Exercise (Day 2): Property Driven Development

Adventures in Property Based Testing

John Hughes

CHALMERS

QuviQ

For these exercises you will need to have GHC installed, together with the current version of QuickCheck. If you don't already have GHC, then I suggest installing the Haskell Platform from this URL:

https://www.haskell.org/platform/

You can install QuickCheck using the Cabal package manager. Run the commands

```
cabal update
cabal install quickcheck
```

which updates the package database and installs the latest version of QuickCheck. At the time of writing, this is version 2.13.1. You can find the documentation for QuickCheck here:

http://hackage.haskell.org/package/QuickCheck-2.13.1/docs/Test-QuickCheck.html

Introducing QuickCheck

If you have used QuickCheck before, you can skip this section.

Open the file **`SimpleStart.hs`** in your editor. This file defines two properties of the **`reverse`** function for lists: one (true) property, that it is its own inverse, and one false property, that it is the identify function. Notice that we give a type signature for the properties: this is to tell QuickCheck what kind of test data to generate, in this case lists of integers. Notice also that the return type is not **`Bool`**, but **`Property`**: this is an abstract datatype of tests. In this case, the **`(===)`** operator tests for equality like **`(==)`**, but returns a **`Property`** rather than a **`Bool`**. This is so that a more informative message can be displayed when tests fail.

Load this file into the Haskell interpreter with the shell command

```
ghci SimpleStart.hs
```

Now you can test each property in the interpreter, by running the commands

```
quickCheck prop_Reverse
quickCheck prop_Wrong
```

In each case, QuickCheck tries to run 100 tests (by default); in the first case, the tests succeed, while in the second case, the tests fail. Run the failing tests repeatedly—how much, and why, does the output vary?

Property-Driven Development

Test-driven development (TDD) is a well-known agile development method, in which no code is written until there is a failing test that requires it to be written. TDD thus encourages developers to write tests before code, and to keep code as simple as possible to pass the existing tests. This exercise will give you the experience of developing code in this way, where the tests are generated for you by QuickCheck, rather than written by hand. Along the way, you will have plenty of experience of discovering and fixing bugs with QuickCheck.

The module we will work with is `ISets.hs`: open it in your editor now. This module defines a datatype `ISet` of *interval sets*, representing sets of integers, in which sequences of consecutive integers are represented compactly. For example,

```
ISet [(1,10),(20,30)]
```

represents the set of integers

```
[1,2,3,4,5,6,7,8,9,10,20,21,22,23,24,25,26,27,28,29,30]
```

`ISet`s must satisfy an invariant, captured by the function `valid`, and can be converted to the list of integers they represent using the function `toList`. The module also contains a QuickCheck generator for the type `ISet`, which enables us to test properties that take `ISet`s as parameters. We shall return to the subject of writing generators later in the week.

Our goal is to implement the functions

```
member :: Int  -> ISet -> Bool
insert :: Int  -> ISet -> ISet
delete :: Int  -> ISet -> ISet
union  :: ISet -> ISet -> ISet
```

The module contains properties for testing each of these functions. How these properties work in detail is not important for this exercise, but (roughly speaking) the properties whose names end in `Valid` check that each operation constructs an `ISet` satisfying the invariant, while the properties whose names end in `Model` check that the `ISet` functions behave consistently with similar functions on lists of integers.

Load the module into ghci. You can test *all* the properties in the module by running

```
runTests
```

You will find that most properties fail (because the operations haven't been implemented yet). You can of course test individual properties by using `quickCheck` as usual.

The module already contains an implementation of the `member` function—and indeed, the tests for `member` pass. This implementation was constructed using property-driven development, as follows. The `member` function is tested by one property:

```
prop_Member x s =
  (x `member` s) === (x `elem` toList s)
```

which simply asserts that the result of `member` is consistent with membership of the list returned by `toList`. Because I expect to recurse on the list of pairs inside the `ISet`, I began by defining a wrapper function

```
member x (ISet xys) = member' x xys
```

which uses an auxiliary (recursive) function to do the real work. Then I defined

```
member' x xys = undefined
```

and tested `prop_Member`. Of course, the test failed:

```
0
ISet []
Exception thrown while showing test case:
  Prelude.undefined
```

This output tells me that `prop_Member` failed when `x` was `0` and `s` was `ISet []`; it failed (of course) because `undefined` was evaluated. So the next step is to add a case to the `member'` function which handles this example.

However, before doing so, *I added this test to the module*:

```
prop_MemberTest1 = prop_Member 0 (ISet [])
```

This defines a specialized version of the `prop_Member` property, that just tests this case; I wrote this code by copying and pasting the counterexample that QuickCheck found into the module.

Now I can refine the definition of `member'`:

```
member' x [] = False
```

The equation I added handles the test case QuickCheck found correctly; of course, I chose to write a more general equation that handles membership of *any* value in the empty set, not just zero. I can repeat the failing test, which now passes:

```
*ISets> quickCheck prop_MemberTest1
+++ OK, passed 1 test.
```

But if I test `prop_Member`, then I find another missing case:

```
0
ISet [(0,0)]
Exception thrown while showing test case:
  ISets.hs:38:1-20: Non-exhaustive patterns in function
member'
```

This tells me that the code for `member'` does not handle the case `member’ 0 [(0,0)]`, and so I can add `prop_MemberTest2` to test *this* case, and continue—for example, by extending the definition of `member'` like this:

```
member' x [] = False
member' x (ISet [(x’,y’)]) =
  x==x’
```

This makes `prop_MemberTest2` pass, but of course, QuickCheck finds another failing example. The file you have been given contains five test cases that each failed at some point during my development.

The reason for saving failed tests in the code is so that we can be sure that the change we made to the code actually fixed this bug. If the bug is indeed fixed, then running `quickCheck` on `prop_Member` will show us a *different* failing case (until the code works), but this *might* be because QuickCheck randomly found a different example. Only be repeating exactly the same test can we be sure that the bug is actually fixed, and this is why these test cases are worth saving.

The `ISets` module already contains properties for testing `insert`, `delete`, and `union`, along with wrapper functions and auxiliary functions defined to be `undefined`. See if you can follow the approach described above to complete these definitions. Try to follow the “test first” philosophy: write no code that is not needed to make the test case you are currently working with pass. When you are done, be sure to use `runTests` to test all the properties in the module—there may be one you forgot about!