Exploring Languages with Interpreters and Functional Programming Chapter 12

H. Conrad Cunningham

26 October 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of October 2018 is a recent version of Firefox from Mozilla.

12 Testing Haskell Programs

12.1 Chapter Introduction

The goal of this chapter is to illustrate the testing techniques by manually constructing test scripts for Haskell functions and modules. It builds on the concepts and techniques surveyed in the previous chapter.

We use two testing examples: the group of factorial functions from Chapters 4 and 9 and the rational arithmetic modules from Chapter 7.

12.2 Organizing Tests

Testers commonly organize unit tests on a system using the *Arrange-Act-Assert pattern* [Beck 2003] [Koskela 2013].

1. *Arrange:* Select input values from the input domain and construct appropriate "objects" to use in testing the test subject.
2. *Act:* Apply some operation from the test subject to appropriate input "objects".
3. *Assert:* Determine whether or not the result satisfies the specification.

Each test should create the test-specific input "objects" it needs and remove those and any result "objects" that would interfere with other tests.

Note: In this chapter, we use the word "object" in a general sense of any data entity, not in the specific sense defined for object-based programming.

12.3 Testing Functions

This section approaches testing of a group of Haskell functions as follows.

Testing level: unit testing of each Haskell function
Testing method: primarily black-box testing of each Haskell function relative to its specification
Testing type: functional testing of each Haskell function relative to its specification

12.3.1 Example

As an example, consider the set of seven factorial functions developed in Chapter 4 and 9 (in source file `Factorial.hs`). All have the requirement to implement the mathematical function

$$fact(n) = \prod_{i=1}^{i=n} i$$

for any $n \geq 0$. The specification is ambiguous on what should be the result of calling the function with a negative argument.

12.3.2 Arrange

To carry out black-box testing, we must *arrange* our input values. The factorial function tests do not require any special testing "objects".

We first partition the input domain. We identify two equivalence classes of inputs for the factorial function:

1. the set of nonnegative integers for which the mathematical function is defined and the Haskell function returns that value within the positive `Int` range
2. the set of nonnegative integers for which the mathematical function is defined but the Haskell function returns a value that overflows the `Int` range

The class 2 values result are errors, but integer overflow is typically not detected by the hardware.

We also note that the negative integers are outside the range of the specification.

Next, we select the following values inside the "lower" boundary of class 1 above:

- 0, empty case at the lower boundary
- 1, smallest nonempty case at the lower boundary

Then we choose representative values within class 1:

- 2, one larger than the smallest nonempty case
- 5, arbitrary value representative of values away from the boundary

Note: The choice of two representative values might be considered a violation of the "minimize test overlap" principle from the previous chapter. So it could be acceptable to drop the input of 2. Of course, we could argue that we should check 2 as a possible boundary value.

We also select the value -1, which is just outside the lower boundary implied by the $n \geq 0$ requirement.

All of the factorial functions have the type signature (where `N` is `1`, `2`, `3`, `4`, `4'`, `5`, or `6`):

```
factN :: Int -> Int
```

Thus the `factN` functions also have an "upper" boundary that depends on the maximum value of the `Int` type on a particular machine. The author is testing these functions on a machine with 64-bit, two's complement integers. Thus the largest integer whose factorial is less than 2^{63} is 20.

We thus select input the following input values:

- 20, which is just inside the upper boundary of class 1
- 21, which is just outside class 1 and inside class 2

12.3.3 Act

We can test a factorial function at a chosen input value by simply applying the function to the value such as the following:

```
fact1 0
```

A Haskell function has no side effects, so we just need to examine the integer result returned by the function to determine whether it satisfies the function's specification.

12.3.4 Assert

We can test the result of a function by stating a Boolean expression—an assertion—that the value satisfies some property that we want to check.

In simple cases like the factorial function, we can just compare the actual result for equality with the expected result. If the comparison yields `True`, then the test subject "passes" the test.

```
fact1 0 == 1
```

12.3.5 Aggregating into test script

There are testing frameworks for Haskell (e.g. HUnit [HUnit 2018], QuickCheck [QuickCheck 2018], or Tasty [Tasty 2018]), but, in this section, we manually develop a simple test script.

We can state a Haskell `IO` program to print the test and whether or not it passes the test. (Simple input and output will eventually be discussed in a previous chapter. For now, see the Haskell Wikibooks [Wikibooks-Haskell 2018] page "Haskell/Simple input and output".)

Below is a Haskell `IO` script that tests class 1 boundary values 0 and 1 and "happy path" representative values 2 and 5.

```
pass :: Bool -> String
pass True  = "PASS"
pass False = "FAIL"

main :: IO ()
main = do
```

```
putStrLn  "\nTesting fact1"
putStrLn ("fact1 0 == 1:      " ++ pass (fact1 0 == 1))
putStrLn ("fact1 1 == 1:      " ++ pass (fact1 1 == 1))
putStrLn ("fact1 2 == 2:      " ++ pass (fact1 2 == 2))
putStrLn ("fact1 5 == 120:    " ++ pass (fact1 5 == 120))
```

The `do` construct begins a sequence of `IO` commands. The `IO` command `putStrLn` outputs a string to the standard output followed by a newline character.

Testing a value below the lower boundary of class 1 is tricky. The specification does not require any particular behavior for -1. As we saw in Chapter 4, some of the function calls result in overflow of the runtime stack, some fail because all of the patterns fail, and some fail with an explicit `error` call. However, all these trigger a Haskell exception.

Our test script can `catch` these exceptions using the following code.

```
putStrLn ("fact1 (-1)  == 1:  "
          ++ pass (fact1 (-1)  == 1))
    `catch` (\(StackOverflow)
              -> putStrLn ("[Stack Overflow] (EXPECTED)"))
    `catch` (\(PatternMatchFail msg)
              -> putStrLn ("[Pattern Match Failure]\n...."
                           ++ msg))
    `catch` (\(ErrorCall msg)
              -> putStrLn ("[Error Call]\n...." ++ msg))
```

To catch the exceptions, the program needs to import the module `Control.Exception` from the Haskell library.

```
import Prelude hiding (catch)
import Control.Exception
```

By catching the exception, the test program prints an appropriate error message and then continues with the next test; otherwise the program would halt when the exception is thrown.

Testing an input value in class 2 (i.e. outside the boundary of class 1) is also tricky.

First, the values we need to test depend on the default integer (`Int`) size on the particular machine.

Second, because the actual value of the factorial is outside the `Int` range, we cannot express the test with Haskell `Int`s. Fortunately, by converting the values to the unbounded `Integer` type, the code can compare the result to the expected value.

The code below tests input values 20 and 21.

```
putStrLn ("fact1 20 == 2432902008176640000:   "
          ++ pass (toInteger (fact1 20) ==
```

```
                                    2432902008176640000))
    putStrLn ("fact1 21 == 51090942171709440000:  "
              ++ pass (toInteger (fact1 21) ==
                                    51090942171709440000)
              ++ " (EXPECT FAIL for 64-bit Int)" )
```

The above is a black-box unit test. It is not specific to any one of the seven factorial functions defined in Chapters 4 and 9. (These are defined in the source file `Factorial.hs`.) The series of tests can be applied any of the functions.

The test script for the entire set of functions from Chapters 4 and 9 (and others) are in the source file `TestFactorial.hs`.

12.4 Testing Modules

This section approaches testing of Haskell modules as follows.

Testing level: module-level testing of each Haskell module
Testing method: primarily black-box testing of each Haskell module relative to its specification
Testing type: functional testing of each Haskell module relative to its specification

Normally, module-level testing requires that unit-level testing be done for each function first. In cases where the functions within a module are strongly coupled, unit-level and module-level testing may be combined into one phase.

12.4.1 Example modules

For this section, we use the rational arithmetic example from Chapter 7.

In the rational arithmetic example, we define two abstract (information-hiding) modules: `RationalRep` and `Rational`.

Given that the `Rational` module depends on the `RationalRep` module, we first consider testing the latter.

12.4.2 Data representation modules

Chapter 7 defines the abstract module `RationalRep` and presents two distinct implementations, `RationalCore` and `RationalDeferGCD`. The two implementations differ in how the rational numbers are represented using data type `Rat`. (See source files `RationalCore.hs` and `RationalDeferGCD.hs`.)

Consider the public function signatures of `RationalRep` (from Chapter 7):

```
makeRat :: Int -> Int -> Rat
numer   :: Rat -> Int
denom   :: Rat -> Int
zeroRat :: Rat
showRat :: Rat -> String
```

Because the results of `makeRat` and `zeroRat` and the inputs to `numer`, `denom`, and `showRat` are abstract, we cannot test them directly as we did the factorial functions in a previous section. For example, we cannot just call `makeRat` with two integers and compare the result to some specific concrete value. Similarly, we cannot test `numer` and `denom` directly by providing them some specific input value.

However, we can test both through the abstract interface, taking advantages of the interface invariant.

RationalRep Interface Invariant (from Chapter 7): For any valid Haskell rational number `r`, all the following hold:

- `r` ∈ `Rat`
- `denom r > 0`
- if `numer r == 0`, then `denom r == 1`
- `numer r` and `denom r` are relatively prime
- the (mathematical) rational number value is $\frac{\texttt{numer r}}{\texttt{denom r}}$

The invariant allows us to check combinations of the functions to see if they give the expected results. For example, suppose we define `x'` and `y'` as follows:

```
x' = numer (makeRat x y)
y' = denom (makeRat x y)
```

Then the interface invariant and contracts for `makeRat`, `numer`, and `denom` allow us to infer that the (mathematical) rational number values $\frac{\texttt{x'}}{\texttt{y'}}$ and $\frac{\texttt{x}}{\texttt{y}}$ are equal.

This enables us to devise pairs of test assertions such as

```
numer (makeRat 1 2) == 1
denom (makeRat 1 2) == 2
```

and

```
numer (makeRat 4 (-2)) == -2
denom (makeRat 4 (-2)) == 1
```

to indirectly test the functions in terms of their interactions with each other. All the tests above should succeed if the module is designed and implemented according to its specification.

Similarly, we cannot directly test the private functions `signum'`, `abs'`, and `gcd'`. But we try to choose inputs the tests above to cover testing of these functions. (Private functions should be tested as the module is being developed to detect any more problems.)

12.4.2.1 Arrange

To conduct black-box testing, we must arrange the input values we wish to test. The module tests do not require any special test objects, but each pair of tests both create a `Rat` object with `makeRat` and select its numerator and denominator with `numer` and `denom`.

However, for convenience, we can define the following shorter names for constants:

```
maxInt  = (maxBound :: Int)
minInt  = (minBound :: Int)
```

TODO: Draw a diagram as discussed

Each pair of tests has two `Int` parameters—the `x` and `y` parameters of `makeRat`. Thus we can visualize the input domain as the integer grid points on an `x-y` coordinate plane using the usual rectangular layout from high school algebra.

We note that any input `x-y` value along the `x`-axis does not correspond to a rational number; the pair of integer values does not satisfy the precondition for `makeRat` and thus result in an `error` exception.

For the purposes of our tests, we divide the rest of the plane into the following additional partitions (equivalence classes):

- the `y`-axis

  Input arguments where `x == 0` may require special processing because of the required unique representation for rational number zero.

- each quadrant of the plane (excluding the axes)

  The `x-y` values in different quadrants may require different processing to handle the `y > 0` and "relatively prime" aspects of the interface invariant.

  Given that the module uses the finite integer type `Int`, we bound the quadrants by the maximum and minimum integer values along each axis.

We identify the following boundary values for special attention in our tests.

- Input pairs along the `x`-axis are outside any of the partitions.
- Input pairs composed of integer values 0, 1, and -1 are on the axes or just inside the "corners" of the quadrants . In addition, these are special values in various mathematical properties.
- Input pairs composed of the maximum `Int` (`maxInt`) and minimum `Int` (`minInt`) values may be near the outer bounds of the partitions.

  Note: If the machine's integer arithmetic uses the two's complement representation, then `minInt` can cause a problem with overflow because its negation is not in `Int`. Because of overflow, `-minInt == minInt`. So we should check both `minInt` and `-maxInt` in most cases.

In addition, we identify representative values for each quadrant. Although we do not partition the quadrants further, in each quadrant we should choose some input values whose (mathematical) rational number values differ and some whose values are the same.

Thus we choose the following `(x,y)` input pairs for testing:

- (0,0), (1,0), and (-1,0) as error inputs along the `x`-axis
- (0,1), (0,-1), (0,9), and (0,-9) as inputs along the `y`-axis
- (1,1), (9,9), and (`maxInt`,`maxInt`) as inputs from the first quadrant and (-1,-1), (-9,-9), and (`-maxInt`,`-maxInt`) as inputs from the third quadrant, all of whom have the same rational number value $\frac{1}{1}$.

  We also test input pairs (`minInt`,`minInt`) and (`-minInt`,`-minInt`), cognizant that the results might depend upon the machine's integer representation.
- (-1,1), (-9,9), and (`-maxInt`,`maxInt`) as inputs from the second quadrant and (1,-1), (9,-9), and (`maxInt`,`-maxInt`) as inputs from the fourth quadrant, all of whom have the same rational number value $-\frac{1}{1}$.

  We also test input pairs (`-minInt`,`minInt`) and (`minInt`,`-minInt`), cognizant that the results might depend upon the machine's integer representation.
- (3,2) and (12,8) as inputs from the first quadrant and (-3,-2) and (-12,-8) as inputs from the third quadrant, all of whom have the same rational number value $\frac{3}{2}$.
- (-3,2) and (-12,8) as inputs from the second quadrant and (3,-2) and (12,-8) as inputs from the fourth quadrant, all of whom have the same rational number value $-\frac{3}{2}$.
- (`maxInt`,1), (`maxInt`,-1) (`-maxInt`,1) and (`-maxInt`,-1) as input values in the "outer corners" of the quadrants.

  We also test input pairs (`minInt`,1) and (`minInt`,-1), cognizant that the results might depend upon the machine's integer representation.

12.4.2.2 Act

As we identified in the introduction to this example, we must carry out a pair of actions in our tests. For example,

```
numer (makeRat 12 8)
```

and

```
denom (makeRat 12 8)
```

for the test of the input pair (12,8).

Note: The code above creates each test object (e.g. `makeRat 12 8`) twice. These could be created once and then used twice to make the tests run slightly faster.

12.4.2.3 Assert

The results of the test actions must then be examined to determine whether they have the expected values. In the case of the `makeRat-numer-denom` tests, it is sufficient to compare the result for equality with the expected result. The expected result must satisfy the interface invariant.

For the two actions listed above, the comparison are

```
numer (makeRat 12 8) == 3
```

and

```
denom (makeRat 12 8) == 2
```

for the test of the input pair (12,8).

12.4.2.4 Aggregate into test script

As with the factorial functions in a previous section, we can bring the various test actions together into a Haskell `IO` program. The excerpt below shows some of the tests.

```
pass :: Bool -> String
pass True  = "PASS"
pass False = "FAIL"

main :: IO ()
main =
    do
        -- Test 3/2
        putStrLn ("numer (makeRat 3 2) == 3:               " ++
                  pass (numer (makeRat 3 2) == 3))
        putStrLn ("denom (makeRat 3 2) == 2:               " ++
                  pass (denom (makeRat 3 2) == 2))
        -- Test -3/-2
        putStrLn ("numer (makeRat (-3) (-2)) == 3:         " ++
                  pass (numer (makeRat (-3) (-2)) == 3))
        putStrLn ("denom (makeRat (-3) (-2)) == 2:         " ++
                  pass (denom (makeRat (-3) (-2)) == 2))
        -- Test 12/8
        putStrLn ("numer (makeRat 12 8) == 3:              " ++
                  pass (numer (makeRat 12 8) == 3))
        putStrLn ("denom (makeRat 12 8) == 2:              " ++
```

```
          pass (denom (makeRat 12 8) == 2))
-- Test -12/-8
putStrLn ("numer (makeRat (-12) (-8)) == 3:        " ++
          pass (numer (makeRat (-12) (-8)) == 3))
putStrLn ("denom (makeRat (-12) (-8)) == 2:        " ++
          pass (denom (makeRat (-12) (-8)) == 2))
-- Test 0/0
putStrLn ("makeRat 0 0 is error:                   "
          ++ show (makeRat 0 0))
    `catch` (\(ErrorCall msg)
                -> putStrLn ("[Error Call] (EXPECTED)\n"
                        ++ msg))
```

The first four pairs of tests above check the test inputs (3,2), (-3,-2), (12,8), and (-12,-8). These are four test inputs, drawn from the first and third quadrants, that all have the same rational number value $\frac{3}{2}$.

The last test above checks whether the error pair (0,0) responds with an error exception as expected.

For the full test script (including tests of `showRat`) examine the source file `TestRatRepCore.hs` or `TestRatRepDefer.hs`.

12.4.2.5 Broken encapsulation

So far, the tests have assumed that any rational number object passed as an argument to `numer`, `denom`, and `showRat` is an object returned by `makeRat`.

However, the encapsulation of the data type `Rat` within a `RationalRep` module is just a convention. `Rat` is really an alias for `(Int,Int)`. The alias is exposed when the module is imported.

A user could call a function and directly pass an integer pair. If the integer pair does not satisfy the interface invariant, then the functions might not return a valid result.

For example, if we call `numer` with the invalid rational number value (1,0), what is returned?

Because this value is outside the specification for `RationalRep`, each implementation could behave differently. In fact, `RationalCore` returns the first component of the tuple and `RationalDeferGCD` throws a "divide by zero" exception.

The test scripts include tests of the invalid value (1,0) for each of the functions `numer`, `denom`, and `showRat`.

A good solution to this broken encapsulation problem is (a) to change `Rat` to a user-defined type and (b) only export the type name but not its components. Then the Haskell compiler will enforce the encapsulation we have assumed. We discuss approach in later chapters.

12.4.3 Rational arithmetic modules

TODO

The interface to the module `Rational` consists of the functions `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat`, the `RationalRep` module's interface. It does not add any new data types, constructors, or destructors.

The `Rational` abstract module's functions preserve the interface invariant for the `RationalRep` abstract module, but it does not add any new components to the invariant.

12.4.3.1 Arrange

TODO

TODO: Draw a diagram to help visualize input domain

12.4.3.2 Act

TODO

12.4.3.3 Assert

TODO

12.4.3.4 Aggregate into test script

TODO: Discuss `TestRational1.hs` and `TestRational2.hs`

12.4.4 Reflection on this example

TODO: Update after completing chapter

I designed and implemented the `Rational` and `RationalCore` modules using the approach described in the early sections of Chapter 7, doing somewhat ad hoc testing of the modules with the REPL. I later developed the `RationalDeferGCD` module, abstracting from the `RationalCore` module. After that, I wrote Chapter 7 to describe the example and the development process. Even later, I constructed the systematic test scripts and wrote Chapters 11 and 12 (this chapter).

As I am closing out the discussion of this example, I find it useful to reflect upon the process.

- The problem seemed quite simple, but I learned there are several subtle issues in the problem and the modules developed to solve it. As the saying goes, "the devil is in the details".

- In my initial development and testing of these simple modules, I got the "happy paths" right and covered the primary error conditions. Although singer Bobby McFerrin's song "Don't Worry, Be Happy" may give good advice for many life circumstances, it should not be taken too literally for software development and testing.
- In writing both Chapter 7 and this chapter, I realized that my statements of the preconditions, postconditions, and interface invariants of `RationalRep` abstraction needed to be reconsidered and restated more carefully. Specifying a good abstract interface for a family of modules is challenging.
- In developing the systematic test scripts, I encountered other issues I had either not considered sufficiently or overlooked totally:
  - the full implications of using the finite data `Int` data type for the rational arithmetic modules
  - the impact of the underlying integer arithmetic representation (e.g. as two's complement) on the Haskell code
  - the effects of calls of functions like `numer`, `denom`, and `showRat` with invalid input data
  - a subtle violation of the interface invariant in the `RationalDeferGCD` implementations of `makeRat` and `showRat`
  - the value of a systematic input domain partitioning for both developing good tests and understanding the problem

It took me much longer to develop the systematic tests and document them than it did to develop the modules initially. I clearly violated the Meszaros's final principle, "ensure commensurate effort and responsibility" described in the previous chapter (also in [Meszaros 2007, Ch. 5]).

For future programming, I learned I need to pay attention to other of Meszaros's principles such as "design for testability", "minimize untestable code", "communicate intent", and perhaps "write tests first" or at least to develop the tests hand-in-hand with the program.

12.5 What Next?

This chapter and the previous one examined software testing concepts and applied them to testing Haskell functions and modules from Chapters 4 and 7.

The next two chapters explore first-order, polymorphic list programming in Haskell.

12.6 Exercises

1. Using the approach of this chapter, develop a black-box unit-testing script for the `fib` and `fib2` Fibonacci functions from Chapter 9. Test the functions with your script.
2. Using the approach of this chapter, develop a black-box unit-testing script for the `expt`, `expt2`, and `expt3` exponentiation functions from Chapter 9. Test the functions with your script.
3. Using the approach of this chapter, develop a black-box unit/module-testing script for the module `Sqrt` from Chapter 6. Test the module with your script.
4. Using the approach of this chapter, develop a black-box unit/module-testing script for the line-segment modules developed in exercises 1-3 of Chapter 7. Test the module with your script.

12.7 Acknowledgements

I wrote this chapter in Summer 2018 for the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming*.

- The presentation builds on the concepts and techniques surveyed in the previous chapter, which was written at the same time.
- The presentation and use of the Arrange-Act-Assert pattern draws on the discussion in [Beck 2003] and [Koskela 2013].
- The testing examples draw on previously existing function and (simple) test script examples and on discussion of the examples in Chapters 4 and 7. However, I did redesign and reimplement the test scripts to be more systematic and to follow the discussion in this new chapter.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

12.8 References

[Beck 2003]: Kent Beck. *Test-Driven Development: By Example*, Addison Wesley, 2003.

[HUnit 2018]: Haskell Organization Hackage. HUnit: A Unit-Testing Framework for Haskell. http://hackage.haskell.org/package/HUnit, accessed 20 July 2018.

[Koskela 2013]: Lasse Koskela. *Effective Unit Testing*, Manning, 2103.

[**QuickCheck 2018**]: Haskell Organization Hackage. QuickCheck: Automatic Checking of Haskell Programs, http://hackage.haskell.org/package/QuickCheck, accessed 20 July 2018.

[**Meszaros 2007**]: Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*, Pearson Education, 2007.

[**Tasty 2018**]: Haskell Organization Hackage. Tasty: Modern and Extensible Testing Framework, https://hackage.haskell.org/package/tasty, accessed 20 July 2018.

[**Wikibooks-Haskell 2018**]: Wikibooks: Open books for an open world. Haskell, https://en.wikibooks.org/wiki/Haskell, accessed 20 July 2018.

12.9 Terms and Concepts

Test, testing level, testing method, testing type, unit and module testing (levels), black-box and gray-box testing (methods), functional testing (type), arrange-act-assert, input domain, input partitioning, representative values (for equivalence classes), boundary values, testing based on the specification, Haskell `IO` program, `do`, `putStrLn`, exceptions.