Property-Based Testing

Patrick Stevens

G-Research

Doge Conf 2019

Let's have something to test

Interval set: a space-efficient set of integers

Defining example:

$$\{1, 2, 3, 4, 8, 9, 10\}$$

$$[(1, 4), (8, 10)]$$

Public API

```
type IntervalSet

[<RequireQualifiedAccess>]
module IntervalSet =
    val empty : IntervalSet
    val add : int -> IntervalSet -> IntervalSet
    val contains : int -> IntervalSet -> bool
```

Implementation

This is a presentation about testing.
Yes, there will be a bug somewhere.

Data structure

```
type private Interval =
    {
        Min : int
        Max : int
    }

type IntervalSet = private IntervalSet of Interval list
```

Implementation: empty

```
[<RequireQualifiedAccess>]
module IntervalSet =
    let empty = IntervalSet []
```

Implementation: insertion

```
let private rec add' (a : int) (ls : Interval list) =
    match ls with
    | [] -> [{ Min = a ; Max = a }]
    | _ -> ...
```

Implementation: insertion

```
let private rec add' (a : int) (ls : Interval list) =
    match ls with
    | [] -> ...
    | interval :: is ->
        if interval.Min <= a && a <= interval.Max then
            ls // no need to add, it's already there
        else ...
```

Implementation: insertion

```
let private rec add' (a : int) (ls : Interval list) =
    match ls with
    | [] -> ...
    | interval :: is ->
        if // already there...
        elif interval.Min - 1 = a then
            { interval with Min = interval.Min - 1 }
            :: is // augment this interval to contain a
        elif interval.Max + 1 = a then
            { interval with Max = interval.Max + 1 }
            :: is // augment this interval to contain a
        ...
```

Implementation: insertion

```
let private rec add' (a : int) (ls : Interval list) =
    match ls with
    | [] -> ...
    | interval :: is ->
        if // already there...
        elif // can be added to this interval...
        else // can't add it here; recurse
            add' a is
```

Implementation: insertion

```
[<RequireQualifiedAccess>]
module IntervalSet =
    let add (a : int) (IntervalSet intervals) =
        add' a intervals
        |> IntervalSet
```

Implementation: containment

```
let rec contains (a : int) (IntervalSet ls) =
    ls
    |> List.tryFind (fun interval ->
        interval.Min <= a && a <= interval.Max)
    |> Option.isSome
```

Start testing!

Helper function for tests:

```
let create (is : int list) : IntervalSet =
    is
    |> List.fold
        (fun set i -> IntervalSet.add i set)
        IntervalSet.empty
```

What can we test?

We should test some different lists and their resulting IntervalSets.

- $[3; 4]$ contains 5? (No.)
- $[3; 5]$ contains 5? (Yes.)
- $[3; 4; 5]$ contains 4? (Yes.)

The test cases

```
create [3; 4]
|> IntervalSet.contains 5
|> shouldEqual false

create [3; 5]
|> IntervalSet.contains 5
|> shouldEqual true

create [3; 4; 5]
|> IntervalSet.contains 4
|> shouldEqual true
```

The test cases

```
create [3; 4]
|> IntervalSet.contains 5
|> shouldEqual false

create [3; 5]
|> IntervalSet.contains 5
|> shouldEqual true

create [3; 4; 5]
|> IntervalSet.contains 4
|> shouldEqual true
```

Hooray, the tests pass!

... but is it right?

... but is it right?

I don't know!

- I'm lazy
- I'm stupid
- I hate testing

Isn't there a better way?

FsCheck can help!

Sneak peek: FsCheck will tell us that this implementation is wrong.

```
Falsifiable, after 35 tests (15 shrinks)
  (StGen 9514417537,296661223)):
Original:
[0; 0; 0; 0; 0; 0; 0; 12; 1; -2; 0; 0; 0; 0; 0; 0; 0]
12
Shrunk:
[12; 0]
12
```

FsCheck's test

The last two lines FsCheck gave us were:

```
[12; 0]
12
```

FsCheck's test

The last two lines FsCheck gave us were:

```
[12; 0]
12
```

FsCheck found this test:

```
create [12; 0]
|> IntervalSet.contains 12
|> shouldEqual true
```

Why do you test?

- Your program does what you want it to.
- Your program doesn't do what you don't want it to.

How do you normally test?

1. Come up with examples.
2. Work out what your program should do on those examples.
3. Run the program and check it did what you wanted.

But what are you *really* doing?

You're testing *properties* through *representative examples*.

Why not just *test properties*?

FsCheck tests properties automatically

- Find edge cases
- Find large, complicated cases
- Shrink large cases automatically
- Make sure you can repeat any failures (the TDD way!)

The failing property for IntervalSet

- Create an IntervalSet from a list of integers...
- then check for containment...
- should be the same as checking the original list.

The failing property for IntervalSet, in code

```
let property (ints : int list) (toCheck : int) : bool =
    create ints
    |> IntervalSet.contains toCheck
    |> (=) (List.contains doesContain ints)
```

Invoke FsCheck

```
open FsCheck

let property (ints : int list) (toCheck : int) : bool =
    create ints
    |> IntervalSet.contains toCheck
    |> (=) (List.contains doesContain ints)

[<Test>]
let testProperty () =
    Check.QuickThrowOnFailure property
```

FsCheck's output, revisited

```
Falsifiable, after 35 tests (15 shrinks)
  (StGen 9514417537,296661223)):
Original:
[0; 0; 0; 0; 0; 0; 0; 12; 1; -2; 0; 0; 0; 0; 0; 0; 0]
12
Shrunk:
[12; 0]
12
```

FsCheck constructed 35 tests before finding a failure. It then shrank the test case to the smallest failure it could find.

Advantages

- No thought required!
- Perfectly reproducible
- Edge cases automatically examined closely
- Randomised testing increases coverage

(Make sure you explicitly test any failures FsCheck finds, so that nothing is lost to the mists of time!)

The bug

```
let private rec add' (a : int) (ls : Interval list) =
    match ls with
    | [] -> [{ Min = a ; Max = a }]
    | interval :: is ->
        if (* contains *) then
            ls
        elif (* is 1 below *) then
        elif (* is 1 above *) then
        else
            add' a is // <-- Oh no!
```

The fix

```
let private rec add' (a : int) (ls : Interval list) =
    match ls with
    | [] -> [{ Min = a ; Max = a }]
    | interval :: is ->
        if (* contains *) then
            ls
        elif (* is 1 below *) then
        elif (* is 1 above *) then
        else
            interval :: add' a is
```

FsCheck is happy

```
Ok, passed 100 tests.
```

Serialisers: the hello-world of black-box testing

A serialiser is defined by a property:

Writing an object, then reading it in, gives the original object.

Signature of a serialiser

```
[<RequireQualifiedAccess>]
module FancyThing =
    val toString : FancyThing -> string
    val parse : string -> FancyThing option
```

Test the serialiser

```
[<Test>]
let roundTripTest () =
    let property (t : FancyThing) : bool =
        t
        |> FancyThing.toString
        |> FancyThing.parse
        |> (=) (Some t)
    Check.QuickThrowOnFailure property
```

What FsCheck gave us

Without needing to know anything about the implementation, FsCheck was still able to produce useful tests!

- Testing with FsCheck

Metatesting

A technique you should use to make your testing more effective.

How can we be sure we tested enough?

Recall the property:

```
let property (ints : int list) (toCheck : int) : bool =
    create ints
    |> IntervalSet.contains toCheck
    |> (=) (List.contains doesContain ints)
```

How can we be sure we tested enough?

Recall the property:

```
let property (ints : int list) (toCheck : int) : bool =
    create ints
    |> IntervalSet.contains toCheck
    |> (=) (List.contains doesContain ints)
```

By fluke (or my incompetence), FsCheck might never generate a "yes, does contain" case.

Gather metrics

F# is impure and side-effectful, so it's extremely easy to gather metrics.

(Instrumentation is an example where purity really does make things harder!)

Gather metrics in the property

```
let property
    (positives : int ref) (negatives : int ref)
    (ints : int list)
    (toCheck : int)
    : bool
    =
    let contains = List.contains doesContain ints
    if contains then
        incr positives
    else
        incr negatives

    create ints
    |> IntervalSet.contains toCheck
    |> (=) contains
```

Invoke the augmented test

```
[<Test>]
let test () =
    let pos = ref 0
    let neg = ref 0
    Check.QuickThrowOnFailure (property pos neg)

    let pos = pos.Value
    let neg = neg.Value

    pos |> shouldBeGreaterThan 0
    neg |> shouldBeGreaterThan 0

    (float pos) / (float pos + float neg)
    |> shouldBeGreaterThan 0.1
```

At least a tenth of the time, we want to be hitting positive cases, but . . .

The test is a bit unreliable!

```
Expected: 0.1
Actual: 0.08

at FsUnitTyped.TopLevelOperators.shouldBeGreaterThan
```

Manipulating the generated cases

We want to generate cases that aren't so often "look for something that's not in the list".
FsCheck gives us access to its *generators* for this purpose.

Generators

We will have the property remain the same, but tell FsCheck to generate different cases.
FsCheck has a number of built-in generators. It also has a computation expression to manipulate generators.

Generator of bounded integers

```
let someInts : Gen<int> = Gen.choose (-100, 100)
Gen.sample 0 5 someInts
// output: [57; -24; 67; -14; 77]
```

Generator of bounded even integers

```
let someInts : Gen<int> = Gen.choose (-100, 100)
let evenIntegers : Gen<int> = gen {
    let! (anyInt : int) = someInt
    return 2 * anyInt
}
Gen.sample 0 5 someInts
// output: [-190; -24; -194; -108; -112]
```

Size

```
let integers : Gen<int list> =
    Gen.sized (fun i ->
        Gen.choose (-100, 100)
        |> Gen.listOfLength i)
```

Generator that sometimes selects from a list

```
let integers : Gen<int list> = ...

let listAndElt : Gen<int * int list> = gen {
    let! (list : int list) = integers
```

Generator that sometimes selects from a list

```
let integers : Gen<int list> = ...

let listAndElt : Gen<int * int list> = gen {
    let! (list : int list) = integers

    let genFromList = Gen.elements list
```

Generator that sometimes selects from a list

```
let integers : Gen<int list> = ...

let listAndElt : Gen<int * int list> = gen {
    let! (list : int list) = integers

    let genFromList = Gen.elements list

    let genNotFromList =
        Gen.choose (-100, 100)
        |> Gen.filter (fun i -> not (List.contains i list))
```

Generator that sometimes selects from a list

```
let integers : Gen<int list> = ...

let listAndElt : Gen<int * int list> = gen {
    let! (list : int list) = integers

    let genFromList = Gen.elements list

    let genNotFromList =
        Gen.choose (-100, 100)
        |> Gen.filter (fun i -> not (List.contains i list))

    let! number = Gen.oneOf [genFromList ; genNotFromList]
    return (number, list)
}
```

Using this generator

The generator makes pairs of an integer and a list, where the integer is 50% likely to appear in the list.

```
[<Test>]
let test () =
    let (pos, neg) = (ref 0), (ref 0)

    (fun (list, elt) -> property pos neg elt list)
    |> Prop.forAll (Arb.fromGen listAndElt)
    |> Check.QuickThrowOnFailure

    let pos = pos.Value |> float
    let neg = neg.Value |> float

    pos / (pos + neg)
    |> shouldBeGreaterThan 0.1
```

```
Ok, passed 100 tests.
```

In fact we now have a positive case about 50% of the time.

Stateful systems

What about state?

Key idea: *describe what to do*, and then do it.

Example: a stream

Simple model: array and pointer.

Starting state, pointer at index 0

72	69	76	76	79	32	87	...

Seek to index 2

72	69	76	76	79	32	87	...

Write 88 at the current index

72	69	88	76	79	32	87	...

Imagine we can't access the implementation, but we want to test it anyway.

```
type Stream

[<RequireQualifiedAccess>]
module Stream =
    val uninitialised : unit -> Stream

    val read : Stream -> byte
    val write : Stream -> byte -> unit
    val seek : Stream -> int -> unit
    val currentIndex : Stream -> int
```

Some things to test

- Write then read
- Seek then get index
- Write, seek away, seek back, read

FsCheck will do these, and do them well, but they are all quite specific.
Isn't the point of FsCheck to take this drudgery away from us?

Obstacles to FsCheck

Why is FsCheck not helping here?

- Testing a mutable object
- No obvious immutable model to use
- How to generate random streams?
- Need shrinking not to interfere with itself

Answer: *describe* what to do, and *then* do it!

Answer: *describe* what to do, and *then* do it!

c.f. initial algebras

Domain model

```
type StreamInteraction =
| Write of byte
| Read
| Seek of int
| CurrentIndex

type TestCase = StreamInteraction list
```

```
[<RequireQualifiedAccess>]
module TestCase =
    let rec prepareStream
        (s : Stream)
        (instructions : TestCase)
        =
        for instruction in instructions do
            match instructions with
            | Write b ->
                Stream.write s b
            | Read ->
                Stream.read s |> ignore
            | Seek n ->
                Stream.seek s n
            | CurrentIndex ->
                Stream.currentIndex s |> ignore
```

Suddenly an immutable model appeared!

- By constructing test cases through their *descriptions*...
- we made an immutable model of the world.
- FsCheck can generate these things completely automatically!

Immediately useful...

This can be used with no further modification to check a very useful property:

```
[<Test>]
let doesNotCrash () =
    let property (instructions : StreamInteraction list) =
        let s = Stream.uninitialised ()
        TestCase.prepareStream s instructions
        true

    Check.QuickThrowOnFailure (property)
```

... and more useful with generalisation

But it really shines with just a little more work.

The index only changes when you expect it to

1. Make a generator for all interactions that shouldn't modify the index
2. Generate a list of interactions
3. Generate a list of interactions which don't modify the index
4. Concatenate
5. Execute, and assert that the index hasn't changed.

The gold standard: an immutable model

1. Define a map of "index" to "current value"
2. Interpret the interactions as acting on that map
3. Generate a list of interactions
4. Verify that the accessible output from the stream is indistinguishable from the output of the map.

Aside: model-based testing

This has a name: *model-based testing*.

Checking behaviour relative to a more easily specified model.

MBT is a subset of property-based testing.
The property is "the system behaves like the model does".
(Contrast: serialiser. Correctness defined by property, not model.)

FsCheck also has dedicated built-in support for MBT.

Sketch: Testing a UI

Test a UI by...

1. Identifying the actions you want to test;
2. Generating lists of actions;
3. Applying the actions;
4. Checking the final state is as expected given the actions.

Small properties, incremental addition

The UI example is good:

- Enormous space of possible actions
- Some actions very hard to test?
- Don't care uniformly about correctness

Don't need to test everything!

A restricted search space

If you're struggling to find properties, restrict the search space!
If you're struggling to define correctness, restrict the search space!

Cut down to a small subset of user interactions, and you will find properties.

Doge Analytics

(Here is where I waffle, because these slides are on public Github)

Why you should use PBT

Property-based testing...

- is easy
- exists in many languages (Python, F#, Haskell, . . .)
- can be added incrementally
- is as comprehensive as you want
- tests anything (black-box or otherwise)