Exercise (Day 5): Labelling and Tuning Tests

Tuning Tests for Interval Sets

If you completed the **ISets** exercise on Day 3, congratulations, you have a chance to reuse your code now! If not, you may start from the provided version of **ISets**, which contains a complete implementation.

Today we will focus on **prop_DeleteModel**, which tests that the delete function behaves according to the model (as a list of integers). Add labelling to this property, to measure:

1. How often the value **x** being deleted is actually present in the set?
2. In cases when **x** *is* present in the set, how often each of the following cases arises:
    a. **x** is the *first* element of some interval (i.e. **x**+1 is also in the set, but **x**-1 is not).
    b. **x** is the *last* element of some interval.
    c. **x** is the *only* element of an interval.
    d. **x** is a *middle* element of some interval (neither the first, nor the last).
3. In cases when **x** is present in the set how often it is in the *first*, *last*, or a *middle* interval of the set (neither first nor last).

Attach these labels to test cases, and use **labelledExamples** to check that your labelling code is correct. (For even more refined information, you could use labels that contain *all* of the above information—such as "only element, middle interval"—by concatenating the strings you used for each type of label).

Measure the distribution of tests:

quickCheck . withMaxSuccess 10000 $ prop_DeleteModel

Is it satisfactory?

Using the **forAll** combinator from QuickCheck, we can supply a custom generator for a property:

forAll :: Gen a -> (a -> Property) -> Property

Write a generator

probablyElement :: Arbitrary a => [a] -> Gen a

which generates a random value of type **a**, that is *more likely* to be an element of the given list. (Hint: consider using **arbitrary**, **elements**, and **oneof** in some combination).

Can you use **probablyElement** in combination with **forAll** to improve the distribution of tests in **prop_DeleteModel**?

Tuning the Registry Tests

Another version of **RegistryModel.hs** is provided with today's exercises. This version has been extended to collect data on the number of threads in the registry, and to classify calls according to whether they succeeded or failed, and (in the case of **register**), *why* the call failed. (The classification and labelling code is in the **monitoring** method of the **StateModel** class).

Generate labelled examples from **prop_Registry** and read through them. Do any surprise you? Are any calls incorrectly labelled? If so, adjust the code in **monitoring** to label them correctly.

Inspect the distribution of call labels (in the table Outcomes). Some types of call may appear quite rarely. Use

quickCheck . checkCoverage $ prop_Registry

to see whether the stated coverage requirements (that each type of call appear at least 4% of the time) are satisfied. If not, *tune* the test case generation so that they are. You can do so in two ways:

- Adjust the weights in the call of **frequency** in **arbitraryAction**, to generate some types of action more often than others.
- Change the way the name arguments to **register** and **unregister** are generated.

The second is more difficult: try it only if adjusting the weights does not succeed.