Functional programming in Swift

Wouter Swierstra

BOB Konf

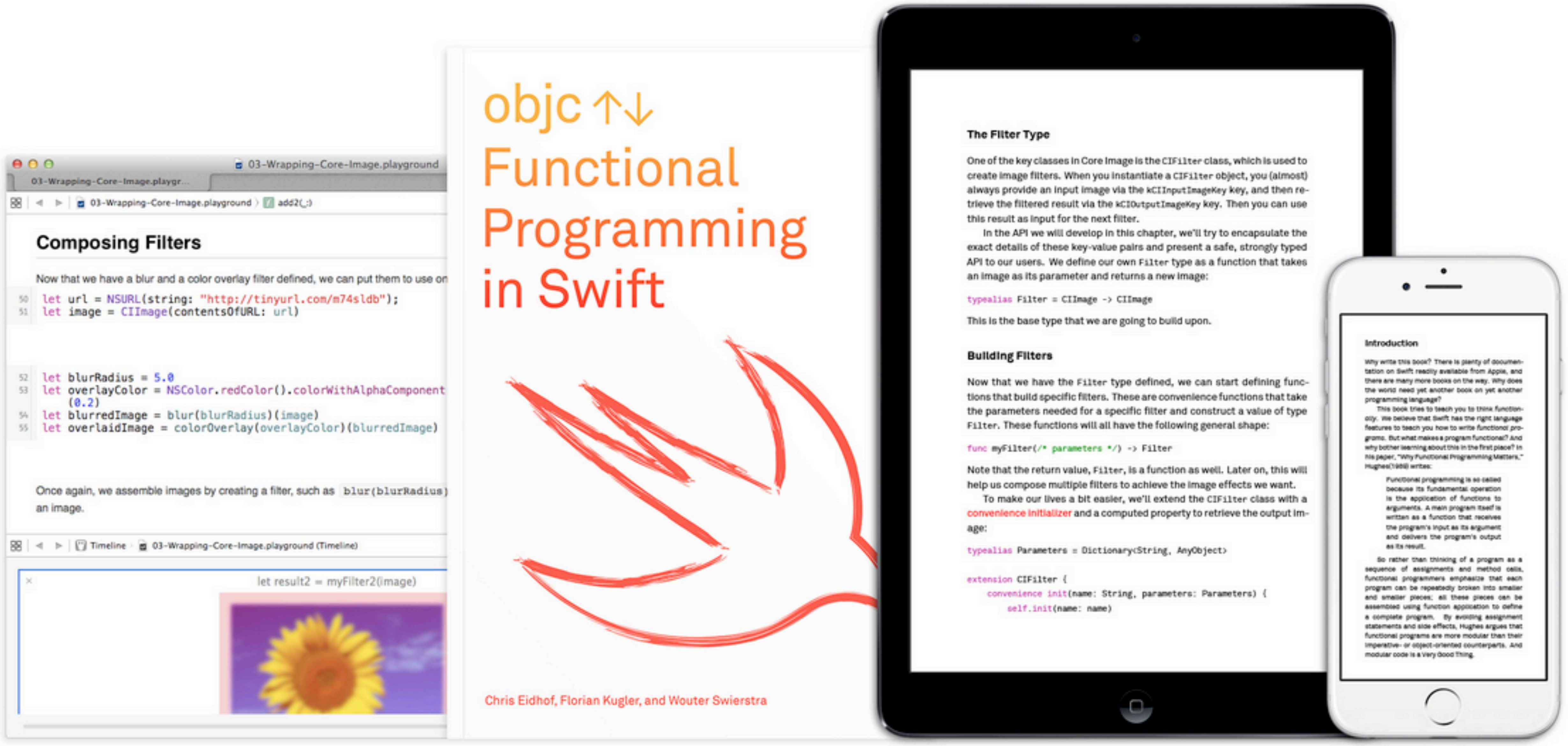
WWDC

June 2014

objc ↑↓

Functional Programming in Swift

by Chris Eidhof, Florian Kugler, and Wouter Swierstra


03-Wrapping-Core-Image.playground
03-Wrapping-Core-Image.playgr...
03-Wrapping-Core-Image.playground 〉 add2(_:)

Composing Filters

Now that we have a blur and a color overlay filter defined, we can put them to use on

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

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

Timeline 03-Wrapping-Core-Image.playground (Timeline)
let result2 = myFilter2(image)
objc ↑↓
Functional Programming in Swift
Chris Eidhof, Florian Kugler, and Wouter Swierstra

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)
```

Introduction

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

This book tries to teach you to think functionally. We believe that Swift has the right language features to teach you how to write functional programs. But what makes a program functional? And why bother learning about this in the first place? In his paper, "Why Functional Programming Matters," Hughes(1989) 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 of a program as a sequence of assignments and method calls, functional programmers emphasize that each program can be repeatedly broken into smaller and smaller pieces; all these pieces can be assembled using function application to define a complete program. By avoiding assignment statements and side effects, Hughes argues that functional programs are more modular than their imperative- or object-oriented counterparts. And modular code is a Very Good Thing.

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 in 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.

Core Image

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

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

A filter is a function that transforms an image.

A trivial filter

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

This filter does nothing, and returns the input image.

An example filter

```
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
    }
}
```

Calling this filter

```
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

```
let img : CIImage = ...
let blurRadius = 5.0
let overlayColor = NSColor.whiteColor().colorWithAlphaComponent(0.2)
let blurredImage = blur(blurRadius)(image)
let overlaidImage = colorOverlay(overlayColor)(blurredImage)
```

How can we compose filters?

Composing filters – function composition

```
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:

```
enum NSStringEncoding {
    NSASCIIStringEncoding = 1,
    NSNEXTSTEPStringEncoding = 2,
    NSJapaneseEUCStringEncoding = 3,
    NSUTF8StringEncoding = 4,
// ...
}
```

Enumerations in Objective C

Why should such expressions make sense?

```
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.

```
+ (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

```
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

```
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

```
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!)
}
```

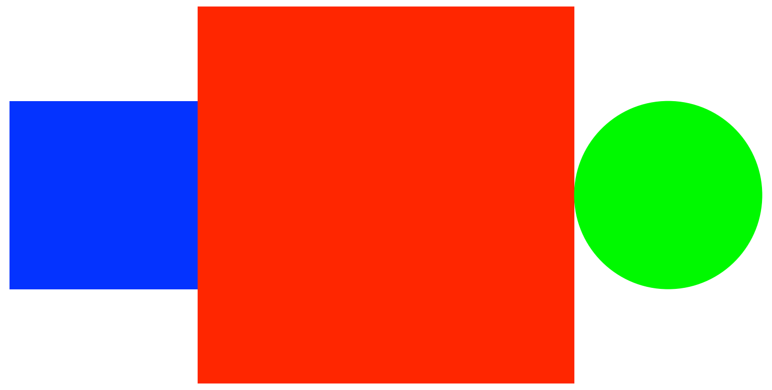
Reading a file

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

Diagrams in Swift

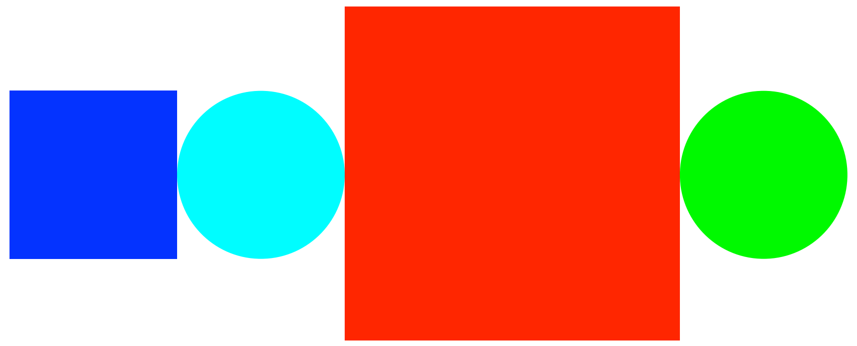
Diagrams in Objective C

```
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))
```

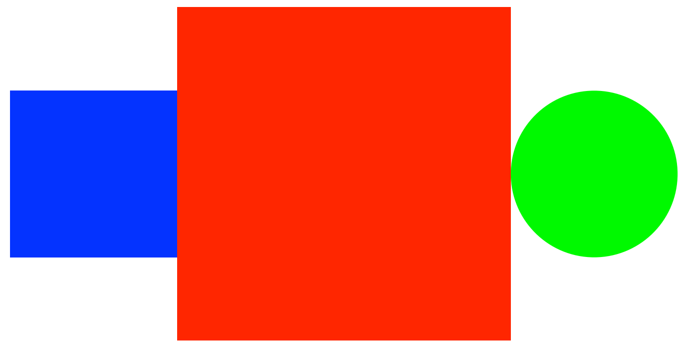


Diagrams in Objective C

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

```
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
```

Intended solution

Adding new shapes is easy:

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

This solution is *compositional*

Diagrams in Swift[1]

```
enum Primitive {
    case Ellipse
    case Rectangle
    case Text(String)
}

enum 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

```
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
```

Drawing composite diagrams

```
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

```
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:

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

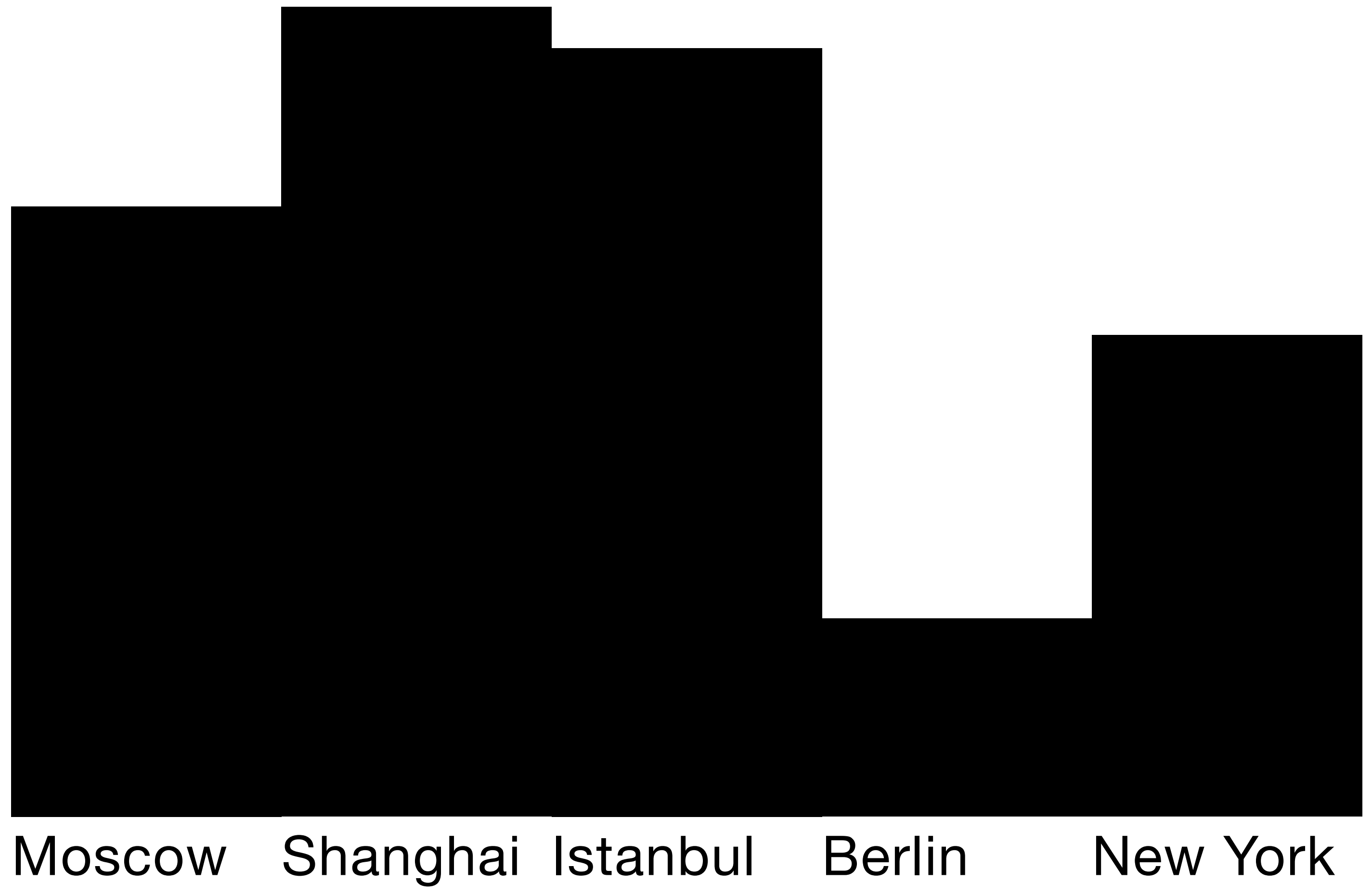
Combining lists of diagrams

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

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

Example: visualizing dictionaries

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



Generating bar graphs

```
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?