Functional Programming is a **style** whose underlying model of computation is the *function.*

$$f(x)$$

# Table of Contents

# Functional Programming Notes

This is a collection of my notes and references to articles and books on functional programming. Examples and links are specific to Scala most of the time.

# Why FP matters?

When writing programs in a modular way, we divide a problem into sub-problems, solve them separately, and glue them together to form a solution. The ways we divide the problem directly depends upon the ways we can glue them together. Therefore, to increase one's ability to modularize a problem conceptually, one must provide new kinds of glues in the programming language. FP languages provide two new, very important kind of glue: **Higher-order functions** and **Lazy evaluation**.

**Higher-order functions** enable simple functions to be glued together to make more complex ones. Define a general higher-order function ( `foldr` ) and some particular specializing functions ( `sum` etc.).

Examples

- sum = foldr (+) 0
- product = foldr (*) 1
- anytrue = foldr or False
- alltrue = foldr and True
- length = foldr count 0 // where count a n = n + 1
- map f = foldr (Cons .f) Nil
- summatrix = sum . map sum

**Lazy evaluation** makes it practical to modularize a program as a **generator** that constructs a large number of possible answers, and a **selector** that chooses the appropriate one. Without lazy evaluation this would not always be practical (or even possible, in the case of infinite generators).

Reference

- https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf
- foldl.com and foldr.com

# Functions

- When we say functions are "first class", we mean they are just like everyone else...so normal classcoach?. We can treat functions like any other data type and there is nothing particularly special about them - store them in arrays, pass them around, assign them to variables, what have you.

## Higher Order Functions

Enable simple functions to be glued together to make more complex ones. Define a general higher-order function ( `foldr` ) and some particular specializing functions ( `sum` etc.).

Examples

- `sum = foldr (+) 0`
- `product = foldr (*) 1`
- `anytrue = foldr or False`
- `alltrue = foldr and True`
- `length = foldr count 0 // where count a n = n + 1`
- `map f = foldr (Cons .f) Nil`
- `summatrix = sum . map sum`

# Pure Functions

From Mostly Adequate Guide to FP

- A pure function is a function that, given the same input, will always return the same output and does not have any observable side effect.
- Depending on external state increases the cognitive load by introducing an external environment. Erlang creator, Joe Armstrong: "The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana...and the entire jungle".
- The philosophy of functional programming postulates that side effects are a primary cause of incorrect behavior. It is not that we're forbidden to use them, rather we want to contain them and run them in a controlled way. We'll learn how to do this when we get to *functors* and *monads* in later chapters, but for now, let's try to keep these insidious functions separate from our pure ones.

- You can transform some impure functions into pure ones by delaying evaluation

## Memoization

- Pure functions can always be cached by input. This is typically done using a technique called memoization.

## Referential Transparency

- A spot of code is referentially transparent when it can be substituted for its evaluated value without changing the behavior of the program.

## Parallel Code

- Here's the coup de grace, we can run any pure function in parallel since it does not need access to shared memory and it cannot, by definition, have a race condition due to some side effect.
- Because we are not encoding order of evaluation (unlike in imperative coding), declarative coding lends itself to parallel computing. This coupled with pure functions is why FP is a good option for the parallel future - we don't really need to do anything special to achieve parallel/concurrent systems.

# Challenges

- We have to juggle data by passing arguments all over the place
- We're forbidden to use state, not to mention effects.

How does one go about writing these masochistic programs? **Curying is the answer.**

# Currying

- You can call a function with fewer arguments than it expects. What is returned is a function that takes the remaining arguments.

# Partial Application

# Usage

- When we spoke about pure functions, we said they take 1 input to 1 output. Currying does exactly this: each single argument returns a new function expecting the remaining arguments.
- It is one tool for the belt that makes functional programming less verbose and tedious.

# Coding by Composition

Composition will be our tool for constructing programs and, as luck would have it, is backed by a powerful theory that ensures things will work out for us. Let's examine this **Category theory**.

# Associative Property

When compositing functions, by math, the composition must satisfy associative property, meaning it doesn't matter how you group two of them. So, should we choose to uppercase the string, we can write:

```
compose(toUpperCase, compose(head, reverse));

// or
compose(compose(toUpperCase, head), reverse);
```

# Map's Composition Law

Reference (Principled Refactor)

```
var law = compose(map(f), map(g)) == map(compose(f, g));
```

# Pointfree coding

- http://drboolean.gitbooks.io/mostly-adequate-guide/content/ch5.html
- Monoids and Combinators enable us to write pointfree code Reference

# Debugging

http://drboolean.gitbooks.io/mostly-adequate-guide/content/ch5.html

# Algebraic Data Types

> **How do you model data in FP?: ADT. Using ADT, we model data with logical *ors* and logical *ands***

- ADT's are only data. No behavior at all.
- In Scala, when we hear ADT, it means Sum Type.
- ADT's doesn't feature sub-type polymorphism, but only *combination/composition* of data types.
- In ADT, structure of the code follows structure of data.

# Enumerated Types

- All the possible values are enumerated as there are finite set of values. E.g. Seasons, Switches
- `case` *classes* and *objects* are used to define enumerated type
- When defining an ADT, we need to
    - First define the type
    - Second define the domain of the type i.e. all possible values of that type. These valuses are called *value constructors*.
- Example

```
//Intoruduce the type
sealed abstract class Season

//Define value constructors
case object Summer extends Season
case object Winter extends Season
case object Fall extends Season
case object Spring extends Season
```

- Pattern matching simplifies the way we work with enumerated types
- It is not always possible to enumerate all possible values of a type, for e.g. `Color`. This is where the sum and product types are helpful.

# Sum Types

- **This or That**
- All the values of a sum type are clearly expressed as a sum of all of its value

constructors.
- Sum types provide individual value constructors for each and every value of that type.
- In Scala, Sum Types are encoded by subclassing
- Examples: Season, Boolean

# Product Types

- **This and That**
- Sometime we can't enumerate all values of a particular type. For e.g. it is not worth to enumerate all possible colors.
- Example

```
sealed case class RGBColor(red:Int, blue:Int, green:Int)
```

# "Sum of Product" Types

- Combination of sum and product types. Sum and Product type together make ADT.
- Mostly have nested definitions

# Recursive Types

- TBD: http://tpolecat.github.io/presentations/cofree/slides

# Standard Library

| Intuition | Type | Value Constructors |
| --- | --- | --- |
| Computations that may fail to return a value | `Option` | `None` , `Some` |
| Computations that may return this or that | `Either` | `Left` , `Right` |
| Computations that may fail with an Exception | `Try` | `Success` , `Failure` |
| Computations that may return many answers | `List` | `Nil` , `::` |

# Transforming ADT

- 2 patterns
  - Pattern Matching

    - Polymorphism
- Pattern matching is preferred as you can see all the implementations at one place. You put the pattern matching in base trait.

# Best Practices

- Whenever a new type is defined, we should define higher-order functions for processing it. Page 8 of Why FP matters?

# References


- http://tpolecat.github.io/presentations/algebraic_types.html
- https://gleichmann.wordpress.com/2011/01/30/functional-scala-algebraic-datatypes-enumerated-types/
- https://gleichmann.wordpress.com/2011/02/05/functional-scala-algebraic-datatypes-sum-and-product-types/
- https://gleichmann.wordpress.com/2011/02/08/functional-scala-algebraic-datatypes-sum-of-products-types/

# Pattern Matching

- Guard conditions can be used to avoid nested pattern-matching
- Pattern matching or OOP polymorphism?
    - Use polymorphism when
        - new sub-classes are expected to be added in future
    - Use pattern matching when
        - new methods are expected to be added to existing class hierarchy
        - the behavior is spanning multiple types
- apply, applySeq

# References

- https://www.youtube.com/watch?v=OpO9uhI22ZA

# Sequencing Computation

- FP is all about transforming values. This is what we can do without using side-effects.
- `A => B => C`. This is sequencing computation.
- fold, map and flatMap
- fold is abstraction over structural recursion on ADT. It is a generic transform for any ADT.
-

# Category Theory

Category theory will play a big part in app architecture, modeling side effects, and ensuring correctness.

Simple and Good Cheat Sheet: https://gist.github.com/cb372/b1bad150e0c412fb7364

## Monad

Simple and informative: https://www.youtube.com/watch?v=Mw_Jnn_Y5iA Monads are Elephants

- Part 1
- Part 2
- Part 3

Programming with Monads

## Monoid

- Monoid without tears

# Property Based Testing

- Unit tests: Reason by example
- PBT: Reason by proof
- Generate -> Run -> Shrink

**Intuition**

1. Specify the behavior of a unit of code as a `Prop`
2. ScalaCheck generates test data to falsify the property until it is exhausted
3. If the property holds true for all such generated test data, then the `Prop` is assumed to pass the test

**Flow**

1. Create an instance of `org.scalacheck.Prop` say `p`
2. Test it by calling `check` function on it `p.check`

# Create

- Universally qunatified Prop can be created using `Prop.forAll`
- `forAll` takes function as the parameter
- The function takes any type as input parameter and returnes either a `Boolean` or another `Prop`
- **Combine** `Prop` using `all, &&, ||, atLeastOne, ==`
- Group related props by
  - Extending from `Properties`
  - Using `include`

# Generate

- `org.scalacheck.Gen`
- Can generate **any** value for a type or a **subset** of values
- Are a Monad so we can sequence/chain them to produce new ones
- Composable: `map` , `flatMap` , `filter` , `suchThat`
  - Using `suchThat` means ScalaCheck treats filtered values as *discarded*
- Are edge-case biased
- ScalaCheck has generators for
  - Primitives

- `Throwable` , `Date`
- `Option` , `Either` , `Tuple1` to `Tuple9` , `Function1` to `Function5`
- Collections: `Array` , `List` , `ArrayList` , `Map` , `Stream` , `Set`
- Helpers: `alphaChar` , `alphaNumChar` , `alphaStr` , `identifier` etc.
- Higher-order: `choose` , `oneOf` , `someOf` , `listOf1`
- Distributions: Random, `Prop.frequency`

- `for` comprehension can be used to generate custom types
- `Arbitrary` is a canonical way of generating data for a specific type. There can be only one canonical generator for type in a given scope. See this video from 41th to 44th min.
- `arbitrary` is a way to convert an `Arbitrary` into a Gen. See this video from 43.30 to 44.10 min.

# Run

-
- ScalaTest
- Specs2
- ScalaMeter
- Simulant

# Shrink

```
scala> implicitly[Shrink[String]].shrink("asdf")
Stream[String] = Stream(as,?)
scala> implicitly[Shrink[String]].shrink("asdf").force
Stream[String] = Stream(as,df,adf,asd,sdf,asf)
scala> implicitly[Shrink[String]].shrink("as").force
Stream[String] = Stream(a,s)
scala> implicitly[Shrink[String]].shrink("a").force
Stream[String] = Stream("")
scala> implicitly[Shrink[String]].shrink("").force
Stream[String] = Stream()
```

# Test Distribution

- Use `collect` and `classify` to examine the distribution of the generated test data
- We can nest both of them to get multi-level grouping

# Patterns

- Symmetry: There and back again (Click for e.g.)
- Multiple paths:
- Induction
- Invariants (Click for e.g.)
    - Idempotence
    - Consistency

# References

- ScalaCheck Cookbook
- ScalaCheck Magic
- Getting the most out of ScalaCheck
- Categories of Properties
- Patterns

# New Generators Project

- Data standard manual
- Names
    - Falsehoods Programmers believe about Names
    - Dublin core name representations
    - First name
        - UK female dataset (Rank, Name, Count)
        - UK male dataset (Rank, Name, Count)
    - Last name
        - US Census - 1990
        - US Census - 2000
    - Honorofics/Titles)
        - By Locale
    - Git repo of Name Database of most countries
    - Gender determination API

# Patterns

1. Selfless Trait Pattern
2. Stackable Trait Pattern

# Reference

- Intro to FP

# Lessons Learned

Compiling the following code

```
def fetch[T](jobId: String, contentType: ContentType)(implicit context: SparkContext,
config: JobConfig) {
    val items: RDD[String] = context.textFile(config.readInputPath(contentType))
    null
  }
```

Will give the following warning

```
[WARNING] warning: a pure expression does nothing in statement position; you may be om
itting necessary parentheses
[WARNING]     null
[WARNING]     ^
```

Value discarding: Making the return type as Unit means that, any value returned from the function is automatically discarded and a Unit is returned (Similar to void in Java)

```
def fetch[T](jobId: String, contentType: ContentType)(implicit context: SparkContext,
config: JobConfig): Unit = {
    val items: RDD[String] = context.textFile(config.readInputPath(contentType))
  }
```

Reference

- Value discarding section
-
    - http://stackoverflow.com/questions/18368346/quite-confused-about-this-code-snippet-return-types-with-without
- http://stackoverflow.com/questions/23206201/scala-expression-evaluation

# Buzz Words

FP world has lot of terminology that programmers coming from an OOP background sometimes struggle with (or at least I did). I list some of those words/phrases here with some introductory reading material to understand them.

| Buzz | Details |
|---|---|
| Referential Transparency | tbd |
| Higher-order functions | tbd |
| Memoization | tbd |
| Sum and Product Types | - Enumerated types<br>- Sum and Product Types<br>- Sum of Products |
| ADT | Abstract Data Types |
| Universal quantifier | Is a logical statement that applies to all elements of a set. For e.g. `forall` in Scala collections and `forAll` in ScalaCheck. |
| Existential quantifier | Is a logical statement that applies to at least one element of the set. For e.g. `exists()` in Scala collections and `exists` in ScalaCheck. |
| Functor | tbd |
| CoFunctor | tbd |
| Applicative | tbd |
| Monad | tbd |
| Monoid | tbd |
| Semigroup | tbd |
| Arrow | tbd |
| Isomorphisms | tbd |
| Dependent Types | tbd |
| 0:14 | tbd |
| 0:15 | tbd |

# References

## Talks

- Functional Design Patterns Good at explaining Monoids, Monads etc.
- Functional type system - Domain Driven Design
- Handling errors in functional world - Railway Oriented Programming
- Building end to end functional app
- Functional Patterns for Scala Beginners
- Good for patterns on ADT, Pattern Matching, Sequence Computation
- http://scalaupnorth.com/2015.html
- http://tpolecat.github.io/presos.html

## Website

- http://tpolecat.github.io/
- http://fsharpforfunandprofit.com/
- http://drboolean.gitbooks.io/mostly-adequate-guide/

## Abstract Data Types

- http://tpolecat.github.io/presentations/algebraic_types.html
- https://gleichmann.wordpress.com/2011/01/30/functional-scala-algebraic-datatypes-enumerated-types/
- https://gleichmann.wordpress.com/2011/02/05/functional-scala-algebraic-datatypes-sum-and-product-types/
- https://gleichmann.wordpress.com/2011/02/08/functional-scala-algebraic-datatypes-sum-of-products-types/
- https://speakerdeck.com/mpilquist/a-tour-of-functional-structures-via-scodec-and-simulacrum

## Coding Conventions

- https://blog.jetbrains.com/scala/2011/03/10/signature-matters/