do not hold are marked with "`FAILED`" in the report.

2.2 Setting expectations

BDD frameworks use "should" or "must" instead of "assert" to express expectations. Hspec provides several combinators which can be used to set expectations about the outcome of examples. These are built on top of HUnit:

```
type Expectation = Assertion
```

A common expectation is that a value should be equal to another value:

```
x `shouldBe` 23
```

This is just another name for HUnit's `@?=` operator. The type is:

```
shouldBe :: (Show a, Eq a) => a -> a -> Expectation
```

Or we may expect that a value should satisfy some predicate.

```
x `shouldSatisfy` (< 23)
```

Here, the value is required to be in the `Show` class, so that a useful error message can be given when the predicate does not hold. The type of `shouldSatisfy` is:

```
shouldSatisfy :: Show a => a -> (a -> Bool)
  -> Expectation
```

Expectations are often used to test `IO` actions, e.g.:

```
launchMissiles >>=
  (`shouldBe` Left "not implemented")
```

This can be expressed more concisely as

```
launchMissiles `shouldReturn` Left "not implemented"
```

where `shouldReturn` has the following type:

```
shouldReturn :: (Show a, Eq a) => IO a -> a
  -> Expectation
```

2.3 Expecting exceptions

HUnit lacks the ability to assert that an `IO` action throws an exception. So Hspec provides a combinator for that:

```
shouldThrow :: Exception e => IO a -> (e -> Bool)
  -> Expectation
```

It takes an `IO` action and a predicate. The predicate servers two purposes. It selects the type of the expected exception and constrains the value. Here is an example:

```
launchMissiles `shouldThrow` (== ExitFailure 1)
```

Hspec provides type-restricted versions of `(const True)` for several standard exceptions to make it more convenient to expect arbitrary exception of a particular type, e.g.:

```
anyArithException :: ArithException -> Bool
anyArithException = const True
```

It can be used like this:

```
evaluate (1 `div` 0) `shouldThrow` anyArithException
```

2.4 Reusing specs

RSpec provides an explicit mechanism to facilitate the reuse of specs [2]. In Haskell we get reusability for free, it does not require any explicit support from Hspec.

Let's look at an example. Here is a spec that documents that a list is a monoid:

```
spec = do
  describe "List as a Monoid" $
    describe "mempty" $ do
      it "is a left identity" $
        property $ \x ->
          mempty <> x == (x :: [Int])

      it "is a right identity" $
        property $ \x ->
          x <> mempty == (x :: [Int])

    describe "<>" $ do
      it "is associative" $
        property $ \x y z ->
          (x <> y) <> z == (x <> (y <> z) :: [Int])
```

It is possible to make this spec reusable by abstracting over the list type:

```
shouldSatisfyMonoidLaws ::
  (Eq a, Show a, Monoid a, Arbitrary a) => a -> Spec
shouldSatisfyMonoidLaws t = do
  describe "mempty" $ do
    it "is a left identity" $
      property $ \x ->
        mempty <> x == x `asTypeOf` t

    it "is a right identity" $
      property $ \x ->
        x <> mempty == x `asTypeOf` t

  describe "<>" $ do
    it "is associative" $
      property $ \x y z ->
        (x <> y) <> z == x <> (y <> z) `asTypeOf` t
```

This can be used to document and test arbitrary monoid instances; the spec from above becomes:

```
spec = do
  describe "List as a Monoid" $ do
    shouldSatisfyMonoidLaws (undefined :: [Int])
```

2.5 Automatic spec discovery

It is a useful convention to have one spec file for each source file. That way it is straightforward to find the corresponding spec for a given piece of code. But this comes at a cost: To run all specs for a given project in one go it is necessary to combine all the specs

from all the spec files into a single spec. For example if we have three modules, `Foo`, `Foo.Bar` and `Baz`, and corresponding specs in `FooSpec`, `Foo.BarSpec` and `BazSpec`, we have to write the following boilerplate:

```
import Test.Hspec

import qualified FooSpec
import qualified Foo.BarSpec
import qualified BazSpec

main :: IO ()
main = hspec $ do
  describe "Foo"     FooSpec.spec
  describe "Foo.Bar" Foo.BarSpec.spec
  describe "Baz"     BazSpec.spec
```

This is error prone, and neither challenging nor interesting. So it should be automated. Hspec provides a solution for that. We make creative use GHC's support for custom preprocessors. The developer only has to creates a *test driver* that contains a single line:

```
-- file Spec.hs
{-# OPTIONS_GHC -F -pgmF hspec-discover #-}
```

This instructs GHC to invoke `hspec-discover` as a preprocessor on the source file. The rest of the source file is empty, so there is nothing to preprocess. Rather than preprocessing, `hspec-discover` scans the file system for all spec files belonging to a project and generates the required boilerplate. `hspec-discover` does not parse any source files, it instead relies on the following conventions:

1. Spec files have to be placed into the same directory as the test driver, or into a subdirectory.
2. The name of a spec file has to end in `Spec.hs`; the module name has to match the file name.
3. Each spec file has to export a top-level binding "`spec`" of type `Spec`.

2.6 Cabal integration

Cabal has support for running test suites [1]. A Cabal file can contain multiple *test suite sections*. When a user invokes "`cabal test`" all corresponding test suites are run. A suitable test suite section for Hspec looks like this:

```
test-suite spec
  type:          exitcode-stdio-1.0
  main-is:       Spec.hs
  build-depends: base, hspec
```

3. Applying BDD to Haskell: An Example

This section shows how behavior-driven development works in practice by means of the Fibonacci function. This is a very simple example, but the method is also applicable to more complicated situations where design decisions might not be as obvious as here.

We present a step-by-step refinement of a library and a corresponding spec. Each line of code is only given once when introduced. Subsequent code fragments are incremental updates. Import statements are omitted. Appendix A contains the complete code of this section.

3.1 Starting with a minimal implementation

We start with the function's documentation, which gives a usage example, together with the simplest possible implementation that makes the example work.

```
-- | Calculate Fibonacci numbers.
--
-- >>> fib 10
-- 55
fib :: Int -> Integer
fib = const 55
```

This first step might appear excessively trivial, but as of now we already have made some choices with regard to the API of the library. By giving a usage example, we were forced to think about which arguments the function takes. We can now use Doctest[4] to verify that the example works:

```
$ doctest Fib.hs
Examples: 1  Tried: 1  Errors: 0  Failures: 0
```

3.2 Getting it right

Next, we add a spec that forces our function to actually calculate Fibonacci numbers. At this stage we do not worry about invalid input or performance. We are happy with an implementation that works for small non-negative integers.

```
newtype Small = Small Int
  deriving Show

instance Arbitrary Small where
  arbitrary = Small . (`mod` 10) <$> arbitrary

main = hspec spec

spec = do
  describe "fib" $ do
    it "calculates arbitrary Fibonacci numbers" $ do
      property $ \(Small n) ->
        fib n == fib (n + 2) - fib (n + 1)
```

This spec fails when run:

```
$ runhaskell FibSpec.hs

fib
 - calculates arbitrary Fibonacci numbers FAILED [1]

1) fib calculates arbitrary Fibonacci numbers FAILED
*** Falsifiable (after 1 test):
Small 1

Finished in 0.0021 seconds

1 example, 1 failure
```

So we add a working implementation:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Both Doctest and Hspec report success now.

3.3 Making it fast

Our implementation is quite inefficient, as made evident by this spec:

```
it "is efficient" $ do
  timeout 10000 (evaluate $ fib 32)
    `shouldReturn` Just 2178309
```

When we run it, Hspec tells us:

[4] http://hackage.haskell.org/package/doctest

```
$ runhaskell FibSpec.hs

fib
 - calculates arbitrary Fibonacci numbers
 - is efficient FAILED [1]

1) fib is efficient FAILED
expected: Just 2178309
 but got: Nothing

Finished in 0.0685 seconds

2 examples, 1 failure
```

And we respond with an efficient implementation:

```
fib :: Int -> Integer
fib n = fibs !! n
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

With a much more efficient implementation at hand, we can now adapt our definition of small non-negative integers, so that a wider range of input values is tested:

```
instance Arbitrary Small where
  arbitrary = Small . (`mod` 1000) <$> arbitrary
```

3.4 Handling invalid input

We have an implementation of the Fibonacci function that calculates correct values for all non-negative integers. But what about negative ones? Let's assume that we want our implementation to throw an exception in this case. We can specify it like this:

```
it "throws ErrorCall on negative input" $ do
  evaluate (fib (-10)) `shouldThrow` anyErrorCall
```

Our implementation already fulfills this spec, so we are fine.

3.5 Dealing with specification changes

At some point we realize that throwing an exception on invalid input is not suitable for our use-case. Hence, we change the specification, so that negative input values should result in zero.

```
it "returns 0 on negative input" $ do
  property $ \n -> n < 0 ==>
    fib n == 0
```

Now our specification fails, so we adapt the implementation accordingly.

```
fib n | n < 0     = 0
      | otherwise = fibs !! n
```

This concludes our example. We have seen how to utilize behavior-driven development to incrementally build a small library. We always extend or adapt the specification first, watch it fail, and then tighten the implementation. The combinators `shouldBe`, `shouldReturn` and `shouldThrow` can be used to specify behavior of functions in an intuitive, easy to read manner.

4. Possible future extensions

Section 2.4 shows how to reuse specs. It would be possible and useful to write a library that provides reusable specs for many of Haskell's standard type classes (similar in purpose to `checkers`[5]). This would provide a convenient mechanism for developers to document and test their instance definitions.

When Hspec finds a failing example it reports the corresponding text description to the developer. It would be useful to include the source location of the failing example, so that the developer can find the corresponding spec immediately. It is possible to solve this with Template Haskell [4]. However, Template Haskell is not available on all platforms supported by GHC. Consequently, depending on Template Haskell would prevent us from running test on those platforms. Extending GHC with a mechanism similar to JHC's `SRCLOC_ANNOTATE` pragma[6] would allow us to provide source locations for failing examples without relying on Template Haskell.

5. Related libraries

SmallCheck [3] is a tool for exhaustive testing up to a certain depth. There is no SmallCheck support for Hspec. But Hspec's mechanism for examples is extensible, so adding support for SmallCheck is easy.

There are several libraries for TDD in Haskell. The most popular option is `test-framework`. It integrates with HUnit, QuickCheck and SmallCheck. Automatic test discovery that relies on naming conventions for test functions is provided by a third-party library[7]. It is implemented with Template Haskell and only works within a single source file. Test discovery across multiple source files is not possible with a mechanism that is based on Template Haskell, because Template Haskell can not generate import statements[8].

The `testpack`[9] library provides a combinator that allows to expect exceptions, similar to `shouldThrow` from Section 2.3. Its type is:

```
assertRaises :: (Show a, Exception e, Show e, Eq e)
  => String -> e -> IO a -> IO ()
```

This has several shortcomings: It only works for exceptions with an `Eq` instance, but several standard exceptions (most prominently `ErrorCall`[10]) have no `Eq` instance. In addition it places an unnecessary `Show` constraint on the result of the `IO` action, further limiting its applicability. Moreover, it only allows to expect specific exception values. Expecting arbitrary exceptions of a particular type is not possible.

6. Conclusion

Section 2 gave a brief overview of Hspec. A DSL for defining specs was presented and combinators for setting expectations were introduced. It was shown how specs can be reused and a mechanism for automatic spec discovery was discussed.

Section 3 uses Hspec to incrementally develop the ubiquitous Fibonacci function. It is a very simple example, but it already illustrates how behavior-driven development with Hspec works in practice.

Section 4 discussed possible future extensions. In particular, a mechanism that gives source locations for failing examples would be useful. A library with reusable specs for standard type classes would provide a convenient mechanism to document and test instance definitions.

Section 5 compared Hspec with existing libraries. Notably `test-framework`, a library for TDD, lacks a mechanism for automatic test discovery that works across multiple source files.

There will be an accompanying tutorial session on Hspec at the HaL8 Haskell Workshop which covers additional topics. Participants will have the opportunity to solve practical problems with Hspec and give behavior-driven development in Haskell a try.

[5] http://hackage.haskell.org/package/checkers

[6] http://repetae.net/computer/jhc/jhc.shtml

[7] http://hackage.haskell.org/package/test-framework-th

[8] See http://hackage.haskell.org/trac/ghc/ticket/1475

[9] http://hackage.haskell.org/package/testpack

[10] This will be fixed in `base-4.7.0.0`.

A. The final code from Section 3

```
-- file Fib.hs
module Fib (fib) where

-- | Calculate Fibonacci numbers.
--
-- >>> fib 10
-- 55
fib :: Int -> Integer
fib n | n < 0     = 0
      | otherwise = fibs !! n

fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

-- file FibSpec.hs
module FibSpec (main, spec) where

import Control.Applicative
import Control.Exception
import System.Timeout

import Test.Hspec
import Test.QuickCheck

import Fib

-- small non-negative integers
newtype Small = Small Int
  deriving Show

instance Arbitrary Small where
  arbitrary = Small . (`mod` 1000) <$> arbitrary

main :: IO ()
main = hspec spec

spec :: Spec
spec = do
  describe "fib" $ do
    it "calculates arbitrary Fibonacci numbers" $ do
      property $ \(Small n) ->
        fib n == fib (n + 2) - fib (n + 1)

    it "is efficient" $ do
      timeout 10000 (evaluate $ fib 32)
        `shouldReturn` Just 2178309

    it "returns 0 on negative input" $ do
      property $ \n -> n < 0 ==>
        fib n == 0
```


Acknowledgments

This paper is derived from a draft paper that has been written by Kazuhiko Yamamoto, Markus Klinik and me in May 2012. I'm grateful to Kazuhiko Yamamoto and Markus Klinik for their work on the draft paper and for giving me permission to base my work upon it.

References

[1] *Cabal User Guide*. URL http://www.haskell.org/cabal/users-guide/.

[2] D. Chelimsky. *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, 2010.

[3] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proc. of Haskell Symposium '08*, 2008.

[4] T. Sheard and S. Jones. Template metaprogramming for haskell. In *Proc. of Haskell Workshop 2002*, 2002.