# objc ↑↓ Functional Programming in Swift

Chris Eidhof, Florian Kugler, and Wouter Swierstra

# Contents

## Chapter 3

# Wrapping Core Image

The previous chapter introduced the concept of *higher-order function* and showed how functions can be passed as arguments to other functions. However, the example used there may seem far removed from the 'real' code that you write on a daily basis. In this chapter, we will show how to use higher-order functions to write a small, functional wrapper around an existing, object-oriented API.

Core Image is a powerful image processing framework, but its API can be a bit clunky to use at times. The Core Image API is loosely typed — image filters are configured using key-value coding. It is all too easy to make mistakes in the type or name of arguments, which can result in runtime errors. The new API we develop will be safe and modular, exploiting *types* to guarantee the absence of such runtime errors.

Don't worry if you're unfamiliar with Core Image or cannot understand all the details of the code fragments in this chapter. The goal isn't to build a complete wrapper around Core Image, but instead to illustrate how concepts from functional programming, such as higher-order functions, can be applied in production code. If you are unfamiliar with Objective-C and programming with dictionaries, you may want to skip this chapter on your first read-through and return to it later.

## The Filter Type

One of the key classes in Core Image is the `CIFilter` class, which is used to create image filters. When you instantiate a `CIFilter` object, you (almost) always provide an input image via the `kCIInputImageKey` key, and then retrieve the filtered result via the `kCIOutputImageKey` key. Then you can use this result as input for the next filter.

In the API we will develop in this chapter, we'll try to encapsulate the exact details of these key-value pairs and present a safe, strongly typed API to our users. We define our own `Filter` type as a function that takes an image as its parameter and returns a new image:

```
typealias Filter = CIImage -> CIImage
```

This is the base type that we are going to build upon.

## Building Filters

Now that we have the `Filter` type defined, we can start defining functions that build specific filters. These are convenience functions that take the parameters needed for a specific filter and construct a value of type `Filter`. These functions will all have the following general shape:

```
func myFilter(/* parameters */) -> Filter
```

Note that the return value, `Filter`, is a function as well. Later on, this will help us compose multiple filters to achieve the image effects we want.

To make our lives a bit easier, we'll extend the `CIFilter` class with a convenience initializer and a computed property to retrieve the output image:

```
typealias Parameters = Dictionary<String, AnyObject>

extension CIFilter {
    convenience init(name: String, parameters: Parameters) {
        self.init(name: name)
```

```
        setDefaults()
        for (key, value: AnyObject) in parameters {
            setValue(value, forKey: key)
        }
    }

    var outputImage: CIImage {
        return self.valueForKey(kCIOutputImageKey) as CIImage
    }
}
```

The convenience initializer takes the name of the filter and a dictionary as parameters. The key-value pairs in the dictionary will be set as parameters on the new filter object. Our convenience initializer follows the Swift pattern of calling the designated initializer first.

The computed property, `outputImage`, provides an easy way to retrieve the output image from the filter object. It looks up the value for the `kCIOutputImageKey` key and casts the result to a value of type `CIImage`. By providing this computed property of type `CIImage`, users of our API no longer need to cast the result of such a lookup operation themselves.

### Blur

With these pieces in place, we can define our first simple filters. The Gaussian blur filter only has the blur radius as its parameter:

```
func blur(radius: Double) -> Filter {
    return { image in
        let parameters: Parameters = [
            kCIInputRadiusKey: radius,
            kCIInputImageKey: image
        ]
        let filter = CIFilter(name: "CIGaussianBlur",
                              parameters:parameters)
        return filter.outputImage
```

```
    }
}
```

That's all there is to it. The `blur` function returns a function that takes an argument `image` of type `CIImage` and returns a new image (`return filter.outputImage`). Because of this, the return value of the `blur` function conforms to the `Filter` type we have defined previously as `CIImage -> CIImage`.

This example is just a thin wrapper around a filter that already exists in Core Image. We can use the same pattern over and over again to create our own filter functions.

### Color Overlay

Let's define a filter that overlays an image with a solid color of our choice. Core Image doesn't have such a filter by default, but we can, of course, compose it from existing filters.

The two building blocks we're going to use for this are the color generator filter (`CIConstantColorGenerator`) and the source-over compositing filter (`CISourceOverCompositing`). Let's first define a filter to generate a constant color plane:

```
func colorGenerator(color: NSColor) -> Filter {
    return { _ in
        let parameters: Parameters = [kCIInputColorKey: color]
        let filter = CIFilter(name:"CIConstantColorGenerator",
                              parameters: parameters)
        return filter.outputImage
    }
}
```

This looks very similar to the `blur` filter we've defined above, with one notable difference: the constant color generator filter does not inspect its input image. Therefore, we don't need to name the image parameter in the function being returned. Instead, we use an unnamed parameter, `_`,

to emphasize that the image argument to the filter we are defining is ignored.

Next, we're going to define the composite filter:

```
func compositeSourceOver(overlay: CIImage) -> Filter {
    return { image in
        let parameters: Parameters = [
            kCIInputBackgroundImageKey: image,
            kCIInputImageKey: overlay
        ]
        let filter = CIFilter(name: "CISourceOverCompositing",
                              parameters: parameters)
        let cropRect = image.extent()
        return filter.outputImage.imageByCroppingToRect(cropRect)
    }
}
```

Here we crop the output image to the size of the input image. This is not strictly necessary, and it depends on how we want the filter to behave. However, this choice works well in the examples we will cover.

Finally, we combine these two filters to create our color overlay filter:

```
func colorOverlay(color: NSColor) -> Filter {
    return { image in
        let overlay = colorGenerator(color)(image)
        return compositeSourceOver(overlay)(image)
    }
}
```

Once again, we return a function that takes an image parameter as its argument. The `colorOverlay` starts by calling the `colorGenerator` filter. The `colorGenerator` filter requires a `color` as its argument and returns a filter, hence the code snippet `colorGenerator(color)` has type `Filter`. The `Filter` type, however, is itself a function from `CIImage` to `CIImage`; we can pass an *additional* argument of type `CIImage` to `colorGenerator(color)` to

compute a new overlay `CIImage`. This is exactly what happens in the definition of `overlay` — we create a filter using the `colorGenerator` function and pass the `image` argument to this filter to create a new image. Similarly, the value returned, `compositeSourceOver(overlay)(image)`, consists of a filter, `compositeSourceOver(overlay)`, being constructed and subsequently applied to the `image` argument.

## Composing Filters

Now that we have a blur and a color overlay filter defined, we can put them to use on an actual image in a combined way: first we blur the image, and then we put a red overlay on top. Let's load an image to work on:

```
let url = NSURL(string: "http://tinyurl.com/m74sldb");
let image = CIImage(contentsOfURL: url)
```

Now we can apply both filters to these by chaining them together:

```
let blurRadius = 5.0
let overlayColor = NSColor.redColor().colorWithAlphaComponent(0.2)
let blurredImage = blur(blurRadius)(image)
let overlaidImage = colorOverlay(overlayColor)(blurredImage)
```

Once again, we assemble images by creating a filter, such as `blur(blurRadius)`, and applying the resulting filter to an image.

### Function Composition

Of course, we could simply combine the two filter calls in the above code in a single expression:

```
let result = colorOverlay(overlayColor)(blur(blurRadius)(image))
```

However, this becomes unreadable very quickly with all these parentheses involved. A nicer way to do this is to compose filters by defining a custom operator for filter composition. To do so, we'll start by defining a function that composes filters:

```
func composeFilters(filter1: Filter, filter2: Filter) -> Filter {
    return { img in filter2(filter1(img)) }
}
```

The `composeFilters` function takes two argument filters and defines a new filter. This composite filter expects an argument `img` of type `CIImage`, and passes it through both `filter1` and `filter2`, respectively. We can use function composition to define our own composite filter, like this:

```
let myFilter1 = composeFilters(blur(blurRadius),
                               colorOverlay(overlayColor))
let result1 = myFilter1(image)
```

We can go one step further to make this even more readable, by introducing an operator for filter composition. Granted, defining your own operators all over the place doesn't necessarily contribute to the readability of your code. However, filter composition is a recurring task in an image processing library, so it makes a lot of sense:

```
infix operator >>> { associativity left }

func >>> (filter1: Filter, filter2: Filter) -> Filter {
    return { img in filter2(filter1(img)) }
}
```

Now we can use the `>>>` operator in the same way we used the `composeFilters` before:

```
let myFilter2 = blur(blurRadius) >>> colorOverlay(overlayColor)
let result2 = myFilter2(image)
```

Since we have defined the `>>>` operator as being left-associative we can read the filters that are applied to an image from left to right — like Unix pipes.

The filter composition operation that we have defined is an example of *function composition*. In mathematics, the composition of the two functions `f` and `g`, sometimes written `f ∘ g`, defines a new function mapping

an input to `x` to `f(g(x))`. With the exception of the order, this is precisely what our `>>>` operator does: it passes an argument image through its two constituent filters.

## Theoretical Background: Currying

In this chapter, we've seen that there are two ways to define a function that takes two arguments. The first style is familiar to most programmers:

```
func add1(x: Int, y: Int) -> Int {
    return x + y
}
```

The `add1` function takes two integer arguments and returns their sum. In Swift, however, we can also define another version of the same function:

```
func add2(x: Int) -> (Int -> Int) {
    return { y in return x + y }
}
```

Here, the function `add2` takes one argument, `x`, and returns a *closure*, expecting a second argument, `y`. These two `add` functions must be invoked differently:

```
add1(1, 2)
add2(1)(2)


> 3
```

In the first case, we pass both arguments to `add1` at the same time; in the second case, we first pass the first argument, `1`, which returns a function, which we then apply to the second argument, `2`. Both versions are equivalent: we can define `add1` in terms of `add2`, and vice versa.

In Swift, we can even leave out one of the return statements and some of the parentheses in the type signature of `add2`, and write:

```
func add2(x: Int) -> Int -> Int {
    return { y in x + y }
}
```

The function arrow, `->`, associates to the right. That is to say, you can read the type `A -> B -> C` as `A -> (B -> C)`. Throughout this book, however, we will typically introduce a type alias for functional types (as we did for the `Region` and `Filter` types), or write explicit parentheses.

The `add1` and `add2` examples show how we can always transform a function that expects multiple arguments into a series of functions that each expect one argument. This process is referred to as *currying*, named after the logician Haskell Curry; we say that `add2` is the *curried* version of `add1`.

There is a third way to curry functions in Swift. Instead of constructing the closure explicitly, as we did in the definition of `add2`, we can also define a curried version of `add1` as follows:

```
func add3(x: Int)(y: Int) -> Int {
    return x + y
}
```

Here we have listed the arguments that `add3` expects, one after the other, each surrounded by its own parentheses. To call `add3` we must, however, provide an explicit name for the second argument:

```
add3(1)(y: 2)
```

So why is currying interesting? As we have seen in this book thus far, there are scenarios where you want to pass functions as arguments to other functions. If we have *uncurried* functions, like `add1`, we can only apply a function to *both* its arguments. On the other hand, for a *curried* function, like `add2`, we have a choice: we can apply it to one *or* two arguments. The functions for creating filters that we have defined in this chapter have all been curried — they all expected an additional image argument. By writing our filters in this style, we were able to compose them easily using the

>>> operator. Had we instead worked with *uncurried* versions of the same functions, it still would have been possible to write the same filters and filter composition operator, but the resulting code would have been much clunkier.

## Discussion

This example illustrates, once again, how we break complex code into small pieces, which can all be reassembled using function application. The goal of this chapter was not to define a complete API around Core Image, but instead to sketch out how higher-order functions and function composition can be used in a more practical case study.

Why go through all this effort? It's true that the Core Image API is already mature and provides all the functionality you might need. But in spite of this, we believe there are several advantages to the API designed in this chapter:

- **Safety** — using the API we have sketched, it is almost impossible to create runtime errors arising from undefined keys or failed casts.
- **Modularity** — it is easy to compose filters using the >>> operator. Doing so allows you to tease apart complex filters into smaller, simpler, reusable components. Additionally, composed filters have the exact same type as their building blocks, so you can use them interchangeably.
- **Clarity** — even if you have never used Core Image, you should be able to assemble simple filters using the functions we have defined. To access the results, you don't need to know about special dictionary keys, such as `kCIOutputImageKey`, or worry about initializing certain keys, such as `kCIInputImageKey` or `kCIInputRadiusKey`. From the types alone, you can almost figure out how to use the API, even without further documentation.

Our API presents a series of functions that can be used to define and compose filters. Any filters that you define are safe to use and reuse. Each

filter can be tested and understood in isolation. We believe these are compelling reasons to favor the design sketched here over the original Core Image API.