Fox Documentation

Release 1.0.0

Jeff Hui

December 16, 2014

Everything about Fox, the property-based testing tool for Objective-C.

Besides this documentation, you can also view the source on GitHub.

CHAPTER 1

Getting Started

New to Fox? Or just wanting to have a taste of it? Start here.

- *Overview* - What is Fox?
- *Installation* - How to get set up.
- *Tutorial* - Get a feel for using Fox.

CHAPTER 2

Generators

Generators are semi-random data producers that are the core to Fox's capabilities. Follow the links below to learn more in detail.

- *Overview*
- *Built-in Generators Reference*
- *Building Custom Generators*
- *Writing Generators with Custom Shrinking*

CHAPTER 3

The Runner

All the guts around configuring and executing Fox's verification of properties.

3.1 What is Fox?

Fox is a port of the property-based testing tool test.check for Objective-C and Swift.

QuickCheck inspired Property-Based Testing Tools are test generators. These tools allow you to describe a property of your program that should always hold true instead of writing hand-crafted test cases. A pseudo-code example would be:

```
// pseudocode: A property for the sort() function
property := ForAll(xs where xs is an Array of Unsigned Integers){
    // perform action
    sorted_numbers := sort(xs)
    // verify that sorted_numbers is in ascending order
    i := 0
    for n in sorted_numbers {
        if i <= n {
            i = n
        } else {
            return FAILURE
        }
    }
    return SUCCESS
}
```

This example tests a sort function. Fox will generate tests based on the requirements of `xs` (any array of unsigned integers) to find a failing example that causes a FAILURE.

In the mathematical sense, Fox is a weak proof checker of a property where the tool tries to assert the property is valid by randomly generating data to find a counter-example.

3.1.1 Shrinking Failures

A benefit of Fox over purely random data generation is Shrinking. An informed random generation is done by size. This allows the tool to reduce the counter-example to a smaller data set without having a user manually separate thes signal from the noise in the randomly generated data.

For example, if a `sort()` implementation failed with an exception when reading 9s. Fox might generate this value to provoke the failure:

```
xs = @[@1, @5, @9, @3, @5] // fails
```

And then proceed to shrink `xs` by trying smaller permutations:

```
xs = @[@5, @9, @3, @5] // still fails
xs = @[@9, @3, @5] // fails
xs = @[@3, @5] // passed
xs = @[@9, @5] // fails
xs = @[@9] // fails
```

Fox does this automatically whenever a failure occurs. This is valuable instead of having to manually debug a failure when random data generation is used.

Ready to get started? *Install Fox*.

3.2 Installation

Fox can be installed in multiple ways. If you don't have a preference, install via git submodule.

Fox honors *semantic versioning* as humanly possible. If you're unsure if a given update is backwards incompatible with your usage. Check out the releases.

3.2.1 Manually (Git Submodule)

Add Fox as a submodule to your project:

```
$ git submodule add https://github.com/jeffh/Fox.git Externals/Fox
```

If you don't want bleeding edge, check out the particular tag of the version:

```
$ cd Externals/Fox
$ git checkout v1.0.1
```

Add `Fox.xcodeproj` to your Xcode project (not `Fox.xcworkspace`). Then link Fox-iOS or Fox-OSX to your test target.

And you're all set up! Dive right in by following the *tutorial*.

3.2.2 CocoaPods

Add to your Podfile for you test target to have the latest stable version of Fox:

```
pod 'Fox', '~>1.0.1'
```

And then `pod install`.

And you're all set up! Dive right in by following the *tutorial*.

3.2.3 Carthage

TODO

3.3 Tutorial

If you haven't installed Fox yet, read up on *Installation*.

This tutorial will use the Objective-C API of Fox. There is a similar Swift API but that's currently alpha and subject to change.

3.3.1 Starting with an Example

Throughout this tutorial, we'll cover the basics of writing property tests. To better understand property tests, let's start with an example-based one first:

```
- (void)testSort {
    NSArray *sortedNumbers = [MySorter sortNumbers:@[@5, @2, @1]];
    XCTAssertEqualObjects(sortedNumbers, @[@1, @2, @5]);
}
```

This is a simple example test about sorting numbers. Let's break down parts of this test and see how Fox rebuilds it up:

```
- (void)testSort {
    // inputs
    NSArray *input = @[@5, @2, @1];
    // behavior to test
    NSArray *sortedNumbers = [MySorter sortNumbers:input];
    // assertion
    XCTAssertEqualObjects(sortedNumbers, @[@1, @2, @5]);
}
```

Fox takes these parts and separates them.

- Inputs are produced using *generator*. Generators describe the type of data to generate.
- Behavior to test remains the same.
- The assertion is based on logical statements of the subject and/or based on the generated inputs. The assertions as usually describe properties of the subject under test.

Let's see how we can convert them to Fox property tests.

3.3.2 Converting to a Property

To convert the sort test into the given property, we describe the intrinsic property of the code under test.

For sorting, the resulting output should have the smallest elements in the start of the array and every element afterwards should be greater than or equal to the element before it:

```
- (void)testSortBySmallestNumber {
    id<FOXGenerator> arraysOfIntegers = FOXArray(FOXInteger());
    FOXAssert(FOXForAll(arraysOfIntegers, ^BOOL(NSArray *integers) {
        // subject under test
        NSArray *sortedNumbers = [MySorter sortNumbers:integers];
        // assertion
        NSNumber *previousNumber = nil;
        for (NSNumber *n in sortedNumbers) {
            if (!previousNumber || [previousNumber integerValue] <= [n integerValue]) {
                previousNumber = n;
            } else {
                return NO; // fail
            }
        }
        return YES; // succeed
    }));
}
```

Let's break that down:

- `FOXInteger` is a *generator* that describes how to produce random integers (NSNumbers).
- `FOXArray` is a *generator* that describes how to generate arbitrary arrays. It takes another generator as an argument. In this case, we're giving it an integer generator to produce randomly sized array of random integers.
- `FOXForAll` defines a property that should always hold true. It takes two arguments, the generator to produce and a block on how to assert against the given generated input.
- `FOXAssert` is how Fox asserts against properties. It will raise an exception if a property does not hold. They're part of Fox's *runner* infrastructure which actually generates all test cases.

The test can be read as:

> Assert that **for all array of integers** named `integer`, sorting `integers` should produce `sortedNumbers`. `sortedNumbers` is an array where the first number is the smallest and subsequent elements are greater than or equal to the element that preceeds it.

3.3.3 Diagnosing Failures

The interesting feature of Fox occurs only when properties fail. Let's write code that will fail the property we just wrote:

```
+ (NSArray *)sortNumbers:(NSArray *)numbers {
    NSMutableArray *sortedNumbers = [[numbers sortedArrayUsingSelector:@selector(compare:)] mutableCo
    if (sortedNumbers.count >= 5) {
        id tmp = sortedNumbers[0];
        sortedNumbers[0] = sortedNumbers[1];
        sortedNumbers[1] = tmp;
    }
    return sortedNumbers;
}
```

Some nefarious little code we added there! We run again we get to see Fox work:

```
Property failed with: ( 0, 0, 0, 0, "-1" )
Location:   // /Users/jeff/workspace/FoxExample/FoxExampleTests/FoxExampleTests.m:41
 FOXForAll(arraysOfIntegers, ^BOOL(NSArray *integers) {
  NSArray *sortedNumbers = [self sortNumbers:integers];
  NSNumber *previousNumber = ((void *)0);
  for (NSNumber *n in sortedNumbers) {
  if (!previousNumber || [previousNumber integerValue] <= [n integerValue]) {
  previousNumber = n;
  }
  else {
   return __objc_no;
   }
   }
   return __objc_yes;
   }
  );

RESULT: FAILED
 seed: 1417500369
 maximum size: 200
 number of tests before failing: 8
 size that failed: 7
 shrink depth: 8
 shrink nodes walked: 52
 value that failed: (
    "-3",
    "-3",
    1,
    "-2",
    "-7",
    "-5"
)
 smallest failing value: (
    0,
    0,
    0,
    0,
    "-1"
)
```

The first line describes the smallest failing example that failed. It's placed there for convenience:

```
Property failed with: ( 0, 0, 0, 0, "-1" )
```

The rest of the first half of the failure describes the location and property that failed.

The latter half of the failure describes specifics on how the smallest failing example was reached:

- `seed` is the random seed that was used to generate the series of tests to run. Along with the maximum size, this can be used to reproduce failures Fox generated.
- `maximum size` is the maximum size hint that Fox used when generating tests. This is useful for reproducing test failures when paired with the seed.
- `number of tests before failing` describes how many tests were generated before the failing test was generated. Mostly for technical curiosity.
- `size that failed` describes the size that was used to generate the original failing test case. The size dicates the general size of the data generated (eg - larger numbers and bigger arrays).
- `shrink depth` indicates how many "changes" performed to shrink the original failing test to produce the smallest one. Mostly for technical curiosity.
- `shrink nodes walked` indicates how many variations Fox produced to find the smallest failing test. Mostly for technical curiosity.
- `value that failed` the original generated value that failed the property. This is before Fox performed any shrinking.
- `smallest failing value` the smallest generated value that still fails the property. This is identical to the value on the first line of this failure description.

So what happened? Fox generates random data until a failure occurs. Once a failure occurs, Fox starts the shrinking process. The shrinking behavior is generator-dependent, but generally alters the data towards the "zero" value:

- For integers, that means moving towards 0 value.
- For arrays, each element shrinks as well as the number of elements moves towards zero.

Each time the value shrinks, Fox will verify it against the property to ensure the test still fails. This is a brute-force process of elimination is an effective way to drop irrevelant noise that random data generation typically produces.

Notice that the last element has significance since it failed to shrink all the way to zero like the other elements. Also note that just because a value has been shrunk to zero doesn't exclude it's potential significance, but it's usually less likely to be significant. In this case, the second to last element happens to be significant.

Warning: Due to the `maximum size` configuration. Fox limits the range of random integers generated. Fox's default maximum size is 200. Observe when you change the failure case to require more than 200 elements for the sort example. See *Configuring Test Generation* for more information.

3.3.4 Testing Stateful APIs

Now this is all well and good for testing purely functional APIs - where the same input produces the same output. What's more interesting is testing stateful APIs.

Before we start, let's talk about the conceptual model Fox uses to verify stateful APIs. We can model **API calls as data** using Fox's *generator* system.

As a simple case, let's test a Queue. We can add and remove objects to it. Removing objects returns the first item in the Queue:

- `[queue add:1]`
- `[queue remove] // => returns 1`
- `[queue add:2]`
- `[queue add:3]`
- `[queue remove] // => returns 2`
- `[queue remove] // => returns 3`

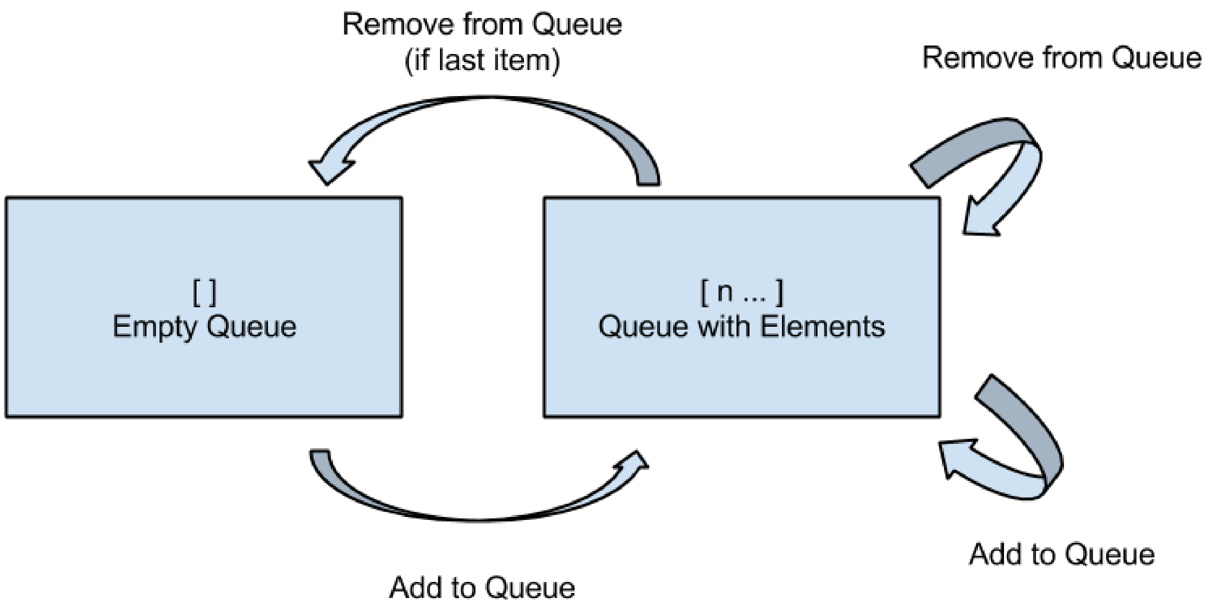
Just generating a series of API calls isn't enough. Fox needs more information about the API:

- **Which API calls are valid to make at any given point?** This is specified in Fox as *preconditions*.
- **What assertions should be after any API call?** This is specified in Fox as *postconditions*.

This is done by describing a state machine. In basic terms, a state machine is two parts: state and transitions.

State indicates data that persists between transitions. Transitions describe how that state changes over time. Transitions can have prerequisites before allowing them to be used.

For this example, we can model the API as a state machine: with transitions for each unique API call, and its state representing what we think the queue should manage. In this case, we'll naively choose an array of the queue's contents as the state.



From there, Fox can **generate a sequence of state transitions** that conform to the state machine. This allows Fox to generate valid sequences of API calls.

We can translate the diagram into code by configuring a `FOXFiniteStateMachine`:

```
- (void)testQueueBehavior {
    // define the state machine with its initial state.
    FOXFiniteStateMachine *stateMachine = [[FOXFiniteStateMachine alloc] initWithInitialModelState:@

    // define the state transition for -[Queue addObject:]
    // we'll only be using randomly generated integers as arguments.
    // note that nextModelState should not mutate the original model state.
    [stateMachine addTransition:[FOXTransition byCallingSelector:@selector(addObject:)
                                                    withGenerator:FOXInteger()
                                                    nextModelState:^id(id modelState, id generatedValue)
                                                        return [modelState arrayByAddingObject:generatedV
                                                    }]];

    // define the state machine for -[Queue removeObject]
    FOXTransition *removeTransition = [FOXTransition byCallingSelector:@selector(removeObject)
                                                          nextModelState:^id(id modelState, id generate
                                                              return [modelState subarrayWithRange:NSMa
                                                          }];
    removeTransition.precondition = ^BOOL(id modelState) {
        return [modelState count] > 0;
    };
    removeTransition.postcondition = ^BOOL(id modelState, id previousModelState, id subject, id gener
        // modelState is the state machine's state after following the transition
        // previousModelState is the state machine's state before following the transition
        // subject is the subject under test. You should not provoke any mutation changes here.
        // generatedValue is the value that the removeTransition generated. We're not using this valu
        // returnedObject is the return value of calling [subject removeObject].
        return [[previousModelState firstObject] isEqual:returnedObject];
    };
    [stateMachine addTransition:removeTransition];

    // generate and execute an arbitrary sequence of API calls
    id<FOXGenerator> executedCommands = FOXExecuteCommands(stateMachine, ^id{
        return [[Queue alloc] init];
    });
    // verify that all the executed commands properly conformed to the state machine.
    FOXAssert(FOXForAll(executedCommands, ^BOOL(NSArray *commands) {
        return FOXExecutedSuccessfully(commands);
    }));
}
```

Note: If you prefer to not have inlined transition definitions, you can always choose to conform to `FOXStateTransition` protocol instead of using `FOXTransition`.

We can now run this to verify the behavior of the queue. This takes more time that the previous example. But Fox's execution time can be tweak if you want faster feedback versus a more thorough test run. See *Configuring Test Generation*.

Just to be on the same page, here's a naive implementation of the queue that passes the property we just wrote:

```
@interface Queue : NSObject
- (void)addObject:(id)object;
- (id)removeObject;
@end

@interface Queue ()
@property (nonatomic) NSMutableArray *items;
@end

@implementation Queue

- (instancetype)init {
    self = [super init];
    if (self) {
        self.items = [NSMutableArray array];
    }
    return self;
}

- (void)addObject:(id)object {
    [self.items addObject:object];
}

- (id)removeObject {
    id object = self.items[0];
    [self.items removeObjectAtIndex:0];
    return object;
}

@end
```

Note: Testing a queue with this technique has obvious testing problems (being the test is like the implementation). But for larger integration tests, this can be useful. They just happen to be less to be concise examples.

To break this, let's modify the queue implementation:

```
- (void)addObject:(id)object {
    if (![object isEqual:@4]) {
        [self.items addObject:object];
    }
}
```

Running the tests again, Fox shows us a similar failure like sort:

```
Property failed with: @[ [subject addObject:4] -> (null), [subject removeObject] -> (null) (Postcondi
Location:   // /Users/jeff/workspace/FoxExample/FoxExampleTests/FoxExampleTests.m:68
FOXForAll(executedCommands, ^BOOL(NSArray *commands) {
return FOXExecutedSuccessfully(commands);
}
);

RESULT: FAILED
seed: 1417510193
maximum size: 200
number of tests before failing: 13
size that failed: 12
shrink depth: 10
shrink nodes walked: 16
value that failed: @[
[subject addObject:-1] -> (null),
[subject addObject:-8] -> (null),
[subject addObject:4] -> (null),
[subject addObject:12] -> (null),
[subject removeObject] -> -1,
[subject addObject:4] -> (null),
[subject addObject:10] -> (null),
[subject removeObject] -> -8,
[subject removeObject] -> 12 (Postcondition FAILED)
    Model before: (
    4,
    12,
    4,
    10
)
    Model after: (
    12,
    4,
    10
),
]
smallest failing value: @[
[subject addObject:4] -> (null),
[subject removeObject] -> (null) (Postcondition FAILED)
    Exception Raised: *** -[__NSArrayM objectAtIndex:]: index 0 beyond bounds for empty array
    Model before: (
    4
)
    Model after: (
),
]
```

Here we can see the original generated test that provoked the failure:

- `[subject addObject:-1]`
- `[subject addObject:-8]`
- `[subject addObject:4]`

- `[subject addObject:12]`
- `[subject removeObject] -> -1`
- `[subject addObject:4]`
- `[subject addObject:10]`
- `[subject removeObject] -> -8`
- `[subject removeObject] -> 12 (Postcondition FAILED)`

That's not as nice to debug as the test after shrinking:

- `[subject addObject:4]`
- `[subject removeObject] -> (null) (Postcondition FAILED)` - out of bounds exception raised.

You may be wondering why the `removeObject` call is still required. This is the only way assertions are made against the queue. Just calling `addObject:` doesn't reveal any issues with an implementation.

And that's most of the power of Fox. You're ready to start writing property tests! Remember:

> "Program testing can at best show the presence of errors, but never their absence."
>
> —Dijkstra

If you want read on, continue to the core of Fox's design: *Generators*.

3.4 Generators

Generators specify directed, random data creation. This means generators know how to create the given data type and how to shrink it. For Objective-C compatibility, generators are only allowed to produce Objective-C objects (`id`).

All generators conform to the `FOXGenerator` protocol and are expected to return a lazy rose tree for consumption by the *Fox runner*.

The power of generators are their composability. Shrinking is provided for *free* if you compose with Fox's built-in generators. In fact, most of Fox's built-in generators are composed on top of `FOXChoose`. Of course you can provide custom shrinking strategies as needed.

For the typed programming enthusiast, generators are functions expected to conform to this type:

> `(id<FOXRandom>, uint32_t) -> FOXRoseTree<U>` where `U` is an Objective-C object.

There are few special cases to this rule. For example, `FOXAssert` expects `FOXRoseTree<FOXPropertyResult>` which the `FORForAll` generator satisfies.

Note: For Haskell programmers, Fox is a decendant to Haskell's QuickCheck 2. Generators are a monadic type which generation is done via the protocol and shrinking is specified by the returned lazy rose tree.

For the list of all generators that Fox provides, read about *Built-in Generators*.

3.4.1 Building Custom Generators

It's easy to compose the built-in generators to produce custom generators for any data type. Let's say we want to generate random permutations of a Person class:

```
// value object. Implementation assumed
@interface Person : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@end
```

We can represent this Person data using by generating an array of values or dictionary of values. Here's how it looks using a dictionary in an property assertion:

```
id<FOXGenerator> dictionaryGenerator = FOXDictionary(@{
    @"firstName": FOXAlphabeticalString(),
    @"lastName": FOXAlphabeticalString()
});
FOXAssert(FOXForAll(dictionaryGenerator, ^BOOL(NSDictionary *data) {
    Person *person = [[Person alloc] init];
    person.firstName = data[@"firstName"];
    person.lastName = data[@"lastName"];
    // assert with person
}));
```

But we want this to be reusable. Using `FOXMap`, we can create a new generator based on the `dictionaryGenerator`:

```
// A new generator that creates random person
id<FOXGenerator> AnyPerson(void) {
    id<FOXGenerator> dictionaryGenerator = FOXDictionary(@{
        @"firstName": FOXAlphabeticalString(),
        @"lastName": FOXAlphabeticalString()
    });
    return FOXMap(dictionaryGenerator, ^id(NSDictionary *data) {
        Person *p = [[Person alloc] init];
        p.firstName = data[@"firstName"];
        p.lastName = data[@"lastName"];
        return p;
    });
}
```

And we can use it like any other generator:

```
FOXAssert(FOXForAll(AnyPerson(), ^BOOL(Person *person) {
    // assert with person
}));
```

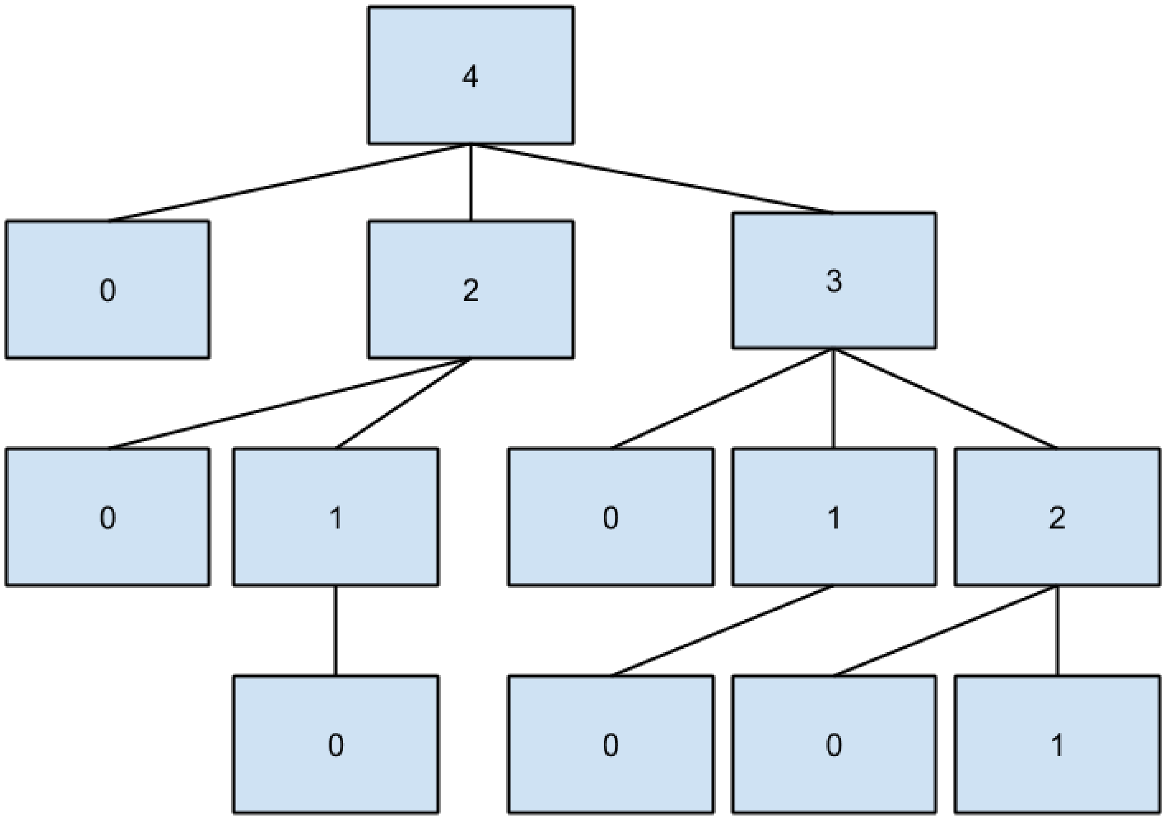
You can see the *reference* for all the generators. Most common generators can be built from the provided mappers.

3.4.2 How Shrinking Works

Generators are just functions that accept a random number generator and a size hint as arguments and then return a rose tree of values.

Rose trees sound fancy, but they're generic trees with an arbitrary number of branches. Each node in the tree represents a value. Fox generators create rose trees instead of individual values. This allows the *runner* to shrink the value by traversing through the children of the tree.

The main shrinking implementation Fox uses are for integers (via `FOXChoose`). For example, if a 4 was generated, the rose tree that `FOXChoose` generates would look like this:



The children of each node represents a smaller value that its parent. Fox will walk depth-first through this tree when a test fails to shrink to the smallest value.

Based on the diagram, the algorithm for shrinking integers prefers:

- Reducing to zero immediately
- Reducing to 50% of the original value
- Reducing the value by 1

This makes it more expensive to find larger integers (because of the redundant checking of zero), but it is generally more common to immediately shrink to the smallest value.

3.4.3 Writing Generators with Custom Shrinking

Warning: **This is significantly more complicated than composing generators**, which is what you want the majority of the time. Composing existing generators will also provide shrinking for free.

Warning: This section assumes knowledge functional programming concepts. It's worth reading up on function composition, map/reduce, recursion, and lazy computation.

It is worth reading up on *How Shrinking Works* if you haven't already.

Let's write a custom integer generator that shrinks to `10` instead of zero. We won't be using anything built on top of `FOXChoose` for demonstrative purposes, but we will be using Fox's *Debugging Functions*.

For the sake of brevity, we'll ignore the problem of maximum storage of integers, but when writing your generators this matters.

Step one, we can easily always generate 10 by returning a child-less rose tree:

```
id<FOXGenerator> MyInteger(void) {
    FOXGenerate(^FOXRoseTree *(id<FOXRandom> random, NSUInteger size) {
        return [[FOXRoseTree alloc] initWithValue:@10];
    });
}
```

`FOXGenerate` is an easy way to create a generator without having to create an object that conforms to `FOXGenerator`. The block is the method body of the one method that the protocol requires.

This is, in fact, what `FOXReturn` does. However, we don't get any randomness:

```
// FOXSample generates 10 random values using the given generator.
FOXSample(MyInteger()); // => @[@3];
```

So let's use the random number generator provided. We'll also use the size to dictate the size we want:

```
id<FOXGenerator> MyInteger(void) {
    FOXGenerate(^FOXRoseTree *(id<FOXRandom> random, NSUInteger size) {
        NSInteger lower = -((NSInteger)size);
        NSInteger upper = (NSInteger)size;
        NSInteger randomInteger = [random randomIntegerWithinMinimum:lower
                                                          andMaximum:upper];
        return [[FOXRoseTree alloc] initWithValue:@(randomInteger)];
    });
}
```

We now generate random integers! But we still don't have any shrinking:

```
// Random integers
FOXSample(MyInteger());
// => @[@-30, @103, @188, @-184, @-22, @-118, @147, @-186, @-128, @-68]

// FOXSampleShrinking takes the first 10 values of the rose tree.
// The first value is the generated value. Subsequent values are
// shrinking values from the first one.
FOXSampleShrinking(MyInteger()) // => @[@-8]; there's no shrinking
```

Let's add a simple shrinking mechanism, we can populate the children of the rose tree we return:

```
id<FOXGenerator> MyInteger(void) {
    FOXGenerate(^FOXRoseTree *(id<FOXRandom> random, NSUInteger size) {
        // remember, we don't care about min / max integer boundaries
        // for this example.
        NSInteger lower = -((NSInteger)size);
        NSInteger upper = (NSInteger)size;
        NSInteger randomInteger = [random randomIntegerWithinMinimum:lower
                                                          andMaximum:upper];
        id<FOXSequence> children = [FOXSequence sequenceFromArray:@[[[FOXRoseTree alloc] initWithValu
        return [[FOXRoseTree alloc] initWithValue:@(randomInteger)
                                         children:children];
    });
}
// Shrinking once
FOXSampleShrinking(MyInteger()) // => @[@-8, @10];
```

Of course, we don't properly handle shrinking for all variations. `FOXSequence` is a port of Clojure's sequence abstraction. They provide opt-in laziness for Fox's rose tree.

We'll mimic the behavior of Fox's integer shrinking algorithm:

- Shrink to 10.
- Shrink towards 10 by 50% of its current value.
- Shrink towards 10 by 1.

We'll do this by defining functions to recursively create our rose tree:

```
// sequenceOfHalfIntegers(@14) -> SEQ(@14, @12, @11)
static id<FOXSequence> sequenceOfHalfIntegers(NSNumber *n) {
    if ([n isEqual:@10]) {
        return nil;
    }
    NSNumber *halfN = @(([n integerValue] - 10) / 2 + 10);
    return [FOXSequence sequenceWithObject:n
                         remainingSequence:sequenceOfHalfIntegers(halfN)];
}
```

`sequenceOfHalfIntegers` creates a sequence of integers that are half increments from n to 10 starting with n. `nil` is equivalent to an empty sequence. Next we define the children values:

```
// eg - sequenceOfSmallerIntegers(@14) -> SEQ(@10, @12, @13)
static id<FOXSequence> sequenceOfSmallerIntegers(NSNumber *n) {
    if ([n isEqual:@10]) {
        return nil;
    }
    return [sequenceOfHalfIntegers(n) sequenceByMapping:^id(NSNumber *m) {
        return @([n integerValue] - ([m integerValue] - 10));
    }];
}
```

`sequenceOfSmallerIntegers` creates a lazy sequence of values between n and 10 (including 10). Each element is `(n - each half number difference to 10)`. Finally, we need to convert this sequence into a rose tree:

```
static FOXRoseTree *roseTreeWithInteger(NSNumber *n) {
    id<FOXSequence> smallerIntegers = sequenceOfSmallerIntegers(n);
    id<FOXSequence> children = [smallerIntegers sequenceByMapping:^id(NSNumber *smallerInteger) {
        return roseTreeWithInteger(smallerInteger);
    }];
    return [[FOXRoseTree alloc] initWithValue:n children:children];
}
```

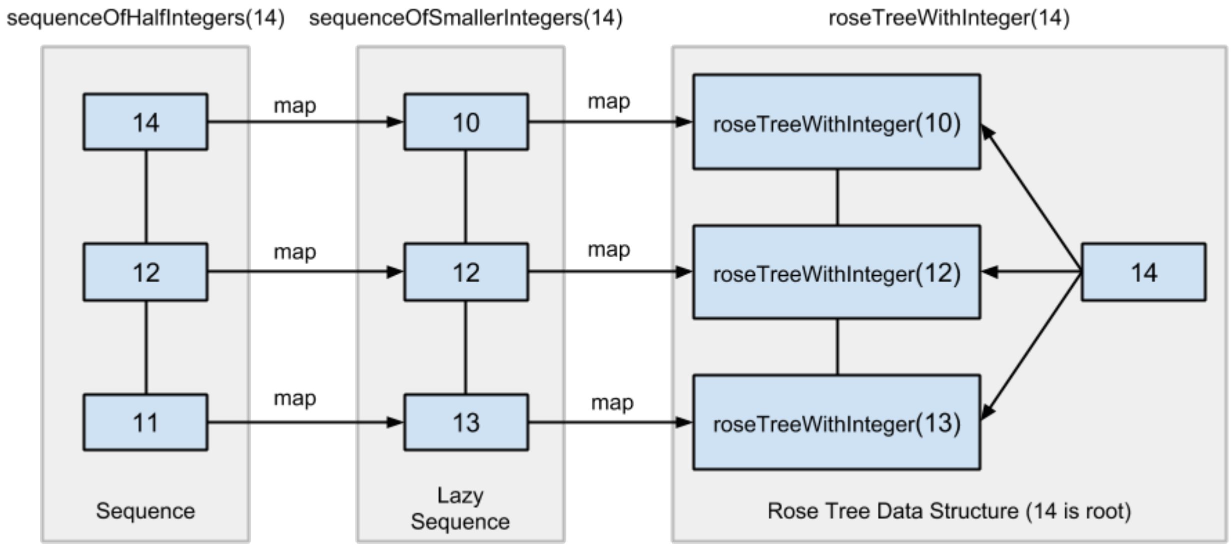
`sequenceOfSmallerIntegers` creates a rose tree for a given number. The children are values from `sequenceOfSmallerIntegers(n)`. The rose tree is recursively generated until `sequenceOfSmallerIntegers` returns an empty sequence (when the number is 10).

Finally, we wire everything together in our function that defines our generator:

```
id<FOXGenerator> MyInteger(void) {
    FOXGenerate(^FOXRoseTree *(id<FOXRandom> random, NSUInteger size) {
        // remember, we don't care about min / max integer boundaries
        // for this example.
        NSInteger lower = -((NSInteger)size);
        NSInteger upper = (NSInteger)size;
        NSInteger randomInteger = [random randomIntegerWithinMinimum:lower
```

```
                                                                         andMaximum:upper];
        return roseTreeWithInteger(@(randomInteger));
    });
}
```

Conceptually, our data pipeline looks like this:



Now we can generate values that shrink to 10! Obviously, this can be applied to more interesting shrinking strategies.

3.5 Runner Overview

The runner is how Fox executes properties, seeds data generation, and triggers shrinking. Although you can create and use the runner directly via `FOXRunner`, it is more common to use wrappers – such as `FOXAssert`.

`FOXAssert` takes a property assertion generator (only `FOXForAll` for now). An exception is raised when the property fails to hold.

A more flexible `FOXAssertWithOptions` can be used to provide parameters that is normally accepted by the runner.

3.5.1 Configuring Test Generation

The primary operation of Fox's runner is to create and execute tests. There are three parameters to configure Fox's test generation:

- The **seed** allows you to seed the random number generator Fox uses. This allows you to set the PRNG to help reproduce failures that Fox may have generated during a test run. Setting this to the default (`0`) will make Fox generate a seed based on the current time.
- The **number of tests** indicates the number of tests Fox will generate for this particular property. More tests generated will more thoroughly cover the property at the cost of time. Setting this to the default (`0`) will make Fox run `500` tests.
- The **maximum size** indicates the maximum size factor Fox will use when generating tests. Generators use this size factor as a hint to produce data of the appropriate sizes. For example, `FOXInteger` will generate integers within the range of 0 to `maximumSize` and `FOXArray` will generate arrays whose number of elements are in the range of 0 to `maximumSize`. Setting this to the default (`0`) will make Fox run with a `maximumSize`

of `200`. If you know this property's data generation can tolerate larger sizes, feel free to increase this. Large collection generation can be prohibitively expensive.

Please note that **seed**, **number of tests**, and **maximum size** should all be recorded to reproduce a failure that Fox may have generated.

3.5.2 Per Assertion Configuration

By default, Fox will generate **500 tests per assertion** with a **maximum size of 200** and a random seed. By changing `FOXAssert` to `FOXAssertWithOptions` we can provide optional configuration by using the `FOXOptions`:

```
FOXAssertWithOptions(FOXForAll(...), (FOXOptions){
    seed=5,                 // default: time(NULL)
    numberOfTests=1000,     // default: 500
    maximumSize=100,        // default: 200
});
```

3.5.3 Global Configuration

Values can be overridden using environment variables to globally change the defaults.

- Setting `FOX_SEED` can specify a specific seed to run for all properties.
- Setting `FOX_NUM_TESTS` sets the number of tests to generate for each property.
- Setting `FOX_MAX_SIZE` sets the maximum size factor Fox will use to when generating tests.

3.5.4 Random Number Generators

Fox provides a hook for custom pseudo-random number generation. This is also what generators receive as their first argument.

The runner uses `FOXDeterministicRandom` which uses C++ random by default. This keeps randomization state isolated from any other potential system that uses a global PRNG. But you can implement the `FOXRandom` protocol to support custom random schemes.

Another implementation Fox provides out of the box is `FOXConstantRandom`, which always generates a constant number. This can be useful for testing generators with example-based tests.

3.5.5 Reporters

The runner also provides a way to observe its operation via a reporter. Reporters are a delegate to the runner. They are invoked synchronously, so be careful about its performance impact on execution.

Delegates cannot influence the execution of the runner, but can provide useful user-facing output about the progress of Fox's execution.

By default, Fox runners do not have a reporter assigned to it. But Fox does provide a couple reporters:

- `FOXStandardReporter` reports in a rspec-like dot reporter.
- `FOXDebugReporter` reports more information about the execution.

The default reporter can be changed by setting it from the instance `FOXAssert` uses: `[[FOXRunner assertInstance] setReporter:reporter]`.

3.6 Built-in Generators

Here are the list of built-in generators that Fox provides. They are either generators of a particular data type or provide computation on top of other generators.

3.6.1 Data Generators

There are many data generators provided for generating data. Most of these generators shrink to zero:

- Numerically zero (or as close as possible)
- Empty collection (or at least shrunk items)

`id<FOXGenerator> FOXInteger(void) // NSNumber`

Generates random integers boxed as a NSNumber. Shrinks to 0:

```
FOXInteger()
// example generations: 0, -1, 1
```

`id<FOXGenerator> FOXPositiveInteger(void) // NSNumber`

Generates random positive integers boxed as a NSNumber. Shrinks to 0:

```
FOXPositiveInteger()
// example generations: 0, 1, 2
```

`id<FOXGenerator> FOXNegativeInteger(void) // NSNumber`

Generates random negative integers boxed as a NSNumber. Shrinks to 0:

```
FOXNegativeInteger()
// example generations: 0, -1, -2
```

`id<FOXGenerator> FOXStrictPositiveInteger(void) // NSNumber`

Generates random positive integers boxed as a NSNumber. Shrinks to 1:

```
FOXStrictPositiveInteger()
// example generations: 1, 2, 3
```

`id<FOXGenerator> FOXStrictNegativeInteger(void) // NSNumber`

Generates random negative integers boxed as a NSNumber. Shrinks to -1:

```
FOXStrictNegativeInteger()
// example generations: -1, -2, -3
```

`id<FOXGenerator> FOXChoose(NSNumber *miniumNumber, NSNumber *maximumNumber) // NSNumber`

Generates random integers boxed as a NSNumber within the given range (inclusive). Shrinks to miniumNumber. The miniumNumber can never be greater than maximumNumber:

```
FOXChoose(@5, @10)
// example generations: 5, 6, 7
```

`id<FOXGenerator> FOXFloat(void) // NSNumber`

Generates random floating point numbers that conform to the IEEE 754 standard in a boxed NSNumber. Shrinks towards zero by shrinking the float’s exponent and mantissa:

```
FOXFloat()
// example generations: 0, 3.436027e+10, -9.860766e-32
```

The generator **does not** generate negative zeros or negative infinities. It is possible to generate positive infinity and NaNs, but is highly unlikely.

id<FOXGenerator> FOXDouble(void) // NSNumber

Generates random doubles that conform to the IEEE 754 standard in a boxed NSNumber. Shrinks towards zero by shrinking the double's exponent and mantissa:

```
FOXDouble()
// example generations: 0, 6.983507489299851e-251, -3.101300322905138e-266
```

The generator **does not** generate negative zeros or negative infinities. It is possible to generate positive infinity and NaNs, but is highly unlikely.

id<FOXGenerator> FOXDecimalNumber(void) // NSDecimalNumber

Generates random decimal numbers. Shrinks towards zero by shrinking the mantissa and exponent.

The generator **does not** generate NaNs:

```
FOXDecimalNumber()
// example generations: 0, -1920000000000000000000000000000000000000000, 790000000000000000000
```

id<FOXGenerator> FOXReturn(id value) // id

Generates only the value provided. Does not shrink:

```
FOXReturn(@2)
// example generations: 2
```

id<FOXGenerator> FOXTuple(NSArray *generators) // NSArray

Generates a fixed-size arrays where each element corresponds to each of the generators provided:

```
FOXTuple(@[FOXInteger(), FOXDecimalNumber()]);
// example generations: @[@0, @0], @[@2, @-129]
```

Shrinking is the smallest value for each of the generators provided. The array does not change size.

id<FOXGenerator> FOXTupleOfGenerators(id<FOXSequence> *generators) // NSArray

Identical to `FOXTuple`, but accepts a FOXSequence of generators instead of an array:

```
id<FOXSequence> generators = [FOXSequence sequenceFromArray:@[FOXInteger(), FOXDecimalNumber()]]
FOXTupleOfGenerators(@[FOXInteger(), FOXDecimalNumber()]);
// example generations: @[@0, @0], @[@2, @-129]
```

id<FOXGenerator> FOXArray(id<FOXGenerator> itemGenerator) // NSArray

Generates a variable-sized array where each element is created via the itemGenerator. Shrinking reduces the size of the array as well as each element generated:

```
FOXArrayOfSize(FOXInteger(), 3)
// example generations: @[@0, @0, @0], @[@2, @-129, @21]
```

id<FOXGenerator> FOXArrayOfSize(id<FOXGenerator> itemGenerator, NSUInteger size) // NSArray

Generates a fixed-size array where each element is created via the itemGenerator. Shrinking only reduces the size of each element generated:

```
id<FOXSequence> generators = [FOXSequence sequenceFromArray:@[FOXInteger(), FOXDecimalNumber()]]
FOXArrayOfSize(FOXInteger(), 3)
// example generations: @[@0, @0, @0], @[@2, @-129, @21]
```

id<FOXGenerator> FOXArrayOfSizeRange(id<FOXGenerator> itemGenerator, NSUInteger minSize, N

Generates a variable-sized array where each element is created via the itemGenerator. The size of the array is within the specified range (inclusive). Shrinking reduces the size of the array to minSize as well as each element generated:

```
id<FOXSequence> generators = [FOXSequence sequenceFromArray:@[FOXInteger(), FOXDecimalNumber()]]
FOXArrayOfSizeRange(FOXInteger(), 1, 2)
// example generations: @[@0], @[@2, @-129]
```

id<FOXGenerator> FOXDictionary(NSDictionary *template) // NSDictionary

Generates random dictionaries of generated values. Keys are known values ahead of time. Specified in *@{<key>: <generator>}* form:

```
FOXDictionary(@{@"name": FOXString(),
                @"age": FOXInteger()});
// example generations: @{@"name": @"", @"age": @0}
```

Only values shrink. The number of pairs the dictionary holds does not shrink.

id<FOXGenerator> FOXSet(id<FOXGenerator> generator) // NSSet

Generates random sets of generated values. The size of the set is not deterministic. Values generated should support the methods required to be placed in an NSSet. Shrinking is per element, which implicitly shrinks the set:

```
FOXSet(FOXInteger())
// example generations: [NSSet setWithObject:@1], [NSSet setWithObjects:@3, @2, nil]
```

id<FOXGenerator> FOXCharacter(void) // NSString

Generates random 1-length sized character string. It may be an unprintable character. Shrinks to smaller ascii numeric values:

```
FOXCharacter()
// example generations: @"\0", @"f", @"k"
```

id<FOXGenerator> FOXAlphabeticalCharacter(void) // NSString

Generates random 1-length sized alphabetical string. Includes both upper and lower case. Shrinks to smaller ascii numeric values:

```
FOXAlphabeticalCharacter()
// example generations: @"A", @"a", @"k"
```

id<FOXGenerator> FOXNumericCharacter(void) // NSString

Generates random 1-length sized numeric string (0-9). Shrinks to smaller ascii numeric values:

```
FOXNumericCharacter()
// example generations: @"0", @"1", @"9"
```

id<FOXGenerator> FOXAlphanumericCharacter(void) // NSString

Generates random 1-length sized numeric string (A-Z,a-z,0-9). Shrinks to smaller ascii numeric values:

```
FOXAlphanumericCharacter()
// example generations: @"A", @"d", @"7"
```

id<FOXGenerator> FOXAsciiCharacter(void) // NSString

Generates random 1-length sized character string. It is ensured to be printable. Shrinks to smaller ascii numeric values:

```
FOXAsciiCharacter()
// example generations: @"A", @"d", @"7", @"%"
```

id<FOXGenerator> FOXString(void) // NSString

Generates random variable length strings. It may be an unprintable string. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXString()
// example generations: @"", @"fo$#@52\n\0", @"sfa453"
```

id<FOXGenerator> FOXStringOfLength(NSUInteger length) // NSString

Generates random fixed-length strings. It may be an unprintable string. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXStringOfLength(5)
// example generations: @"fdg j", @"f#%2\0", @"23zzf"
```

id<FOXGenerator> FOXStringOfLengthRange(NSUInteger minLength, NSUInteger maxLength) // NSSt

Generates random variable length strings within the given range (inclusive). It may be an unprintable string. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXStringOfLengthRange(3, 5)
// example generations: @"fgsj", @"b 2", @"65a\n\0"
```

id<FOXGenerator> FOXAsciiString(void) // NSString

Generates random variable length ascii-only strings. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXAsciiString()
// example generations: @"fgsj", @"b 2", @"65a"
```

id<FOXGenerator> FOXAsciiStringOfLength(NSUInteger length) // NSString

Generates random fixed-length ascii-only strings. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXAsciiStringOfLength(5)
// example generations: @"fgsj1", @"b 122", @"65abb"
```

id<FOXGenerator> FOXAsciiStringOfLengthRange(NSUInteger minLength, NSUInteger maxLength) //

Generates random variable length ascii-only strings within the given range (inclusive). Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXAsciiStringOfLengthRange(2, 5)
// example generations: @"fg", @" 122", @"abb"
```

id<FOXGenerator> FOXAlphabeticalString(void) // NSString

Generates random variable length alphabetical strings. Includes upper and lower cased strings. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXAlphabeticalString()
// example generations: @"fg", @"admm", @"oiuteoer"
```

id<FOXGenerator> FOXAlphabeticalStringOfLength(NSUInteger length) // NSString

Generates random fixed-length alphabetical strings. Includes upper and lower cased letters. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXAlphabeticalStringOfLength(4)
// example generations: @"fguu", @"admm", @"ueer"
```

id<FOXGenerator> FOXAlphabeticalStringOfLengthRange(NSUInteger minLength, NSUInteger maxLe

Generates random variable length alphabetical strings within the given range (inclusive). Includes upper and lower cased strings. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXAlphabeticalStringOfLengthRange(2, 4)
// example generations: @"fguu", @"adm", @"ee"
```

id<FOXGenerator> FOXAlphanumericalString(void) // NSString

Generates random variable length alphanumeric strings. Includes upper and lower cased strings. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXAlphanumericalString()
// example generations: @"fg9u", @"a3M", @"fkljlkbd3241ee"
```

id<FOXGenerator> FOXAlphanumericalStringOfLength(NSUInteger length) // NSString

Generates random fixed-length alphanumeric strings. Includes upper and lower cased letters. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXAlphanumericalStringOfLength(3)
// example generations: @"fg9", @"a3M", @"1ee"
```

id<FOXGenerator> FOXAlphanumericalStringOfLengthRange(NSUInteger minLength, NSUInteger maxL

Generates random variable length alphanumeric strings within the given range (inclusive). Includes upper and lower cased strings. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXAlphanumericalStringOfLengthRange(2, 3)
// example generations: @"fg9", @"aM", @"1e"
```

id<FOXGenerator> FOXNumericalString(void) // NSString

Generates random variable length numeric strings (0-9). Includes upper and lower cased strings. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXNumericalString()
// example generations: @"", @"62", @"0913024"
```

id<FOXGenerator> FOXNumericalStringOfLength(NSUInteger length) // NSString

Generates random fixed-length numeric strings (0-9). Includes upper and lower cased letters. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXNumericalStringOfLength(3)
// example generations: @"521", @"620", @"091"
```

id<FOXGenerator> FOXNumericalStringOfLengthRange(NSUInteger minLength, NSUInteger maxLengt

Generates random variable length numeric strings (0-9) within the given range (inclusive). Includes upper and lower cased strings. Shrinks to smaller ascii numeric values and smaller length strings:

```
FOXNumericalStringOfLengthRange(2, 5)
// example generations: @"21", @"620", @"05991"
```

id<FOXGenerator> **FOXElements**(NSArray **values*)

Generates one of the specified values at random. Does not shrink:

```
FOXElements(@[@1, @5, @9]);
// example generations: @1, @5, @9
```

id<FOXGenerator> FOXSimpleType(void) // id

Generates random simple types. A simple type is a data type that is not made of other types. The value generated may not be safe to print to console. Shrinks according to the data type generated.

Currently, the generators this uses are:

- FOXInteger()
- FOXDouble()
- FOXString()
- FOXBoolean()

But this generator may change to cover more data types at any time.

`id<FOXGenerator> FOXPrintableSimpleType(void) // id`

Generates random simple types. A simple type is a data type that is not made of other types. The value generated is ensured to be printable to console. Shrinks according to the data type generated.

Currently, the generators this uses are:

- FOXInteger()
- FOXDouble()
- FOXAsciiString()
- FOXBoolean()

But this generator may change to cover more data types at any time.

`id<FOXGenerator> FOXCompositeType(id<FOXGenerator> itemGenerator) // id`

Generates random composite types. A composite type contains other data types. Elements of the composite type are from the provided itemGenerator.. Shrinks according to the data type generated.

Currently, the generators this uses are:

- FOXArray()
- FOXSet()

But this generator may change to cover more data types at any time.

`id<FOXGenerator> FOXAnyObject(void) // id`

Generates random simple or composite types. Shrinking is dependent on the type generated.

Currently the generators this uses are:

- FOXSimpleType()
- FOXCompositeType()

But this generator may change to cover more data types at any time.

`id<FOXGenerator> FOXAnyPrintableObject(void) // id`

Generates random printable simple or composite types. Shrinking is dependent on the type generated.

Currently the generators this uses are:

- FOXPrintableSimpleType()
- FOXCompositeType()

But this generator may change to cover more data types at any time.

3.6.2 Computation Generators

Also, you can compose some computation work on top of data generators. The resulting generator adopts the same shrinking properties as the original generator.

id<FOXGenerator> **FOXMap**(id<FOXGenerator> *generator*, id(^fn)(id *generatedValue*))

Applies a block to each generated value. Shrinking is dependent on the original generator:

```
// create a generator that produces strictly positive integers.
FOXMap(FOXInteger(), ^id(NSNumber *value) {
    return @(ABS([value integerValue]) ?: 1);
});
```

id<FOXGenerator> **FOXBind** (id<FOXGenerator> *generator*, id<FOXGenerator> (^fn)(id *generatedValue*))

Applies a block to the value that the original generator generates. The block is expected to return a new generator. Shrinking is dependent on the returned generator. This is a way to create a new generator from the input of another generator's value:

```
// create a generator that produces arrays of random capacities
// does not shrink because of FOXReturn's generator behavior.
FOXBind(FOXPositiveInteger(), ^id<FOXGenerator>(NSNumber *value) {
    return FOXReturn([NSArray arrayWithCapacity:[value integerValue]]);
});
```

id<FOXGenerator> **FOXResize** (id<FOXGenerator> *generator*, NSUInteger *newSize*)

Overrides the given generator's size parameter with the specified size. Prevents shrinking:

```
// Similar to FOXArrayOfSizeRange(FOXInteger(), @0, @10)
FOXResize(FOXArray(FOXInteger()), 10);
```

id<FOXGenerator> **FOXOptional** (id<FOXGenerator> *generator*)

Creates a new generator that has a 25% chance of returning *nil* instead of the provided generated value:

```
// A 25% chance of returning nil instead of NSNumber
FOXOptional(FOXInteger())
// example generations: @1, @5, nil, @22
```

id<FOXGenerator> **FOXFrequency** (NSArray **tuples*)

Dispatches to one of many generators by probability. Takes an array of tuples (2-sized array) - `@[@[@probability_uint, generator]]`. Shrinking follows whatever generator is returned.

// equivalent to FOXOptional(FOXInteger()) FOXFrequency(@[@[@1, FOXReturn(nil)],

@[@3, FOXInteger()]]);

// example generations: @1, @5, nil, @22

id<FOXGenerator> **FOXSized** (id<FOXGenerator> (^fn)(NSUInteger *size*))

Encloses the given block to create generator that is dependent on the size hint generators receive when generating values:

```
// returns a generator that creates arrays with specific capacities.
// the capacities grow as the size hint grows. A large size hint can
// still generate smaller size values.
//
// No shrinking because we're using FOXReturn.
FOXSized(^id<FOXGenerator>(NSUInteger size) {
    return FOXReturn([NSArray arrayWithCapacity:size]);
});
```

id<FOXGenerator> **FOXSuchThat** (id<FOXGenerator> *generator*, BOOL(^predicate)(id *generatedValue*))

Returns each generated value if-and-only-if it satisfies the given block. If the filter excludes more than 10 values in a row, the resulting generator assumes it has reached maximum shrinking:

```
// inefficiently generates only even numbers.
FOXSuchThat(FOXInteger(), ^BOOL(NSNumber *value) {
    return [value integerValue] % 2 == 0;
});
```

Warning: Using `FOXSuchThat` and `FOXSuchThatWithMaxTries` are "filter" generators and can lead to significant waste in test generation by Fox. While it gives you the most flexibility the kind of generated data, it is the most computationally expensive. Use other generators when possible.

id<FOXGenerator> **FOXSuchThatWithMaxTries** (id<FOXGenerator> *generator*, BOOL(^predicate)(id *generatedValue)*, NSInteger *maxTries*)

Returns each generated value iff it satisfies the given block. If the filter excludes more than the given max tries in a row, the resulting generator assumes it has reached maximum shrinking:

```
// inefficiently generates numbers divisible by 10.
FOXSuchThat(FOXInteger(), ^BOOL(NSNumber *value) {
    return [value integerValue] % 10 == 0;
});
```

Warning: Using `FOXSuchThat` and `FOXSuchThatWithMaxTries` are "filter" generators and can lead to significant waste in test generation by Fox. While it gives you the most flexibility the kind of generated data, it is the most computationally expensive. Use other generators when possible.

id<FOXGenerator> **FOXOneOf** (NSArray **generators*)

Returns generated values by randomly picking from an array of generators. Shrinking is dependent on the generator chosen:

```
// evenly distributed between integers and strings
FOXOneOf(@[FOXInteger(), FOXString()]);
// example generations: @1, @"bgj%)#x", @9
```

id<FOXGenerator> **FOXForAll** (id<FOXGenerator> *generator*, BOOL (^then)(id *generatedValue)*)

Asserts using the block and a generator and produces test assertion results (FOXPropertyResult). FOXPropertyResult is a data structure storing the results of the assertion. Shrinking tests against smaller values of the given generator:

```
FOXForAll(FOXInteger(), ^BOOL(NSNumber *generatedValue) {
    // will fail eventually
    return [generatedValue integerValue] > 0;
});
// example generations: <FOXPropertyResult: pass>, <FOXPropertyResult: fail>
```

id<FOXGenerator> **FOXForSome** (id<FOXGenerator> *generator*, FOXPropertyStatus (^then)(id *generatedValue)*)

Like FOXForAll, but allows the assertion block to "skip" potentially invalid test cases:

```
FOXForAll(FOXInteger(), ^BOOL(NSNumber *generatedValue) {
    // skip tests if 0 was generated
    if ([generatedValue integerValue] == 0) {
        return FOXPropertyStatusSkipped;
    }
    // will fail eventually
    return [generatedValue integerValue] > 0;
});
// example generations: <FOXPropertyResult: pass>, <FOXPropertyResult: fail>, <FOXPropertyResult
```

id<FOXGenerator> **FOXCommands** (id<FFOXStateMachine> *stateMachine*)

Generates arrays of FOXCommands that satisfies a given state machine.

id<FOXGenerator> **FOXExecuteCommands** (id<FOXStateMachine> *stateMachine*)

Generates arrays of FOXExecutedCommands that satisfies a given state machine and executed against a subject. Can be passed to FOXExecutedSuccessfully to verify if the subject conforms to the state machine.

3.6.3 Debugging Functions

Fox comes with a handful of functions that can help you diagnose generator problems.

id<FOXGenerator> **`FOXSample`** (id<FOXGenerator> *generator*)
Samples 10 values that generator produces.

id<FOXGenerator> **`FOXSampleWithCount`** (id<FOXGenerator> *generator*, NSUInteger *numberOfSamples*)
Samples a number of values that a generator produces.

id<FOXGenerator> **`FOXSampleShrinking`** (id<FOXGenerator> *generator*)
Samples 10 steps of shrinking from a value that a generator produces.

id<FOXGenerator> **`FOXSampleShrinkingWithCount`** (id<FOXGenerator> *generator*, NSUInteger *numberOfSamples*)
Samples a number of steps of shrinking from a value that a generator produces.

Index