Introduction to Functional Programming (FP) with Swift

(Bob Raman - Dec 2014)

Contents

History

- 1930 - Alonzo Church introduced *Lambda Calculus*
- *Lambda Calculus* - formal system to investigate functions, application and recursion
  - First class citizens - functions can be parameters and can be returned
  - Anonymous
  - Main inspiration for functional programming
- 1950 - Lisp (John McCarthy)

What is Functional Programming ?

- Programming with “pure" functions
  - “pure” functions are math like.
  - $y = f(x)$
  - Given a input x the output of a function f is y
- No side-effects
- No *implicit* state
- No mutable data

Referential Transparency + Equational Reasoning

- *Referential Transparency* - a function is RT if its result depends only on the value of its inputs.
- *Equational reasoning* - “equals can be replaced by equals”.
  - let x = f a
    in … x + x
  - Substitute for (f a) for x in (x + x)
- Unlike imperative languages this allows the ability to reason about the program.

Referential Transparency + Equational Reasoning

Examples:

- func add(a: Int, b: Int) -> Int { return (a + b) }

- func getTime() -> NSDate { return NSDate() }
    - Pure FP languages have no side-effects - so no IO.
    - These languages define an action which the runtime system then executes. The action is the same each time but the result of executing it depends on the circumstances of when it is executed.
    - Think of it as passing the current environment into the method and getting an altered environment back.

Basics

Immutability in Swift

- Value Types vs Reference Types
  - Values are always copied - during assignment or passing to a function
  - In Swift, almost everything is a value type
    - struct, numbers, strings, …
    - Classes are the exception
  - Values
    - let str = “foobar”

Learning functional techniques with Swift

- Playground is too buggy and slow (in Xcode Version 6.1.1 (6A2008a))
- repl (Read Evaluate Print Loop)
    - Comes bundled with Yosemite
    - Seems to be more stable than Playground

Recursion

- Limited call stack size (8MB for objective C)
- Tail Call Optimisation
  - Perform the recursive call last
  - The compiler turns the tail call into a loop so you can have infinite stack - tail call optimisation (TCO).
- Swift only provides TCO in a limited set of cases.

Recursion - Example

```
func recurse(var aList:Array<Int>)->() {
  if aList.count == 0 {
    return
  }
  print("Count: \(aList.count)")
  aList.removeLast()
  return recurse(aList)
}

recurse([Int](count:100000,repeatedValue:1)) // works
```

Tail Recursion - Haskell

Not Tail Recursive

```
mysum [] = 0
mysum (x:xs) = x + mysum xs
```

Tail Recursive

```
sumtco [] accum = accum
sumtco (x:xs) accum = sumtco xs (accum+x)
```

FP Languages

Statically typed

- Haskell - pure
- Scala - JVM based
- Swift - iOS
- ...

Dynamically typed

- Clojure - Lisp + JVM based
- ...

Key Tenets (Principles)

- Composability
  - ability to build large programs
- Immutability
  - No mutable state means that programs are easier to reason about
    - test, understand
- Statelessness
- Lazy evaluation

Why is FP becoming more mainstream?

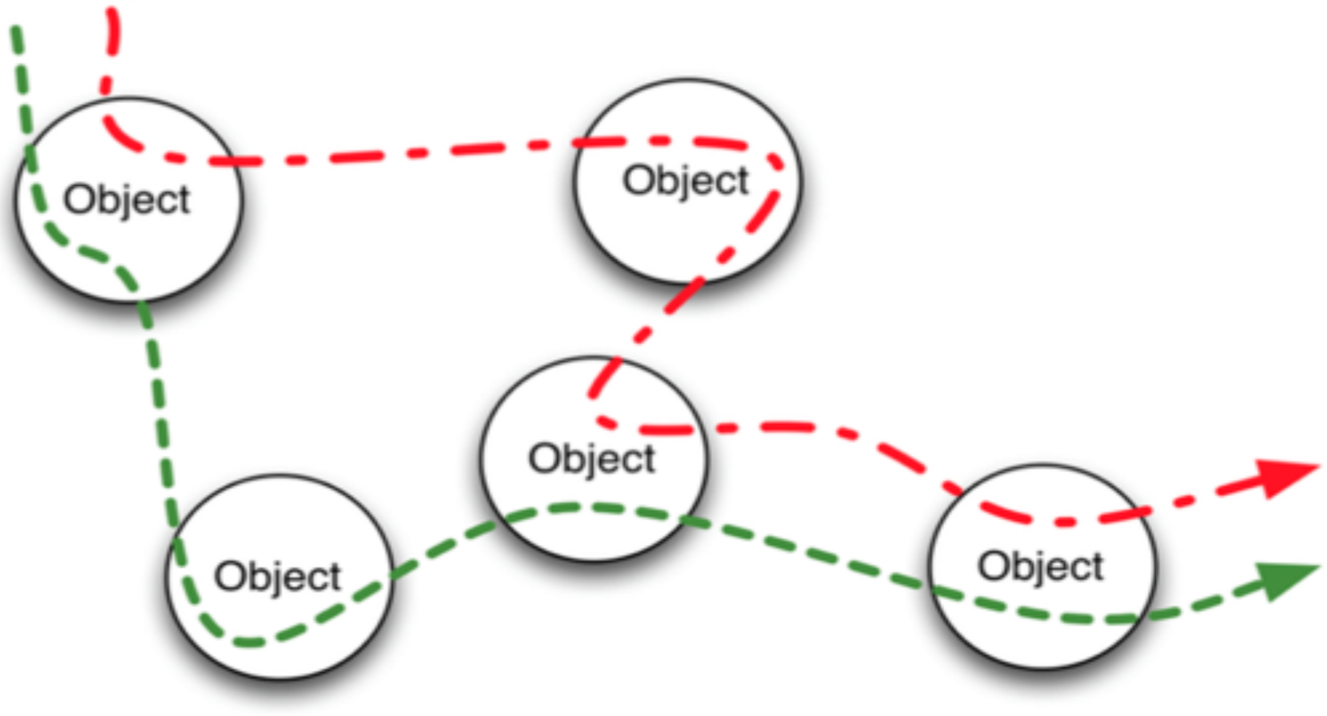
- Concurrent and multicore programming become easier as there is no state
  - Apple A8 (iPhone 6)
    - Dual Core 1.38 Ghz Armv8
  - Apple A8x (iPad Air 2)
    - A8 + 1 core
- Gives us the tools to cope with complex software

OOP vs Functional

OOP vs Functional Thinking

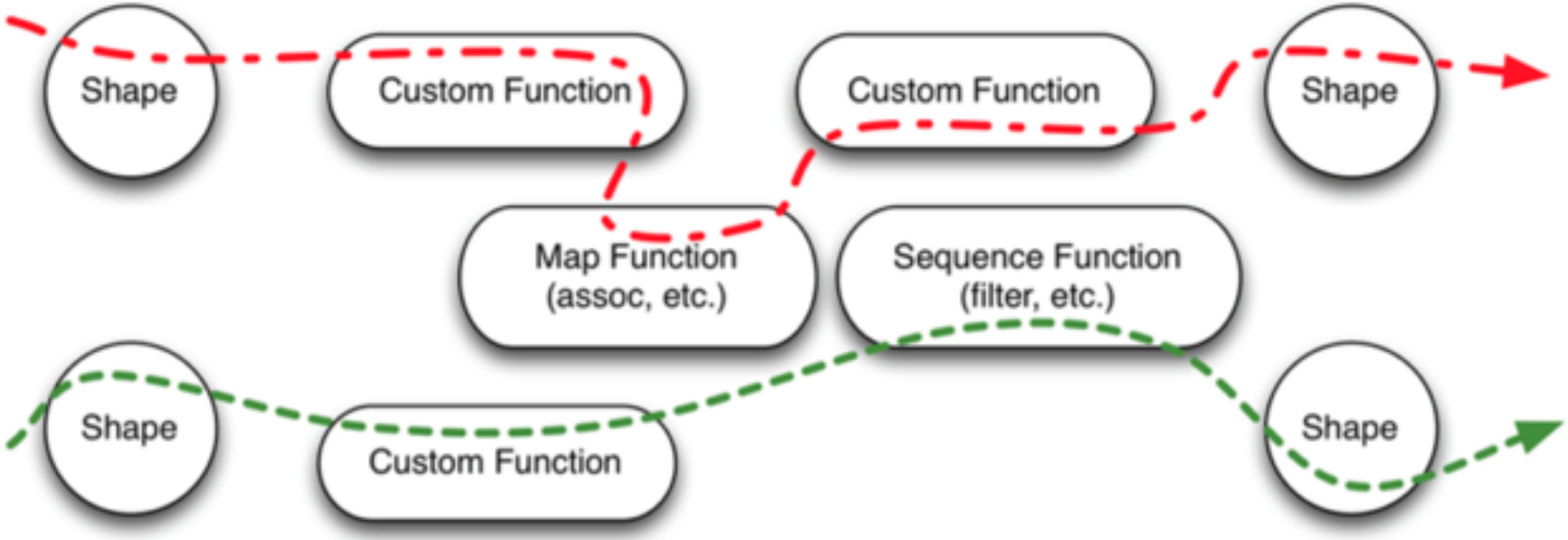
"OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimising moving parts." - Michael Feathers

OP vs Functional Thinking - OOP [3]



Stable relationships and varying paths

OOP vs Functional Thinking - Functional [3]



Many specialized flows and shapes

FP Language Features

List Comprehension

- Apply an anonymous function across a list of elements

```
func square(numbers: [Int]) -> [Int] {
  var squares = [Int]()
  for n in numbers {
        squares.append(n*n)
        }
  return squares
}
```

List Comprehension

- *Map* - apply a function to a list of values and return a new list.
  - let squares = [1,2,5].map { $0 * $0 }
    - This is an example of *Higher Order Function (HOF)* - passing a function "{ $0 * $0 }" to another function “map"
    - A function is called HOF if it either takes a function as argument or returns a function as a result.
  - Closure expression
    - let squarer = { (i:Int) -> Int in return i*i }
    - [1,2,5].map(squarer)

List Comprehension

- *Filter* - apply a filter out function to a list of values and return a new list
  - let odds = [1,2,5].filter { $0 % 2 == 1 }

- *Reduce* (aka fold) - Iterate over an array and produce a single result
  - let sum = [1,2,5].reduce(0) { $0 + $1 }

Currying

- Named after Haskell Curry (mathematician)
- Takes a function with multiple parameters and converts it into a function with a single parameter
    - ie. take a function that takes a parameter and return another function which takes a parameter and does some processing.
- Main benefit is that it allows for specialised and generalised functions.
- Define for the general case and allow users to specialise.

Currying

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

- Return a closure expecting a second argument
  ```
  func add1(x: Int) -> (Int -> Int) {
      return {y in x + y}
  }
  ```

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

Currying

- Why is this useful ?
    - increment = add1(1)
    - increment(x)

- Another way of representing currying.
    - func add3(x:Int)(y:Int) -> Int { return x + y }
    - add3(1)(y:10) // maybe a bug???

Lexical Closure

- Generally functions with free variables that are bound in the lexical environment
  - Capture variables that are declared in their context
- Related concept is lambda
  - Refers to anonymous functions

Lexical Closure

```
func sum() -> (Int) -> () {
    var sum = 0
    func inner(i: Int) {
        sum += i
        println("sum = \(sum)")
    }
    return inner
}
let f = sum()
f(10) // Outputs 10
f(20) // Outputs 30
```

Lazy Evaluation [2]

- Three ways that laziness can be used:
  - lazy var image
    - “image” is only created when accessed
    - useful when for instance loading an image from disk
  - Sequences
    - Generators - A process that generates new elements on request
    - Sequences - Allow for replay and rewind of generators
  - @autoclosure
    - Lazy evaluation of expressions
    - Used in the implementation of “assert()” as Swift does not use cpp

Generators

- for (x in xs) {
  //.. }
- Compiler converts above to:

```
var _g = xs.generate()
while let x = _g.next() {
//
}
```

- _g.next() - generates values on the fly rather than store the values in memory

Protocols

- protocol GeneratorType {

  typealias Element

  func next() -> Element?

  }

- protocol SequenceType {

  typealias Generator: GeneratorType

  func generate() -> Generator

  }

Infinite List

- List of all positive integers

```
class Integers : SequenceType {
    func generate() -> GeneratorOf<Int> {
        var n = -1
        return GeneratorOf { ++n }
    }
}
```

Infinite List

- extension LazySequence {

```
    var first: LazySequence.Generator.Element? {
        for x in self {
            return x
        }
        return nil
    }
}
```

Infinite List

```
var integers = lazy(Integers())
var value = integers
            .filter {$0 % 2== 1}
            .map{$0 * $0}
            .filter{$0 > 100}
            .first!
```

@autoclosure

```
func f(pred: @autoclosure () -> Bool) {
    if pred() {
        println("It's true")
    }
}

// Can pass f({expensiveOp > 1}) as:
f(expensiveOp > 1)
```

Optionals

- Optionals - Avoid nil handling or defensive if statements.
  - Let anyone dealing with an optional variable know that there may be no value - similar to Null Object pattern
  - Provides compile time checks that can prevent common programming errors at runtime
  - Example
    - class Dog { var name: String? }

Optionals...

```
func findDogOrigin(type:String) -> String? {
    if (type == "German Shepherd") {
        return "Germany"
    }
    return nil
}

let origin = "Dog is from " + findDogOrigin(“Bulldog”)
// compile-time error - unwrapped optional!
```

Optionals...

```
// Test that the optional is not nil and then unwrap
let origin = findDogOrigin("Bulldog")
if origin != nil {
   let msg = "Dog is from" + origin!
   println(msg)
}
```

Optionals

```
// Unwrap using optional binding
// In other languages such as Scala you would use
// Pattern matching to de-construct the optional

if let origin1 = findDogOrigin(“Bulldog") {
   let msg = "Dog is from" + origin1
   println(msg)
}
```

Also optional chaining support

Pattern Matching

```
func welcomeMessage(name: String, age: Int) -> String? {
    switch (name, age) {
      case (_, let a) where a <= 0: return nil
      case (_, 0..<18): return "not grown up"
      case ("john", 18): return "sorry john, not allowed"
      case (let n, _): return "hi \(n), welcome!"
    }
}

welcomeMessage("alice",17)
```

Pattern Matching - Limitations with Arrays

```
func testArrayPattern(array: [Int]) -> String? {
    switch (array) {
      case (10,_): return "Found a 10 at position 0"
      case (10,10,_): return "Found a 10 followed by a 10 at start"
    }
}
```

Cannot code like “testArrayPattern” out of the box!

For those who are interested it is possible to solve the array issue using “discriminated unions” in Swift but the solution is quite long winded. (See http://vperi.com/2014/06/07/list-traversal-with-discriminated-unions-in-swift/)

FP and GOF Design Patterns [1]

- Design Patterns in FP either:
  - Absorbed by the language
  - Exists but implementation is different
  - Not possible to be supported in language

FP and GOF Design Patterns

- Currying
  - Acts as a “factory” for functions.

```
func add(x: Int) -> (Int -> Int) {
  return {y in x + y}
}
let increment = add(1)
```

Create a function that returns other functions based on some criteria

Essense of factory

FP and GOF Design Patterns

- Template Method - Groovy example

```
abstract class Customer {
    def plan
    def Customer() {
    plan = []
    }
    def abstract checkCredit()
    def abstract ship()
    def process() {
    checkCredit()
    ship()
    }
}
```

FP and GOF Design Patterns

- Template Method in Swift

```
class Customer {
   var plans: [ () -> () ] = []
   init() {
       plans.append({ println("Check credit") }) // check Credit
       plans.append({ println("Ship me") }) // ship
    }
     func process() {
      for plan in plans {
             plan()
       }
    }}
```

Type Inference

- Compiler works out the type of a variable or function without the programmer being specific.
    - Swift does not appear to use Hindley-Milner algorithm
    - Swift supports some type inference - “operates at the level of a single expression or statement". i.e. must be able infer type by checking the expression or its sub-expressions.

Type Inference

- Use *typealias* to convey your intention with type
- Let the compiler do the work for you!

```
typealias UserInput = String
struct SanitizedString {
    let value : String = ""
    init(input : UserInput) {
        value = sanitize(input)
    }
    func sanitize(input: UserInput) -> String {
        // do something
        return input }
}
```

Composability

- Currying helps you compose functions.

```
infix operator |> { associativity left precedence 80 }
    func |> <A,B> (x: A, f: (A -> B)) -> B {
            return f(x)
     }

    func sqr(x:Int) -> Int { println("sqr”); return x*x }
    func incr(x:Int) -> Int { println("incr”); return x + 1 }
    let val = (4 |> sqr |> incr)
```

Summary

- Whilst not a pure functional language, Swift has enough feature for you to write code in a functional manner.
    - immutability
    - lazy evaluation
    - composability
    - (Some)Pattern matching

References

- [1] http://www.ibm.com/developerworks/java/library/j-ft10/index.html
- [2] “Laziness in Swift" by Maciej Konieczny, https://vimeo.com/105219529
- [3] Functional Programming for the Object-Oriented Programmer, Brian Marick