

Part of the "The Return of the EDFH" series ([link](#))

Generating interesting inputs for property-based testing

And how to classify them

15 Feb 2021

In the previous post we attempted to define some properties for a run-length encoding (RLE) implementation, but got stuck because the random values being generated by FsCheck were not very useful.

In this post we'll look at a couple of ways of generating "interesting" inputs, and how to observe them so that we can be sure that they are indeed interesting.

Observing the generated data

The first thing we should do is add some kind of monitoring to see how many of the inputs are interesting.

So what is an "interesting" input? For this scenario, it's a string that has some runs in it. Which means that a string consisting of random characters like this...

```
%q6,NDUwm9~ 8I?a-ruc(@6Gi_+pT;1SdZ|H
```

...is not very interesting as input for an RLE implementation.

Without trying to reimplement the RLE logic, one way to determine whether there are runs is to see if the number of distinct characters is much less than the length of the string. If this is true, then by the pigeonhole principle there *must* be duplicates of some character. This doesn't *ensure* that there are runs, but if we make the difference large enough, most of the "interesting" inputs will have runs.

So here's the definition of our `isInterestingString` function:

```
let isInterestingString inputStr =  
    if System.String.IsNullOrEmpty inputStr then  
        false
```

```

else
  let distinctChars =
    inputStr
    |> Seq.countBy id
    |> Seq.length
    distinctChars <= (inputStr.Length / 2)

```

And if we test it, we can see that it works pretty well.

```

isInterestingString "" //=> false
isInterestingString "aa" //=> true
isInterestingString "abc" //=> false
isInterestingString "aabbccc" //=> true
isInterestingString "aabaaac" //=> true
isInterestingString "abcabc" //=> true (but no runs)

```

To monitor whether an input is interesting, we will use the FsCheck function **Prop.classify**.

Prop.classify is just one of a number of functions for working with properties. More on properties at the FsCheck documentation. Or check out the complete API.

To test all this, let's create a dummy property **propIsInterestingString** which we can use to monitor the input generated by FsCheck. The actual property test itself should always succeed, so we'll just use **true**. Here's the code:

```

let propIsInterestingString input =
  let isInterestingInput = isInterestingString input

  true // we don't care about the actual test
  |> Prop.classify (not isInterestingInput) "not interesting"
  |> Prop.classify isInterestingInput "interesting"

```

And now let's check it:

```

FsCheck.Check.Quick propIsInterestingString
// Ok, passed 100 tests (100% not interesting).

```

We find that 100% of the inputs are not interesting. So we need to make better inputs!

Generating interesting strings, part 1

One way to do this is to use a filter to remove all strings that are not interesting. But that would be horrendously inefficient, as interesting strings are extremely rare.

Instead, let's *generate* interesting strings. For our first attempt, we'll start with something very simple: we will generate a list of 'a' characters and a list of 'b' characters, and then concatenate the two lists, giving us some nice runs.

In order to do this, we will build our own generator (see earlier discussion of generators and shrinkers). FsCheck provides a useful set of functions for making generators, such as **Gen.constant** to generate a constant, **Gen.choose** to pick a random number from an interval, and **Gen.elements** to pick a random element from a list. Once you have a basic generator, you can **map** and **filter** its output, and also combine multiple generators with **map2**, **oneOf**, etc.

For more on working with generators, see the FsCheck documentation.

- Overview of using generators
- The generator API

So, here's our code using the generators:

```
let arbTwoCharString =
  // helper function to create strings from a list of chars
  let listToString chars =
    chars |> List.toArray |> System.String

  // random lists of 'a's and 'b's
  let genListA = Gen.constant 'a' |> Gen.listOf
  let genListB = Gen.constant 'b' |> Gen.listOf

  (genListA, genListB)
  ||> Gen.map2 (fun listA listB -> listA @ listB )
  |> Gen.map listToString
  |> Arb.fromGen
```

We generate a list of 'a' characters and list of 'b' characters, then use **map2** to concatenate them, and then convert the resulting list into a string. As the very last step, we build an **Arbitrary** from the generator, which is what we will need for the testing phase. We're not providing a custom shrinker right now.

Next, let's sample some random strings from our new generator to see what they look like:

```
arbTwoCharString.Generator |> Gen.sample 10 10
(*
[ "aaabbbbbbb"; "aaaaaaaaabb"; "b"; "abbbbbbbbb";
  "aaabbbb"; "bbbbbb"; "aaaaaaaaabbbbbbb";
  "a"; "aabbbb"; "aaaaabbbbbbbbb"]
*)
```

That looks pretty good. Most of the strings have runs, just as we want.

Now we can apply this generator to the `propIsInterestingString` property we created earlier. We will use `Prop.forAll` to construct a new property using the custom generator, and then test the new property with `Check.Quick` in the usual way.

```
// make a new property from the old one, with input from our
generator
let prop = Prop.forAll arbTwoCharString propIsInterestingString
// check it
Check.Quick prop

(*
Ok, passed 100 tests.
97% interesting.
3% not interesting.
*)
```

And this output is much better! Almost all the inputs are interesting.

Generating interesting strings, part 2

The strings we’re generating have at most two runs, which is not very representative of the real strings that we want to run-length encode. We could enhance our generator to include multiple lists of characters, but it gets a little complicated, so let’s approach this problem from a completely different direction.

One of the most common uses for run-length encoding is to compress images. We can think of a monochrome image as an array of 0s and 1s, with 1 representing a black pixel. Now let’s consider an image with only a few black pixels, which in turn means lots of long runs of white pixels, perfect as input for our tests.

How can we generate such “images”? How about starting with an array of white pixels and randomly flipping some of them to black?


```
type RleImpl = string -> (char*int) list
```

```
let propUsesAllCharacters (impl:RleImpl) inputStr =  
  let output = impl inputStr  
  let expected =  
    if System.String.IsNullOrEmpty inputStr then  
      []  
    else  
      inputStr  
      |> Seq.distinct  
      |> Seq.toList  
  let actual =  
    output  
    |> Seq.map fst  
    |> Seq.distinct  
    |> Seq.toList  
  expected = actual
```

Note: As implemented, this property is actually *stronger* than “contains all the characters from the input”. If we wanted that, we should convert **expected** and **actual** into unordered sets before comparing them. But because we are leaving them as lists, the property as implemented is actually “contains all the characters from the input *and in the same order*”.

Prop #2: Two adjacent characters in the output cannot be the same

```
let propAdjacentCharactersAreNotSame (impl:RleImpl) inputStr =  
  let output = impl inputStr  
  let actual =  
    output  
    |> Seq.map fst  
    |> Seq.toList  
  let expected =  
    actual  
    |> removeDupAdjacentChars // should have no effect  
  expected = actual // should be the same
```

Reminder: The **removeDupAdjacentChars** function in this code was defined in the previous post.

Prop #3: The sum of the run lengths in the output must equal the length of the input

```
let propRunLengthSum_eq_inputLength (impl:RleImpl) inputStr =  
  let output = impl inputStr
```

```
let expected = inputStr.Length
let actual = output |> List.sumBy snd
expected = actual // should be the same
```

Here, we simply sum the second part of each `(char, run-length)` tuple.

Prop #4: If the input is reversed, the output must also be reversed

```
/// Helper to reverse strings
let strRev (str:string) =
    str
    |> Seq.rev
    |> Seq.toArray
    |> System.String

let propInputReversed_implies_outputReversed (impl:RleImpl)
inputStr =
    // original
    let output1 =
        inputStr |> impl

    // reversed
    let output2 =
        inputStr |> strRev |> impl

List.rev output1 = output2 // should be the same
```

Combining the properties

Finally we can combine all four properties into a single compound property. Each of the four sub-properties is given a label with `@|` so that when the compound property fails, we know which sub-property caused the failure.

```
let propRle (impl:RleImpl) inputStr =
    let prop1 =
        propUsesAllCharacters impl inputStr
        |@ "propUsesAllCharacters"
    let prop2 =
        propAdjacentCharactersAreNotSame impl inputStr
        |@ "propAdjacentCharactersAreNotSame"
    let prop3 =
        propRunLengthSum_eq_inputLength impl inputStr
        |@ "propRunLengthSum_eq_inputLength"
    let prop4 =
        propInputReversed_implies_outputReversed impl inputStr
```



```
|@ "propInputReversed_implies_outputReversed"
```

```
// combine them
```

```
prop1 .&. prop2 .&. prop3 .&. prop4
```

Testing the EDFH implementations

Now finally, we can test the EDFH implementations against the compound property.

The first EDFH implementation simply returned an empty list.

```
/// Return an empty list
```

```
let rle_empty (inputStr:string) : (char*int) list =  
    []
```

We would expect it to fail on the first property: “The output must contain all the characters from the input”.

```
let prop = Prop.forAll arbPixels (propRle rle_empty)
```

```
// -- expect to fail on propUsesAllCharacters
```

```
// check it
```

```
Check.Quick prop
```

```
(*
```

```
Falsifiable, after 1 test (0 shrinks)
```

```
Label of failing property: propUsesAllCharacters
```

```
*)
```

And indeed it does.

EDFH implementation #2

The next EDFH implementation simply returned each char as its own run, with a run length of 1.

```
/// Return each char with count 1
```

```
let rle_allChars inputStr =
```

```
    // add null check
```

```
    if System.String.IsNullOrEmpty inputStr then
```

```
        []
```

```
    else
```

```
        inputStr
```

```
|> Seq.toList
|> List.map (fun ch -> (ch,1))
```

We would expect it to fail on the second property: “No two adjacent characters in the output can be the same”.

```
let prop = Prop.forAll arbPixels (propRle rle_allChars)
// -- expect to fail on propAdjacentCharactersAreNotSame

// check it
Check.Quick prop
(*
Falsifiable, after 1 test (0 shrinks)
Label of failing property: propAdjacentCharactersAreNotSame
*)
```

And indeed it does.

EDFH implementation #3

The third EDFH implementation avoided the duplicate characters issues by doing a **distinct** first.

```
let rle_distinct inputStr =
// add null check
if System.String.IsNullOrEmpty inputStr then
[]
else
inputStr
|> Seq.distinct
|> Seq.toList
|> List.map (fun ch -> (ch,1))
```

It would pass the second property: “No two adjacent characters in the output can be the same” but we would expect it to fail on the third property: “The sum of the run lengths in the output must equal the total length of the input”.

```
let prop = Prop.forAll arbPixels (propRle rle_distinct)
// -- expect to fail on propRunLengthSum_eq_inputLength

// check it
Check.Quick prop
(*
Falsifiable, after 1 test (0 shrinks)
```

```
Label of failing property: propRunLengthSum_eq_inputLength
*)
```

And it does!

EDFH implementation #4

The last EDFH implementation avoided the duplicate characters issues *and* got the overall run lengths right by doing a **groupBy** operation.

```
let rle_countBy inputStr =
  if System.String.IsNullOrEmpty inputStr then
    []
  else
    inputStr
    |> Seq.countBy id
    |> Seq.toList
```

And this is why we added a fourth property to catch this: “If the input is reversed, the output must also be reversed”.

```
let prop = Prop.forAll arbPixels (propRle rle_countBy)
// -- expect to fail on propInputReversed_implies_outputReversed

// check it
Check.Quick prop
(*
Falsifiable, after 1 test (0 shrinks)
Label of failing property: propInputReversed_implies_outputReversed
*)
```

And it fails as expected.

Testing the correct implementations

After all those bad implementations, let’s look at some correct implementations. We can use our four properties to have confidence that a particular implementation is correct.

Correct implementation #1

Our first implementation will use recursion. It will strip off the run of the first character, leaving a smaller list. It will then apply the same logic to that smaller list.

```

let rle_recursive inputStr =

  // inner recursive function
  let rec loop input =
    match input with
    | [] -> []
    | head::_ ->
      [
        // get a run
        let runLength = List.length (List.takeWhile ((=) head) input)
        // return it
        yield head,runLength
        // skip the run and repeat
        yield! loop (List.skip runLength input)
      ]

  // main
  inputStr |> Seq.toList |> loop

```

If we test it, it seems to work as expected.

```

rle_recursive "aaaabbbcca"
// [('a', 4); ('b', 3); ('c', 2); ('a', 1)]

```

But does it really? Let's run it through the property checker to be sure:

```

let prop = Prop.forAll arbPixels (propRle rle_recursive)
// -- expect it to not fail

// check it
Check.Quick prop
(*
Ok, passed 100 tests.
*)

```

And yes – no properties failed!

Correct implementation #2

The recursive implementation above might have some problems with input strings that are very large. First of all, the inner loop is not tail recursive, so the stack might overflow. Also, by continually creating sub-lists it is creating lots of garbage, which can affect performance.

An alternative approach is to iterate over the input once, using `Seq.fold`. Here's a basic implementation:

```
let rle_fold inputStr =
    // This implementation iterates over the list
    // using the 'folder' function and accumulates
    // into 'acc'

    // helper
    let folder (currChar,currCount,acc) inputChar =
        if currChar <> inputChar then
            // push old run onto accumulator
            let acc' = (currChar,currCount) :: acc
            // start new run
            (inputChar,1,acc')
        else
            // same run, so increment count
            (currChar,currCount+1,acc)

    // helper
    let toFinalList (currChar,currCount,acc) =
        // push final run onto acc
        (currChar,currCount) :: acc
        |> List.rev

    // main
    if System.String.IsNullOrEmpty inputStr then
        []
    else
        let head = inputStr.[0]
        let tail = inputStr.[1..inputStr.Length-1]
        let initialState = (head,1,[])
        tail
        |> Seq.fold folder initialState
        |> toFinalList
```

We could optimize this more by having a mutable accumulator, using arrays rather than lists, and so on. But it's good enough to demonstrate the principle.

Here's some interactive testing to make sure that it works as expected:

```
rle_fold "" //=> []
rle_fold "a" //=> [('a',1)]
rle_fold "aa" //=> [('a',2)]
rle_fold "ab" //=> [('a',1); ('b',1)]
rle_fold "aab" //=> [('a',2); ('b',1)]
```

```
rle_fold "abb" //=> [('a',1); ('b',2)]
rle_fold "aaaabbbcca"
//=> [('a',4); ('b',3); ('c',2); ('a',1)]
```

But of course, using the property checker is a better way to be sure:

```
let prop = Prop.forAll arbPixels (propRle rle_fold)
// -- expect it to not fail

// check it
Check.Quick prop
(*
Ok, passed 100 tests.
*)
```

And it does pass all the tests.

So the logic is correct, but as noted above, we should also do some performance testing on large inputs and optimization before we can consider this production ready. That's a whole other topic!

Optimization can sometimes introduce bugs, but now that we have these properties, we can test the optimized code in the same way and be confident that any errors will be detected immediately.

Summary

In this post we first found a way to generate “interesting” inputs, and then using these inputs, ran the properties from last time against the EDFH implementations. They all failed! And then we defined two correct implementations that did satisfy all the properties.

So are we done now? No. It turns out that the EDFH can still create an implementation that satisfies all the properties! To finally defeat the EDFH, we'll need to do better.

That will be the topic of the next installment.


Source code used in this post is available here.

← 1. The Return of the Enterprise Developer From Hell

3. The EDFH is defeated once again →

The "The Return of the EDFH" series

- The Return of the Enterprise Developer From Hell
More malicious compliance, more property-based testing
- **Generating interesting inputs for property-based testing**
And how to classify them
- The EDFH is defeated once again

Written by ScottW. Found a typo or error? Follow me on  Twitter.

Comments