⚠ This is a pre-release version. Many things are missing. ⚠

Thanks for buying the pre-release book. We're very happy that you chose to support us. We're planning to publish the first content early July. We'll be updating this PDF on a regular basis, but for the latest available content, check out https://github.com/objcio/fpinswift-beta

We would love to hear from you. If there are any topics you'd like to see included, or if there's anything that deserves a better explanation, let us know. File an issue on GitHub, or send an email to mail@objc.io

Florian, Wouter and Chris

objc ↑↓

Functional Programming in Swift

By Chris Eidhof, Florian Kugler and Wouter Swierstra

Introduction

Why write this book? There is plenty of documentation on Swift readily available from Apple and many more books on the way. Why does the world need yet another book on yet another programming language?

This books tries to teach you to think *functionally*. We believe that Swift has just the right language features to teach you how to write *functional programs*. But what makes a program functional? Or why bother learning about this in the first place? In his paper on Why Functional Programming Matters, John Hughes writes:

> Functional programming is so called because its fundamental operation is the application of functions to arguments. A main program itself is written as a function that receives the program's input as its argument and delivers the program's output as its result.

So rather than thinking as a program as a sequence of assignments and method calls, functional programmers emphasise that each program can be repeatedly broken into smaller and smaller pieces; these pieces are reassembled by passing them as arguments to functions. By avoiding assignment statements and side-effects, Hughes argues that functional programs are more modular and easier to maintain than their imperative or object oriented counterparts. And this is a Very Good Thing.

In our experience, learning to think functionally is not an easy thing. It challenges the way they've been trained to decompose problems. Programmers used to writing for-loops find recursion confusing; the lack of assignment statements and global state is crippling; at first sight, closures, generics, higher-order functions, and monads are just plain weird.

In this book, we want to demystify functional programming and dispell some of the prejudices people may have against it. You don't need to have a PhD in mathematics to use these ideas to improve your code! We won't cover the complete Swift language specification or teach you to set up your first project in XCode. But we will try to teach you the basic principles of pure functional programming that will make you a better developer in any language.

Thinking Functionally

What this chapter is about

Functions in Swift are *first-class values*, i.e., functions may be passed as arguments to other functions. This idea may seem strange if you're used to working with simple types, such as integers, booleans or structs. In this chapter, we will try to motivate why first-class functions are useful and give a first example of functional programming in action.

Example: Battleship

We'll introduce first-class functions using the example of an algorithm you would implement if you would want to create a Battleship game.[1] The problem we'll look at boils down to determining whether or not a given point is in range, without being too close to friendly ships or ourselves.

As a first approximation, you might write a very simple function that checks whether or not a point is in range:

```
typealias Position = CGPoint

func inRange1(target: Position, range: Double) -> Bool {
   return sqrt(target.x * target.x + target.y * target.y) <= range
}
```

Note that we are using Swift's typealias construct let us introduce a new name for an existing type. From now on, whenever we write `Position`, feel free to read `CGPoint`, a pair of an `x` and `y` coordinate.

Now this works fine, if you assume that the we are always located at the origin. Perhaps it would be better though to pass in additional arguments representing our current location:

```swift
func inRange2(target: Position, ownPosition: Position, range: Double) -> Bool {
   let dx = ownPosition.x - target.x
   let dy = ownPosition.y - target.y
   return sqrt(dx * dx + dy * dy) <= range
}
```

But now you realize that you also want to avoid targeting ships if they are too close to yourself. So you change your code again:

```swift
let minimumDistance = 2.0

func inRange3(target: Position, ownPosition: Position, range: Double) -> Bool {
   let dx = ownPosition.x - target.x
   let dy = ownPosition.y - target.y
   return sqrt(dx * dx + dy * dy) <= range
          && sqrt(dx * dx + dy * dy) >= minimumDistance
}
```

Finally, you also need to avoid targeting ships that are too close to one of your other ships. To handle this, you add further arguments that represent the location of a friendly ship, and update your code accordingly:

```swift
func inRange4(target: Position, ownPosition: Position, friendly: Position, range: Double) -> Bool {
   let dx = ownPosition.x - target.x
   let dy = ownPosition.y - target.y
   let friendlyDx = friendly.x - target.x
   let friendlyDy = friendly.y - target.y
   return sqrt(dx * dx + dy * dy) <= range
          && sqrt(dx * dx + dy * dy) >= minimumDistance
          && !(sqrt(friendlyDx * friendlyDx + friendlyDy * friendlyDy) >= minimumDistance)
}
```

As this code evolves, it becomes harder and harder to maintain. This method expresses a complicated calculation in one big lump of code. Let’s try to refactor this into smaller, compositional pieces.

First-class functions

There are different approaches to refactoring this code. One obvious pattern would be to introduce a function that computes the distance between two points; or functions that check when two points are 'close' or 'far away' (for some definition of close and far). In this chapter, however, we'll take a slightly different approach.

The original problem boiled down to defining a function that determined when a point was in range or not. The type of such a function would be something like:

```
func pointInRange(point: Position) -> Bool {
    // ...
}
```

The type of this function is going to be so important, that we're going to give it a separate name:

```
typealias Region = (Position) -> Bool
```

From now on, the `Region` type will refer to functions from a `Position` to a `Bool`. This isn't strictly necessary, but it can make some of the type signatures that we'll see below a bit easier to digest.

Instead of defining an object or struct to represent regions, we represent a region by a *function* that determines if a given point is in the region or not. If you're not used to functional programming this may seem strange, but remember: functions in Swift are first-class values!

Going forward, we will write functions that create, manipulate and combine such regions.

The first region we define is a `circle`, centered around the origin:

```
func circle(radius: Double) -> Region {
    return { point in sqrt(point.x * point.x + point.y * point.y) <= radius }
}
```

Note that, given a radius `r`, the call `circle(r)` *returns a function*. Here we use Swift's notation for closures to construct the function that we wish to return. Given an argument position, `point`, we check that the `point` is in the region delimited by a circle of the given radius centered around the origin.

Of course, not all circles are centered around the origin. We could add more arguments to the `circle` function to account for this. Instead, though, we will write a *region transformer:*

```
func shift(offset: Position, region: Region) -> Region {
    return { point in
        let shiftedPoint = Position(x: point.x - offset.x, y: point.y - offset.y)
        return region(shiftedPoint)
    }
}
```

The call `shift(offset, region)` moves the region to the right and up by `offet.x` and `offset.y` respectively. How is it implemented? Well we need to return a `Region`, that is a function from points to `Bool`. To do this, we start writing another closure, introducing the point we need to check. From this point, we compute a new point with the coordinates `point.x - offset.x` and `point.y - offset.y`. Finally, we check that this new point is in the *original* region by passing it as an arguments to the `region` function.

Interestingly, there are lots of other ways to transform existing regions. For instance, we may want to define a new region by inverting a region. The resulting region consists of all the points outside the original region:

```
func invert(region: Region) -> Region {
    return { point in !region(point) }
}
```

We can also write functions that combine existing regions into larger, complex regions. For instance, these two functions take the points that are in *both* argument regions or *either* argument region, respectively:

```
func intersection(region1 : Region, region2 : Region) -> Region {
    return { point in region1(point) && region2(point) }
}

func union(region1 : Region, region2 : Region) -> Region {
    return { point in region1(point) || region2(point) }
}
```

Of course, we can use these functions to define even richer regions. The `crop` function takes two regions as argument, `region` and `minusRegion`, and constructs a region with all points that are in the first, but not in the second region.

```
func difference(region: Region, minusRegion: Region) -> Region {
    return intersection(region, invert(minusRegion))
}
```

This example shows how Swift lets you compute and pass around functions no differently than integers or booleans.

Now let's turn our attention back to our original example. With this small library in place, we can now refactor the complicated `inRange` function as follows:

```
func inRange(ownPosition: Position, target: Position, friendly: Position, range: D
ouble) -> Bool {
  let targetRegion = difference(circle(range), circle(minimumDistance))
  let friendlyRegion = shift(friendly, circle(minimumDistance))
  return difference(targetRegion, friendlyRegion)(target)
}
```

The way we've defined the `Region` type does have its disadvantages. In particular, we cannot inspect *how* a region was constructed: is it composed of smaller regions? Or is it simply a circle around the origin? The only thing we can do is to check whether or not a given point is within a region or not. If we would want to visualize a region, we would have to sample enough points to generate a (black and white) bitmap.

Type-driven development

In the introduction, we mentioned how functional programs take the application of functions to arguments as the canonical way to assemble bigger programs. In this chapter, we have seen a concrete example of this functional design methodology. We have defined a series of functions for describing regions. Each of these functions is not very powerful by itself. Yet together, they can describe complex regions that you wouldn't want to write from scratch.

The solution is simple and elegant. It is quite different from what you might write, had you just refactored the `inRange`$_4$ function into separate methods. The crucial design decision we made was *how* to define regions. Once we chose the `Region` type, all the other definitions followed naturally. The moral of the example is **choose your types carefully**. More than anything else, types guide the development process.

QuickCheck

In recent years, testing has become much more prevalent in Objective-C. Many popular libraries are now tested automatically with continuous integration tools. The standard framework for writing unit tests is XCTest. In addition, a lot of third-party frameworks are available (such as Specta, Kiwi and FBSnapshotTestCase), and a number of frameworks are currently being developed in Swift.

All of these frameworks follow a similar pattern: they typically consist of some fragment of code, together with an expected result. The code is then executed; its result is then compared to the expected result mentioned in the test. Different libraries test at different levels: some might test individual methods, some test classes and some perform integration testing (running the entire app). In this chapter, we will build a small library for property-based testing of Swift code.

When writing unit tests, the input data is static and defined by the programmer. For example, when unit-testing an addition method, we might write a test that verifies that `1 + 1` is equal to `2`. If the implementation of addition changes in such a way that this property is broken, the test will fail. More generally, however, we might test that addition is commutative, or in other words, that `a + b` is equal to `b + a`. To test this, we might write a test case that verifies that `42 + 7` is equal to `7 + 42`.

In this chapter, we'll build Swift port (a part of) QuickCheck,[2] a Haskell library for random testing. Instead of writing individual unit tests that each test a function is correct for some particular input, QuickCheck allows you to describe abstract *properties* of your functions and *generate* tests to verify them.

This is best illustrated with an example. Suppose we want to verify that plus is commutative. To do so, we start by writing a function that checks where `x + y` is equal to `y + x` for two integers `x` and `y`:

```
func plusIsCommutative(x : Int, y : Int) -> Bool {
    return x + y == y + x
}
```

Checking this statement with QuickCheck is as simple as calling the `check` function:

```
check("Plus should be commutative", plusIsCommutative)


> "Plus should be commutative" passed 100 tests.
> ()
```

The `check` function works by calling the `plusIsCommutative` function with two random integers, over and over again. If the statement isn't true, it will print out the input that caused the test to fail. The key insight here is that we can describe abstract *properties* of our code (like commutativity) using *functions* that return a `Bool` (like `plusIsCommutative`). The `check` function now uses this property to *generate* unit tests; giving much better code coverage than you could achieve using hand-written unit tests.

Of course, not all tests pass. For example, we can define a statement that describes that the subtraction is commutative:

```
func minusIsCommutative(x : Int, y : Int) -> Bool {
    return x - y == y - x
}
```

Now, if we run QuickCheck on this function, we will get a failing test case:

```
check("Minus should be commutative", minusIsCommutative)


> "Minus should be commutative" doesn't hold: (1, 0)
> ()
```

Using Swift's syntax for trailing closures, we can also write tests directly, without defining the property (such as `plusIsCommutative` or `minusIsCommutative`) separately:

```
check("Additive identity") {(x : Int) in x + 0 == x }


> "Additive identity" passed 100 tests.
> ()
```

Building QuickCheck

In order to build this library, we will need to do a couple of things. First, we need a way to generate random values for different kinds of types. Then, we need to implement the `check` function, which will feed our test with random values a number of times. Should a test fail, then we'd like to make the input smaller. For example, if our test fails on an array with 100 elements, we'll try to make it smaller and see if the test still fails. Finally, we'll need to do some extra work to make sure our check function works on types that have generics.

Generating Random Values

First, let's define a protocol that knows how to generate arbitrary values. As the return type, we use `Self`, which is the type of the class that implements it.

```
protocol Arbitrary {
    class func arbitrary() -> Self
}
```

First, let's write an instance for `Int`. We use the `arc4random` function from the standard library and convert it into an `Int`:

```
extension Int : Arbitrary {
    static func arbitrary() -> Int {
        return Int(arc4random())
    }
}
```

Now we can generate random integers like this:

```
Int.arbitrary()


> 1296035768
```

To generate random strings, we need to do a little bit more work. First, we generate a random length `x` between 0 and 100. Then, we generate `x` random characters, and append them to the string:

```
extension String : Arbitrary {
    static func arbitrary() -> String {
        let randomLength = random(from: 0, to: 100)
        var string = ""
        for _ in 0..randomLength {
            let randomInt : Int = random(from: 13, to: 255)
            string += Character(UnicodeScalar(randomInt))
        }
        return string
    }

}
```

We can call it in the same way as we generate random `Int`s, except, that we call it on the `String` class:

```
String.arbitrary()


> rÇÖoàò7Ü¡ú0TAÅ¯r¾:G«
```

Implementing the `check` function

Now we are ready to implement a first version of our check function.

```
func check1<A: Arbitrary>(message: String, prop: A -> Bool) -> () {
    for _ in 0..numberOfIterations {
        let value = A.arbitrary()
        if !prop(value) {
            println("\"\(message)\" doesn't hold: \(value)")
            return
        }
    }
    println("\"\(message)\" passed \(numberOfIterations) tests.")
}
```

The `check1` function consists of a simple loop that generates random input for the argument property in every iteration. If a counterexample is found, it is printed and the function returns; if no counterexample is found, the `check1` function reports the number of successful tests that have passed. (Note that we called the function `check1`, because we'll write the final version a bit later).

Here's how we can use this function to test properties:

```
check1("Additive identity") {(x : Int) in x + 0 == x }


> "Additive identity" passed 100 tests.
> ()
```

Making values smaller

If we run our `check1` function on strings, we might get quite a long failure message:

```
check1("Every string starts with Hello") {(s: String) in s.hasPrefix("Hello")}


> "Every string starts with Hello" doesn't hold: R>nþ!!3GÛ\IxhÌÅØä]:ÈæAsyz±lûàÆ®Hy
ÿ;MÅ¨üÔĉV7Ò<ÐIâsß«ý
> ()
```

Ideally, we'd like our failing input to be a short as possible. In general, the smaller the counterexample, the easier it is to spot which piece of code is causing the failure. In principle, the user could try to trim the input that triggered the failure and try rerunning the test; rather than place the burden on the user, however, we will automate this process.

To do so, we will make an extra protocol called `Smaller`, which does only one thing: it tries to shrink the counterexample.

```
protocol Smaller {
    func smaller() -> Self?
}
```

Note that the return type of the `small` function is marked as optional. There are cases when it is not clear how to shrink test data any further. For example, there is no obvious way to shrink an empty array. We will return `nil` in that case.

In our instance for integers we just try to divide the integer by two until we reach zero:

```
extension Int : Smaller {
    func smaller() -> Int? {
        return self == 0 ? nil : self / 2
    }
}
```

We can now test our instance:

```
100.smaller()


> 50
```

For strings, we just take the substring from the first index (unless the string is empty).

```
extension String : Smaller {
    func smaller() -> String? {
        return self.isEmpty ? nil : self.substringFromIndex(1)
    }
}
```

To use the `Smaller` protocol in the `check` function, will need the ability to shrink any test data generated by our `check` function. To do so, we will redefine our `Arbitrary` protocol to extend the `Smaller` protocol:

```
protocol Arbitrary : Smaller {
    class func arbitrary() -> Self
}
```

Iterating while we found the value

We can now redefine our `check` function to shrink any test data that triggers a failure. To do this, we use the `iterateWhile` function that takes a condition, an initial value and repeatedly applies a function as long as the condition holds.

```
func check2<A: Arbitrary>(message: String, prop: A -> Bool) -> () {
    for _ in 0..numberOfIterations {
        let value = A.arbitrary()
        if !prop(value) {
            let smallerValue = iterateWhile({ value in !prop(value) }, initialValu
e: value) {
              $0.smaller()
            }
            println("\"\(message)\" doesn't hold: \(smallerValue)")
            return
        }
    }
    println("\"\(message)\" passed \(numberOfIterations) tests.")
}
```

The `iterateWhile` function is defined as follows:

```
func iterateWhile<A>(condition: A -> Bool, #initialValue: A, next: A -> A?) -> A {
    var value = initialValue
    while let x = next(value) {
        if condition(x) {
           value = x
        } else {
            return value
        }
    }
    return value
}
```

Adding support for tuples and arrays

⚠ The rest of this chapter needs to be revised a bit. ⚠

Let’s suppose we write a version of QuickSort in Swift:

```
func qsort(var array: Int[]) -> Int[] {
    if array.count == 0 { return [] }
    let pivot = array.removeAtIndex(0)
    let lesser = array.filter { $0 < pivot }
    let greater = array.filter { $0 >= pivot }
    return qsort(lesser) + [pivot] + qsort(greater)
}
```

And we can write a property to check our version of QuickSort against the built-in sort function:

```
check2("qsort should behave like sort", { (x: Int[]) in return qsort(x) == sort(x)
})
```

However, the compiler warns us that `Int[]` doesn't conform to the `Arbitrary` protocol. In order to implement `Arbitrary`, we first have to implement `Smaller`:

```
extension Array : Smaller {
    func smaller() -> Array<T>? {
        if self.count == 0 { return nil }
        var copy = self
        copy.removeAtIndex(0)
        return copy
    }
}
```

Now, if we want to test this function, we also need an `Arbitrary` instance that produces arrays. However, to define an instance for `Array`, we also need to make sure that the element type of the array is also an instance of `Arbitrary`. For example, in order to generate an array of random numbers, we first need to make sure that we can generate random numbers.

Unfortunately, it is currently not possible to express this restriction at the type level, making it impossible to make `Array` conform to the `Arbitrary` protocol. However, what we *can* do is overload the `check` function. First, now that the compiler knows we can generate random values of `X`, we can write a function that generates a random array filled with random `X` values:

```
func check<X : Arbitrary>(message: String, prop : Array<X> -> Bool) -> () {
    let arbitraryArray : () -> Array<X> = {
        let randomLength = Int(arc4random() % 50)
        return Array(0..randomLength).map { _ in return X.arbitrary() }
    }
    let smaller : Array<X> -> Array<X>? = {
      return $0.smaller()
    }
    ...
}
```

Now, instead of duplicating the logic from `check`$_2$, we can extract a helper function. This takes an extra parameter of type `ArbitraryI<A>`, which is a struct with two functions: one for generating arbitrary values of `A`, and one for making values of `A` smaller:

```
func checkHelper<A>(arbitraryInstance: ArbitraryI<A>, prop: A -> Bool, message: St
ring) -> () {
    for _ in 0..numberOfIterations {
        let value = arbitraryInstance.arbitrary()
        if !prop(value) {
            let smallerValue = iterateWhile({ !prop($0) }, initialValue: value, ar
bitraryInstance.smaller)
            println("\"\(message)\" doesn't hold: \(smallerValue)")
            return
        }
    }
    println("\"\(message)\" passed \(numberOfIterations) tests.")
}
```

The struct is very simple, it just wraps up the two functions. Alternatively, the functions could have been passed as parameters, but because they always belong together, it’s simpler to pass them around as one value.

```
struct ArbitraryI<T> {
    let arbitrary : () -> T
    let smaller: T -> T?
}
```

Now, we can finish our `check` function for arrays:

```
func check<X : Arbitrary>(message: String, prop : Array<X> -> Bool) -> () {
    let arbitraryArray : () -> Array<X> = {
        let randomLength = Int(arc4random() % 50)
        return Array(0..randomLength).map { _ in return X.arbitrary() }
     }
    let instance = ArbitraryI(arbitrary: arbitraryArray, smaller: { $0.smaller() }
)
    checkHelper(instance, prop, message)
}
```

And we can write an overloaded variant of `check` that works on every type that conforms to `Arbitrary`:

```
func check<X : Arbitrary>(message: String, prop : X -> Bool) -> () {
    let instance = ArbitraryI(arbitrary: { X.arbitrary() }, smaller: { $0.smaller(
) })
    checkHelper(instance, prop, message)
}
```

Now, we can finally run `check` to verify our QuickSort implementation:

```
check("qsort should behave like sort", { (x: Int[]) in return qsort(x) == sort(x)
})


> "qsort should behave like sort" passed 100 tests.
> ()
```

1. The code presented here is inspired by the Haskell solution to a problem posed by the ARPA documented here: Hudak, Paul, and Mark P. Jones. *Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity*. Technical report, Yale University, Dept. of CS, New Haven, CT, 1994. ↩
2. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.1361 “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs” ↩