Functional Swift

@chriseidhof

Łódź wiOSłuje - August, 2014

What's Functional Programming?

- Pure
- Referentially transparent
- Typed

Our data set

```
let cities : [String:Int] =
    [ "Warszawa": 1706624
    , "Kraków":    766583
    , "Łódź":      753192
    , "Wrocław":   632930
    , "Poznań":    567932
    ]

let names = Array(cities.keys)
let populations = Array(cities.values)
```

Names

› [Poznań, Warszawa, Wrocław, Kraków, Łódź]

Populations

- [567932, 1706624, 632930, 766583, 753192]

Map

```
func addCity(s: String) -> String {
  return s + " is a city"
}

names.map(addCity)
```

```
› [Poznań is a city, Warszawa is a city, Wrocław is a city, Kraków is a city, Łódź is a city]
```

Filter

```
func isLodz(s: String) -> Bool {
  return s == "Łódź"
}

names.filter(isLodz)

> [Łódź]
```

Filter, simplified

```
names.filter({ (s: String) -> Bool in
  return s == "Łódź"
})
```

```
> [Łódź]
```

Filter, more simplified

```
names.filter({ s in
  return s == "Łódź"
})
```

```
> [Łódź]
```

Filter, even more simplified

```
names.filter({
  return $0 == "Łódź"
})
```

```
> [Łódź]
```

Filter, simplest

```
names.filter { $0 == "Łódź" }
> [Łódź]
populations.filter { $0 > 1000000 }
> [1706624]
```

Sum of an array

```
func sum(arr: [Int]) -> Int {
  var result = 0
  for i in arr {
    result += i
  }
  return result
}

sum(Array(1..<10))

> 45
```

Product of an array

```
func product(arr: [Int]) -> Int {
  var result = 1
  for i in arr {
    result *= i
  }
  return result
}

product(Array(1..<10))

> 362880
```

Reduce

```
func reduce(initialValue: Int,
            combine: (Int,Int) -> Int,
            arr: [Int]) -> Int {
  var result = initialValue
  for i in arr {
    result = combine(result,i)
  }
  return result
}
```

Reduce

```
reduce(0, +, Array(1..<10))
> 45
reduce(1, *, Array(1..<10))
> 362880
```

Sum and Product

```
let sum     = { reduce(0,+,$0) }

let product = { reduce(1,*,$0) }
```

Concatenate

```
func concat(strings: [String]) -> String {
    var result = ""
    for x in strings {
        result += x
    }
    return result
}

concat(names)

> PoznańWarszawaWrocławKrakówŁódź
```

Generics

```
func reduce<A>(initialValue: A,
               combine: (A,A) -> A,
               arr: [A]) -> A {
  var result = initialValue
  for i in arr {
    result = combine(result,i)
  }
  return result
}

reduce("", +, names)

> PoznańWarszawaWrocławKrakówŁódź
```

Adding line-breaks

```
reduce("", { $0 + "\n" + $1 }, names)
```

```
> Poznań
> Warszawa
> Wrocław
> Kraków
> Łódź
```

Making reduce more generic

```
func reduce<A,R>(initialValue: R,
                combine: (R,A) -> R,
                arr: [A]) -> R {
    var result = initialValue
    for i in arr {
        result = combine(result,i)
    }
    return result
}
```

Example: Core Image

The Objective-C way

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

A Swift Filter

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

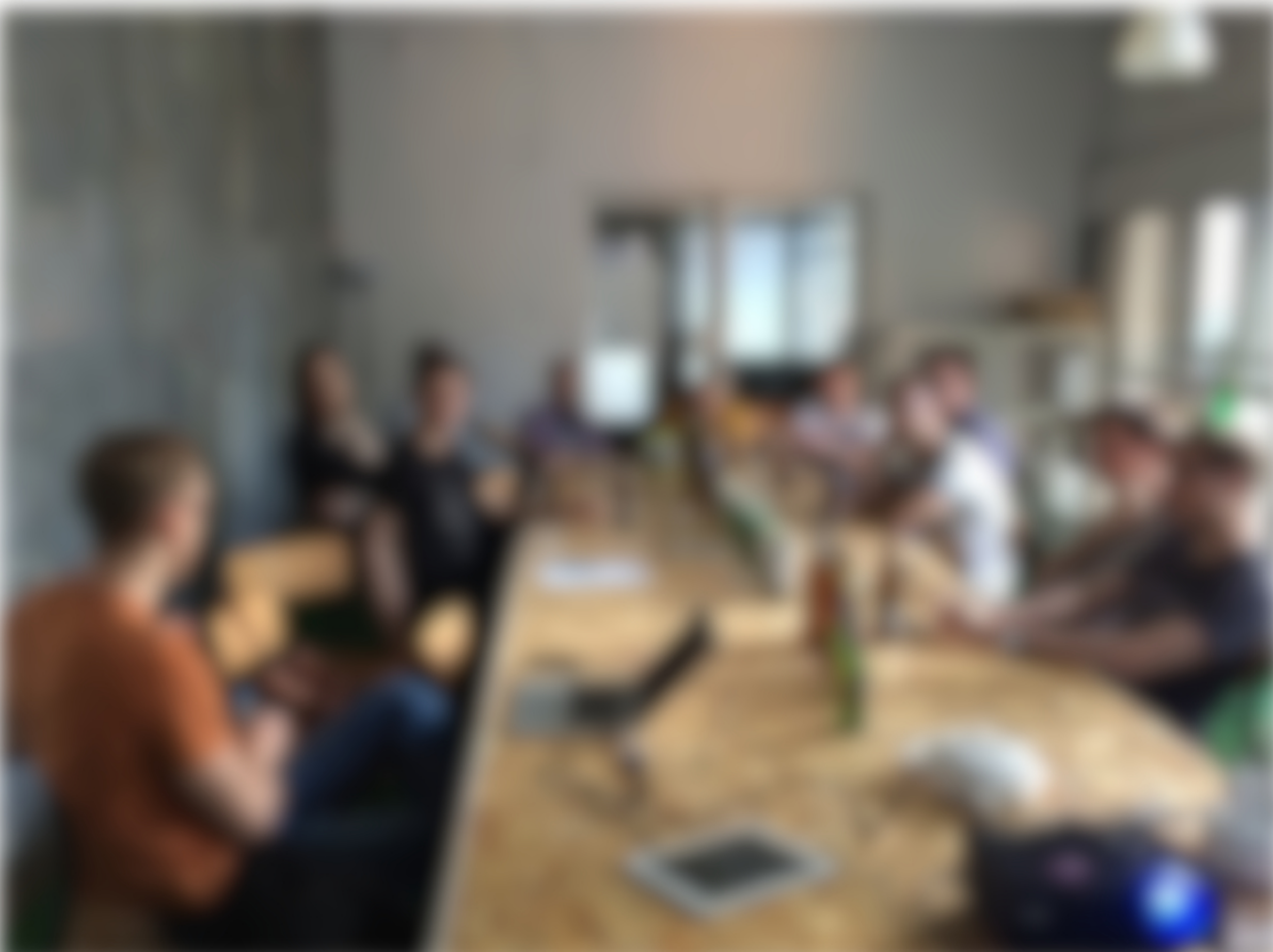
Blur

```
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
    }
}
```

Example

```
let url = NSURL(string: "http://bit.ly/1pabRsM");
let image = CIImage(contentsOfURL: url)

let blurBy5 = blur(5)
let blurred = blurBy5(image)
```



Color Generator

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

Composite Source Over

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

Color Overlay

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

Combining everything

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

Combining everything, take 2

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

Filter composition

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

Using filter composition

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

Filter composition with an operator

```
infix operator |> { associativity left }

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

Using filter composition

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

Function composition

```
func |> (f1: B -> C, f2: A -> B) -> A -> C {
    return {x in f1(f2(x))}
}
```

Example: Spreadsheet

Expressions

```
enum Expression {
    case Number(Int) // e.g. 10
    case Reference(String,Int) // A0
    case BinaryExpression(String,Expression,Expression) // 1 + A9
    case FunctionCall(String,Expression) // SUM(...)
}
```

Parsing references

```
let reference = { Token.Reference($0,$1) } </> capital <*> naturalNumber
```

Parsing expressions

```
prim =  numberOrReference <|> functionCall
                          <|> parens(expression)
```

Parsing expressions

```
let operators : [[String]] =
  [ [":"]
  , ["*", "/"]
  , ["+", "-"]
  ]
let expression = pack(operators, prim)
```

Parsing results

We can now convert this:

```
parse(expression, "SUM(A1:A9)")
```

into this:

```
Expression.FunctionCall("SUM",
  Expression.BinaryExpression(
    ":",
    Expression.Reference("A",1),
    Expression.Reference("A",9)
  )
)
```

Evaluating expressions

The result enum

```
enum Result {
    case IntResult(Int)
    case StringResult(String)
    case ListResult([Result])
    case EvaluationError(String)
}
```

The evaluation function

```
func evaluate(expressions: [Expression?]) -> [Result] {
    return expressions.map(evaluateExpression(expressions))
}
```

Evaluating an expression

```
evaluateExpression([42,10*10,A0+A1])('A1')
> 100

evaluateExpression([42,10*10,A0+A1])('A2')
> 142
```

Evaluating an expression

```
func evaluateExpression(context: [Expression?]) -> Expression? -> Result {
    return {e in e.map { expression in
        let compute = evaluateExpression(context)
        switch (expression) {
        case .Number(let x): return Result.IntResult(x)
        case .Reference("A", let idx): return compute(context[idx])
        case .BinaryExpression(let s, let l, let r):
            return evaluateBinary(s, compute, l, r)
        case .FunctionCall(let f, let p):
            return evaluateFunction(f, compute(p))
        default:
            return .EvaluationError("Couldn't evaluate expression")
        }
        } ?? .EvaluationError("Couldn't parse expression")
    }
}
```

Mixing FP and OO

```
class SpreadsheetDatasource : NSObject,
                              NSTableViewDataSource,
                              EditedRow
```

Mixing FP and OO

```
var arr: [String]

func tableView(aTableView: NSTableView,
    objectValueForTableColumn: NSTableColumn,
    row: Int) -> AnyObject {
  return editedRow == row ? arr[row] : results[row]
}
```

Mixing FP and OO

```swift
func calculateExpressions() {
    let expressions: [Expression?] = arr.map {
        if let tokens = parse(tokenize(), $0) {
            return parse(expression(), tokens)
        }
        return nil
    }
    results = evaluate(expressions)
}
```

Conclusion

FP is a massively powerful tool in your toolbox. Use it together with OO, and build awesome stuff.

objc ↑↓

Functional Programming in Swift

By Chris Eidhof, Florian Kugler and Wouter Swierstra

@chriseidhof