REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Shrinking Failing Tests

A problem with randomly creating data is that the results can be a bit noisy. Take this example which tries to verify that *lists:delete/2* will not leave an element in a list.

It asserts that when you call *lists:delete/2* on a list that the value which you removed will no longer be in the list.

Delete from list

```
-module(delete_from_list).
-include_lib("proper/include/proper.hrl").
-include_lib("eunit/include/eunit.hrl").
-compile(export_all).



prop_not_in_list() ->
    ?FORALL({El, List},
            {integer(), list(integer())},
            not(lists:member(El, lists:delete(El, List)))).
```

However the standard lists:delete/2 function will only remove the first value it finds, leaving any others (Yes, this is the documented way that the function works). Depending on the state of the random number generator it may find an error condition like this one: .counter example

```
{2,[-53,39,2,11,6,72,-6,-10,-24,0,-14,-14,-13,18,17,2,31,15,71,-6,10]}
```

The problem is that if you don't know that the issue is that the key to be removed **2** is there twice it is not obvious. Maybe you will figure it out after staring at it for a few minutes, but it is also possible that you would think that the problem is something else entirely.

To help you QuickCheck will engage in what is called **Shrinking**. It will take that initial counter example and try to simplify it. After shirking you may come up with this counter example. In this example there are only two items in the list so you can tell that none of the other elements contributed to the error. At this point you will notice that *lists:delete/2* won't remove the second instance of the 0. (And then go read the manual page which would tell you that).

counter example after shrinking

```
{0,[0,0]}
```

1.1 How Shrinking Works

Shrinking can be seen as depth first search of a tree. You start with a failing example. Then you apply one Shrinking Operation [shrink-data]. Then apply the property again to the smaller data set. If it still fails repeat the process.

1.2 How data types shrink

The three different QuickCheck implementations each shrink the data a bit differently. But the basics will be the same from all of them.

Table 1.1: Basic Shrinking

Type	Operation
Integer	Move Closer to 0
Float	Move Closer to 0
Atom (Proper)	Drop Characters
Bitstring	Convert 1's to 0's or drop characters
Function	Return Simpler values
Tuple	Simplify Elements
Loose Tuple	Drop Elements or Simplify Elements
Vector	Simplify Elements
List	Drop Elements or Simplify Elements
Union/Elements	Move closer to head of the list or Simplify elements
oneof(PropEr)	Move closer to head of the list or Simplify elements
oneof (eqc)	Simplify Elements
Literal Term	Doesn't shrink

1.3 Testing your generator's shrinking

If you wish to know how a generator will shrink all three QuickCheck's include a *sampleshrink/1* function, which will take a generator, create some data and shrink it. In PropEr it is in the *proper_gen* module, for Triq it is in *triq_dom* and for eqc it is in *eqc_gen*.

If you are just using the built in generators then you can assume that they will shrink in sensible ways. But when you start building your own generators, or composing generators in complex ways you may wish to see what is going in the case things don't do what you expect.

This example, shown in PropEr, shows how elements shrink, it starts with a longer list (in this case length 13), and then starts removing items. The second argument to sampleshrink is the size parameter which lets the generator know how large a data structure to build.

shrink example

```
13> proper_gen:sampleshrink(proper_types:list(proper_types:oneof(words:list())),30).
[<<"hundred-feathered">>,<<"school-magisterial">>,<<"swollen-cheeked">>,
 <<"sponge-leaved">>,<<"plate printer">>,<<"flax olive">>,
 <<"pressure regulator">>,<<"urinogenital sinus">>,<<"shop-made">>,
 <<"fire underwriter">>,<<"knee punch">>,<<"cockatoo bush">>,
 <<"tear-reconciled">>]
[<<"hundred-feathered">>,<<"school-magisterial">>,<<"swollen-cheeked">>,
 <<"sponge-leaved">>,<<"plate printer">>,<<"flax olive">>,
 <<"pressure regulator">>]
[<<"hundred-feathered">>,<<"school-magisterial">>,<<"swollen-cheeked">>,
 <<"sponge-leaved">>]
[<<"hundred-feathered">>,<<"school-magisterial">>]
ok
```

1.4 Implementing custom shrinking

If you want your data to shrink in an unconventional way for some reason there are ways to do that. Normally a number will shrink to 0, but in some cases you might not want it to, Say you wish to generate a percentage, that will normally be close to 100% not to 0.

To do this use a ?LET/3 [?] macro to create a custom generator. In this example the code generates an integer between 1 and 100, then subtracts that from 100. When this goes to shrink the *integer/2* generator will tend to produce smaller numbers, which will of course result in the *percent/0* generator making smaller numbers.

Shrink Percentage

```
percent() ->
    ?LET(X, integer(0, 100), 100 - X).
```

If we sample what this code does (with PropEr) we will get this range

```
75> proper_gen:sample(proper_examples:percent()).
43
50
81
94
55
80
32
67
71
48
73
ok
```

If we ask PropEr to shrink it it will give us this, and in fact every time I ran it, it ended at 100.

```
76> proper_gen:sampleshrink(proper_examples:percent()).
90
100
ok
```

1.5 The *?SHRINK*/2 macro

The *?SHRINK/2* macro allows the programmer to specify alternative generators to be used in the shrinking process.

In this example we generate a float, but when shrinking switch to an integer when it is shrinking.

Shrink

```
shrink_ex() ->
    ?SHRINK(float(), [int()]).
```

```
6> proper_gen:sampleshrink(proper_examples:shrink_ex()).
-2.6776036684953928
10
ok
```

You can also specify more that one shrink generator. The *?SHRINK* macro will tend to use the arguments closer to the front of the list.

If you want to combine *?SHRINK2* with *?LET/3* you can use *?LETSHRINK/3*

Note
More examples will be shown in [?] and [?].