Currying in Practice : Image Filter with Swift

Currying

Filter Type

```
typealias Filter = (CIImage) -> CIImage
```

Blur

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

```

Color Overlay

Now let's define a filter that can overlay a fixed color overlay on the image. The two basic components we will use are: the color generation filter (CIConstantColorGenerator) and the image overlay compositing filter (CISourceOverCompositing).

First let's define a filter that generates a fixed color:

Finally, we create a color overlay filter by combining the two filters:

```swift
func colorGenerator(color: NSColor) -> Filter {
    return { _ in
        guard let c = CIColor(color: color) else { fatalError () }
        let parameters = [kCIInputColorKey: c]
        guard let filter = CIFilter (name: "CIConstantColorGenerator",
            withInputParameters: parameters) else { fatalError() }
        guard let outputImage = filter.outputImage else { fatalError() }
        return outputImage
    }
}
```

Next, we will define the compositing filter:

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

}
```

We return a function that takes an image as an argument. colorOverlay first calls the colorGenerator filter. colorGenerator takes a color as an argument and then returns a new filter, so the

colorGenerator(color) is of Filter type. The Filter type itself is a function from CIImage to CIImage. So we can pass an additional CIImage type parameter to the colorGenerator(color) function and end up with a new overlay of CIImage type. That's all that happens in the process of defining the overlay, which can be roughly summarized as follows.

First create a filter using the colorGenerator function, and then pass an image parameter to that filter to create a new image.

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

```

Composing Filters

Now we can try it. Apply the two filters in a chain to the loaded image

```swift
let url = NSURL(string: "http://placeholder.jpg")!
let image = CIImage(contentsOfURL: url)!
let blurRadius = 5.0
let overlayColor = NSColor.redColor().colorWithAlphaComponent(0.2)
let blurredImage = blur(blurRadius)(image)
let overlaidImage = colorOverlay(overlayColor)(blurredImage)

```

We can simply combine the two expressions that call the filters in the above code into one line.

The code loses its readability.

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

We can also define a function to combine the filters:

The composeFilters function takes two Filter type arguments and returns a newly defined filter. This compose filter takes a CIImage type image parameter, passes it to filter1, gets the return value and then passes it to filter2. We can use the compose function to define a compose filter, like this:

```swift

func composeFilters(filter1: Filter , _ filter2 : Filter ) -> Filter {
    return { image in filter2 ( filter1 (image)) }
}

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

We can go one step further and introduce a custom operator for filter composition.

We can make our operator left-associative.

Reference

[1]. https://www.objc.io/books/functional-swift/

Notes

At first, I want to follow the following chapters of this book to build a simple parser which can parse `10+ 4 * 3` and modify the code of the Spreadsheet Application so that it can build and run successfully.

I spent more than 20 hours studying the last few chapters of the book. But later, especially during finals week, incompatible versions, too much syntax I wasn't familiar with and so on caused me to not finally complete the mini-project along the way I had originally intended.

In the end, I had no choice other than to choose a simple example that really helps me. Because at first I had trouble understanding how to use the Currying in a practical application.

I consider it a pretty cool book **especially the last few chapters** that combine the knowledge of the course and my interests very well.

> Case Study: Parser Combinators
>
> Case Study: Building a Spreadsheet Application https://github.com/objcio/functional-swift
>
> Parsing
>
> Functors, Applicative Functors, and Monads