Functional programming in Swift

Wouter Swierstra

BOB Konf
Functional Programming in Swift

by Chris Eidhof, Florian Kugler, and Wouter Swierstra

The Filter Type

Use the `filter` method on a `NSImage` to create image filters. When you return a `CGImage` object, you should always provide an input image via the `CGImageSource` key, and then return the filtered image via the `CGImageDestination` key. Thereafter, you can use the returned filter on other images.

In this chapter, we’ll be manipulating the `NSImage` objects to make them more visually appealing. We define our own `filter` type as a function that takes an image as an argument and returns a new image.

**Typealias:** `Filter` = `CGImageDestination -> CGImage`

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

Building Filters

Now that we have the `Filter` type defined, we can start defining functions that take specific filters. These are convenience functions that take the parameters needed for a specific filter and create a new instance of `Filter`. These functions can all have the following signature:

```swift
func filterImage(_ input: Image, with parameters ...)
```

Note that the `filterImage` function is a convenience function. Later on, it will help compose multiple filters to achieve the image effects we want.

To make our code a bit more elegant, we might also like to provide classes with a `convenience initializer` and a computed property to retrieve the output image.

**Typealias:** `Parameters = [(Key: String, Value: Any)]`

**extension:** `Filter` {
`convenience initializer` (with parameters) `parameters` (as Parameters) `}`

objc ↑↓

Composing Filters

Now that we have a blur and a color overlay filter defined, we can put them together to achieve our desired effect.

```swift
let blurRadius = 5.0
let overlayColor = UIColor.red.withAlphaComponent(0.5)

let blurImage = blur(radius: blurRadius, image: input)
let overlayImage = colorOverlay(overlayColor: overlayColor, inputImage: blurImage)

let finalImage = overlayImage
```
What kind of language is Swift?
It’s like Objective-C, without the C.

— Craig Federighi, WWDC
"Objective-C without the C" implies something subtractive, but Swift dramatically expands the design space through the introduction of generics and functional programming concepts.

– Chris Lattner, Apple Developer Forums
What is functional programming?
Maps and filters

Many people describe functional programming as being about map, filter and reduce:

\[ [1,2,3].map(x \mapsto x + 1) \]

These are examples of functions drawn from FP.

But this is like saying object oriented programming is about Shapes and Animals.
Characteristics of FP

- Modularity
- Effective usage of types
- Careful treatment of state and effects
What you will learn

The aim of this talk is not to teach you Swift...

... or the frameworks and IDE for iOS and OS X development.

But rather showcase how you can use some of the ideas from functional programming in your projects.
San Francisco
The challenge

The Core Image API is a bit clunky.

CIFilter *hueAdjust = [CIFilter filterWithName:@"CIHueAdjust"];
[hueAdjust setDefaults];
[hueAdjust setValue: myCIImage forKey: kCIInputImageKey];
[hueAdjust setValue: @2.094f forKey: kCIInputAngleKey];

The resulting image after running the filter can be retrieved from the kCIOutputImageKey.
The challenge

The Core Image API has some drawbacks:

• It's easy to forget to set some parameters (or set the wrong parameters) – causing a run-time crash;

• Not type safe – you can set the wrong type of value for some key, which again causes a run-time crash;

• Not modular – there's no easy way to compose two filters.
A functional solution

```plaintext
typealias Filter = CIImage -> CIImage

A filter is a function that transforms an image.
```
A trivial filter

```swift
func noFilter() -> Filter {
    return {image in return image}
}
```

This filter does nothing, and returns the input image.
An example filter

```swift
func blur(radius: Double) -> Filter {
    return {image in
        let parameters : Parameters =
            [kCIInputRadiusKey: radius, kCIInputImageKey: image]
        // Here we're calling a convenience initializer
        // that sets certain defaults
        let blurFilter = CIFilter(name:"CIGaussianBlur",
                                    parameters:parameters)
        return blurFilter.outputImage
    }
}
```
let url = NSURL(string: "http://tinyurl.com/sfswift")
let image : CIImage = CIImage(contentsOfURL: url)

let blurBy5 : Filter = blur(5)
let blurred : CIImage = blurBy5(image)
Other filters

```swift
func compositeSourceOver(overlay: CIImage) -> Filter {
...
}

func colorGenerator(color: NSColor) -> Filter {
...
}

func colorOverlay(color: NSColor) -> Filter {
...
}
```
Filtering more than once

```swift
let img : CIImage = ...
let blurRadius = 5.0
let overlayColor = NSColor.whiteColor().color.WithAlphaComponent(0.2)
let blurredImage = blur(blurRadius)(image)
let overlaidImage = colorOverlay(overlayColor)(blurredImage)
```
How can we compose filters?
Composing filters – function composition

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

let img = ...
let compositeFilter = compose(blur(blurRadius),
                                colorOverlay(overlayColor))

let filteredImg = compositeFilter(img)
```
A composition operator

infix operator >>> { associativity left }

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

let myFilter = blur(blurRadius) >>> colorOverlay(overlayColor)

We can now chain together filters, similarly to Unix pipes.
Taking stock

We have a wrapper around a fragment of Core Image that is both **type-safe** and **modular**.

Using higher-order functions (similar to Objective-C's *blocks*)...

... but there are alternative definitions that have the same compositional behaviour, that are entirely first-order.
Enumerations
Enumerations in Objective C

Enumerations are thin wrapper around a collection of integer constants:

```c
enum NSStringEncoding {
    NSASCIIStringEncoding = 1,
    NSNEXTSTEPStringEncoding = 2,
    NSJapaneseEUCStringEncoding = 3,
    NSUTF8StringEncoding = 4,
    // ... 
}
```
Enumerations in Objective C

Why should such expressions make sense?

```c
if (NSASCIIStringEncoding + NSNEXTSTEPStringEncoding == NSJapaneseEUCStringEncoding) {
...
}
```
Enumerations in Swift

In Swift on the other hand, enumerations:

• introduce a new type, separate from the underlying integers;
• may have associated values;
• can be decomposed by pattern matching.
Reading a file – Obj C

The **type** of the **NSString** initializer isn't helpful.

```objc
+ (instancetype)stringWithContentsOfFile:(NSString *)path
    encoding:(NSStringEncoding)enc
    error:(NSError **)error
```

To check for errors, do I inspect the `error` or the return value?
Reading a file – Swift

```swift
func readFile(path: String, encoding: NSStringEncoding) -> String? {
    var maybeError: NSError? = nil
    return NSString(contentsOfFile: path,
                    encoding: encoding,
                    error: &maybeError)
}
```

The type is telling us more, but we can't get our hands on the NSError when the function fails.
Enumerations

```swift
enum Result {
    case Success(String)
    case Failure(NSError)
}
```

A value of type Result is tagged as being either:

- **Success** – in which case we have the file contents;
- **Failure** – in which case we have an NSError.
readFile revisited

```swift
func readFile(path: String, encoding: NSStringEncoding) -> Result {
    var maybeError: NSError?
    let maybeString: String? = NSString(contentsOfFile: path,
                                         encoding: encoding,
                                         error: &maybeError)

    if let string = maybeString {
        return Result.Success(string)
    } else {
        return Result.Failure(maybeError!)
    }
}
```
Reading a file

```swift
switch readFile("README.md", NSASCIIStringEncoding) {
    case let Result.Success(contents):
        // Process file contents
        ...
    case let Result.Failure(error):
        // Handle error
        ...
}
```
Diagrams in Swift
Diagrams in Objective C

NSColor.blueColor().setFill()
CGContextFillRect(context, CGRectMake(0.0, 37.5, 75.0, 75.0))
NSColor.redColor().setFill()
CGContextFillRect(context, CGRectMake(75.0, 0.0, 150.0, 150.0))
NSColor.greenColor().setFill()
CGContextFillEllipseInRect(context,
                          CGRectMake(225.0, 37.5, 75.0, 75.0))
Diagrams in Objective C

But what if I want to draw this:

Instead of this:
Diagrams in Objective C

The drawing commands are non-compositional:

- They have hard-coded coordinates;
- They focus on *how* things should be drawn rather than *what* should be drawn;
- Any changes to a sub-drawing require rewriting the complete code.
A functional solution...

Instead of drawing commands directly, we'll design a *domain specific language* for diagram descriptions.

1. Define an enumeration that describes diagrams

2. 'Interpret' this description using a Core Graphics
Intended solution

```swift
let blueSquare = square(side: 1).fill(NSColor.blueColor())
let redSquare = square(side: 2).fill(NSColor.redColor())
let greenCircle = circle(radius: 1).fill(NSColor.greenColor())
let example1 = blueSquare ||| redSquare ||| greenCircle
```
Adding new shapes is easy:

```swift
let cyanCircle = circle(radius: 1).fill(NSColor.cyanColor())
let example2 = blueSquare ||| cyanCircle ||| redSquare ||| greenCircle
```

This solution is *compositional*
enum Primitive {
    case Ellipse
    case Rectangle
    case Text(String)
}

denum Diagram {
    case Prim(CGSize, Primitive)
    case Beside(Diagram, Diagram)
    case Below(Diagram, Diagram)
    case Attributed(Attribute, Diagram)
    case Align(Vector2D, Diagram)
}

1 Recursive enumerations need a workaround
Example: computing the size

extension Diagram {
    var size: CGSize {
        switch self {
            case let .Prim(size, _):
                return size
            case let .Attributed(_, x):
                return x.size
            case let .Beside(l, r):
                let sizeL = l.size
                let sizeR = r.size
                return CGSizeMake(sizeL.width + sizeR.width,
                                    max(sizeL.height, sizeR.height))
        }
    }
}
Drawing a diagram

```swift
func draw(context: CGContextRef, bounds: CGRect, diagram: Diagram) {
    switch diagram {
    case let .Prim(size, .Ellipse):
        let frame = fit(defaultAlign, size, bounds)
        CGContextFillEllipseInRect(context, frame)
    // And similar cases for drawing text and squares
    }
}
```
func draw(context: CGContextRef, bounds: CGRect, diagram: Diagram) {
    switch diagram {
    ...
    case let .Beside(left, right):
        let (lFrame, rFrame) =
            splitHorizontal(bounds, left.size/diagram.size)
        draw(context, lFrame, left)
        draw(context, rFrame, right)
    }

A few more cases for vertical composition, alignment, etc.
Building a more complete library

On top of this we can define *combinators* to make it easier to define complex diagrams:

```swift
func square(side: CGFloat) -> Diagram {
    return rect(width: side, height: side)
}

infix operator ||| { associativity left }
func ||| (l: Diagram, r: Diagram) -> Diagram {
    return Diagram.Beside(l,r)
}

infix operator --- { associativity left }
func --- (l: Diagram, r: Diagram) -> Diagram {
    return Diagram.Below(l,r)
}
```
Adding attributes or alignment

```swift
extension Diagram {
    func fill(color: NSColor) -> Diagram {
        return Diagram.Attributed(Attribute.FillColor(color), self)
    }

    func alignTop() -> Diagram {
        return Diagram.Align(Vector2D(x: 0.5, y: 1), self)
    }
}
```

So we can now write:

```swift
let redSquare = square(side: 2).fill(NSColor.redColor())
let greenCircle = circle(radius: 1).fill(NSColor.greenColor())
let example1 = greenCircle.alignTop() ||| redSquare
```
Combining lists of diagrams

```swift
let empty: Diagram = rect(width: 0, height: 0)

func hcat(diagrams: [Diagram]) -> Diagram {
    return diagrams.reduce(empty, |||
}
```
Example: visualizing dictionaries

```javascript
let cities = [
  "Moscow": 10.56,
  "Shanghai": 14.01,
  "Istanbul": 13.3,
  "Berlin": 3.43,
  "New York": 8.33
];
```
Generating bar graphs

```swift
func barGraph(input: [(String, Double)]) -> Diagram {
    let normalizedValues : [CGFloat] = normalize(input)
    let bars = hcat(normalizedValues.map { x in
        rect(width: 1, height: 3 * x)
        .fill(NSColor.blackColor())
        .alignBottom()
    })
    let labels = hcat(input.map { x in
        text(width: 1, height: 0.3, text: x.0)
        .alignTop()
    })
    return bars --- labels
}
```
Diagrams

• A compositional language for defining simple diagrams
• Easy to extend with new combinators
• Separates the *what* from the *how*
• The same techniques can be used in other programming languages, but they feel more natural in Swift.
Things I haven't talked about

- Generics & protocols
- Reference types versus value types
- Sequences & generators
- QuickCheck & testing functional code
- Currying, parser combinators, type-level programming...
The future of Swift

Apple is starting to actively push Swift...

Swift offers developers both a great platform and a great language
Learning more

• Lots of Apple documentation
• Swift tutorial today
• Functional Swift Conference
• obj.io – Issue 16
• Summer School on Applied Functional Programming in Utrecht.
Questions?