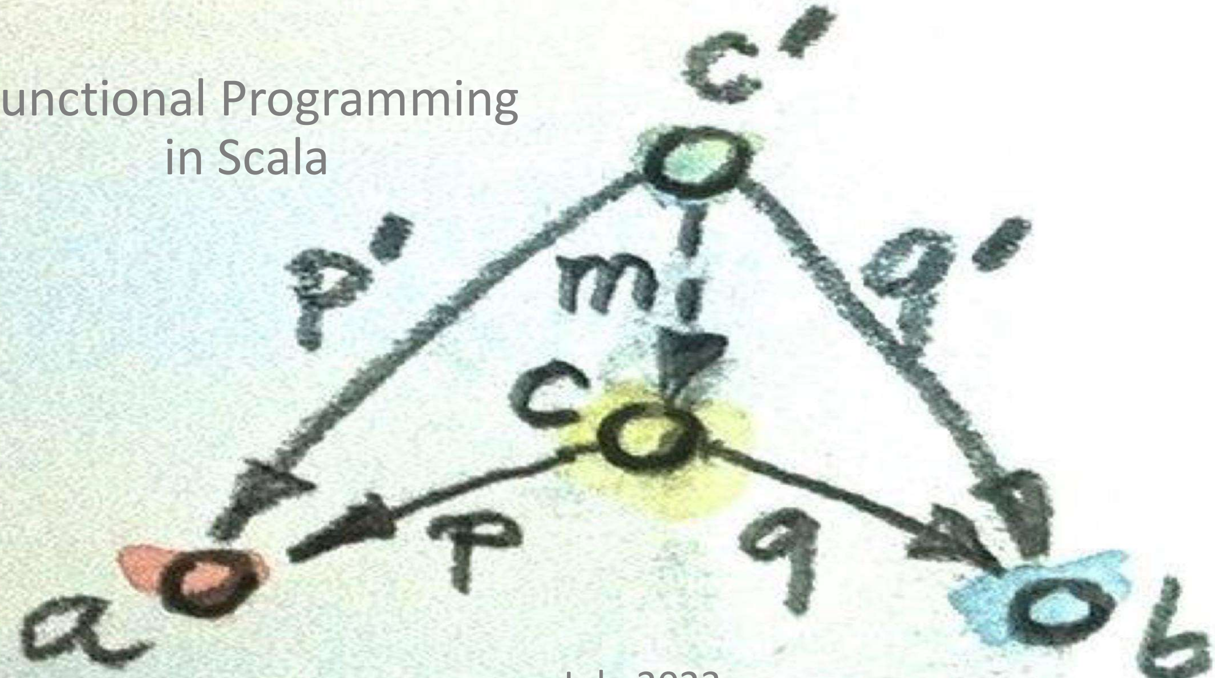
Functional Programming in Scala



July 2023

Prerequisites – Not Really ☺

- A reasonable background in functional programming, especially in Scala
- Recommended prerequisite course: “**Principles of Functional Programming in Scala**”, Coursera Course by Martin Odersky
- Other experiences in some other functional programming languages, such as Haskell, F#, Clojure, etc. would be fine.

Textbook

Functional Programming in Scala,

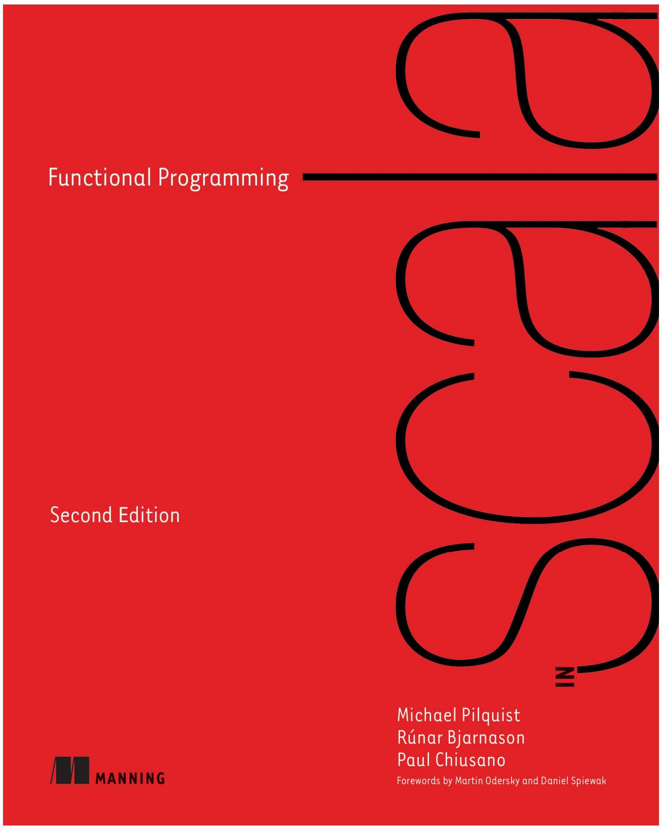
2nd Edition,

Michael Pilquist,

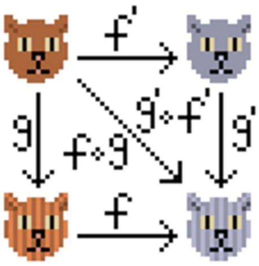
Rúnar Bjarnason, and

Paul Chiusano,

Manning, 2023



Course Contents



Tools

- Scala Language 3.3.0
- IDE: VS Code or IntelliJ

Functional Programming

Functional programming

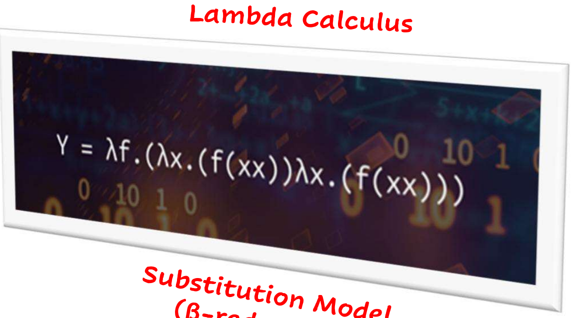
- FP is a *declarative-style* programming.
- Functional programs are composed of nested *pure functions*.
- A pure function is one with *referential transparency* (and therefore *no side effects*).
- Everything is basically *immutable*.
  - Use *recursions* instead of loops
- FP is based on *lambda calculus* and *category theory*.

Alonzo Church
(1903-1995)

Core Concepts of Functional Programming

Pure Functions
Referential Transparency
Composition
High-Order Functions
Immutability
Currying
Recursions
Parametric Polymorphism
Algebraic Data Types
Higher Kind Types
Lazy evaluation

Lambda Calculus



Substitution Model
(β-reduction)

Imperative Programming

- Tell **how** to do it.

- **Mutability**

Root of all Evils!

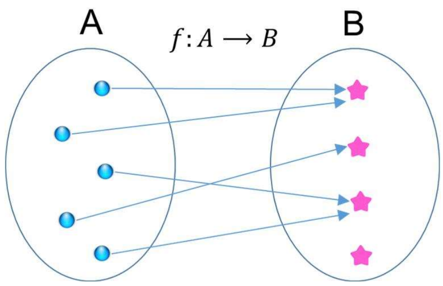
```
def qsort(xs: Array[Int], low: Int, high: Int) {
  def swap(i: Int, j: Int) = {
    val temp = xs(i)
    xs(i) = xs(j)
    xs(j) = temp
  }

  val pivot = xs((low + high) / 2)
  var i = low
  var j = high
  while (i <= j) {
    while (pivot < xs(j)) j -= 1
    while (pivot > xs(i)) i += 1
    if (i <= j) {
      swap(i, j)
      i += 1
      j -= 1
    }
  }
  if (low < j) qsort(xs, low, j)
  if (i < high) qsort(xs, i, high)
}
```

Declarative Programming

- Tell **what** to do.
- **Immutability**

```
def qsort(xs: List[Int]): List[Int] = {
  if (xs.size <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    val (l, e, r) = partition(xs, pivot)
    List.concat(qsort(l), e, qsort(r))
  }
}
```

Pure Functions



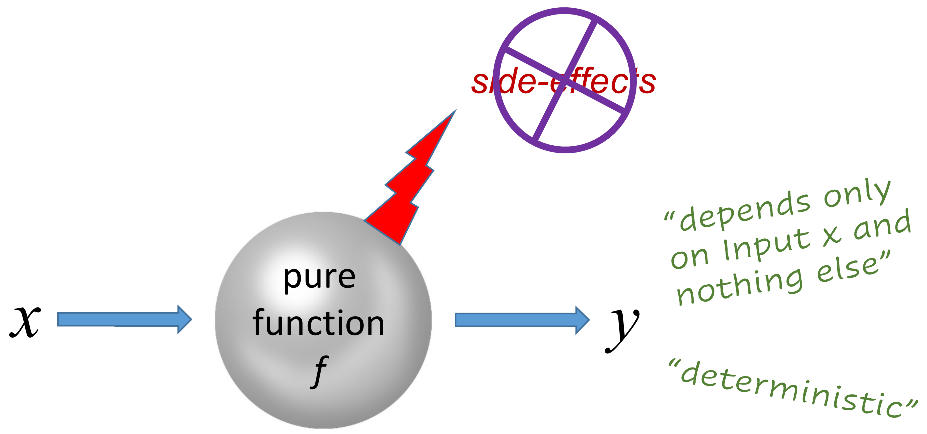
FP means programming with **pure functions**.

A pure function is one with **no side effects**.

f : A => B

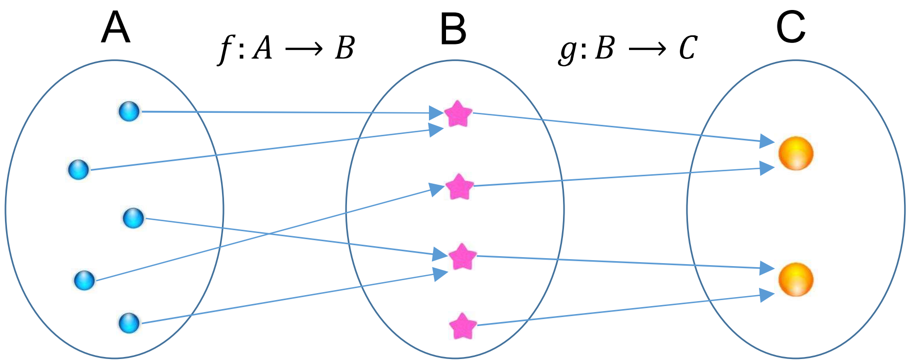
- Relates every value *a* of type **A** to exactly one value *b* of type **B** such that *b* is determined solely by the value of *a*.

A **pure** function has **no observable effect** on the execution of the program **other than to compute a result given its inputs**.



y = sin(x)

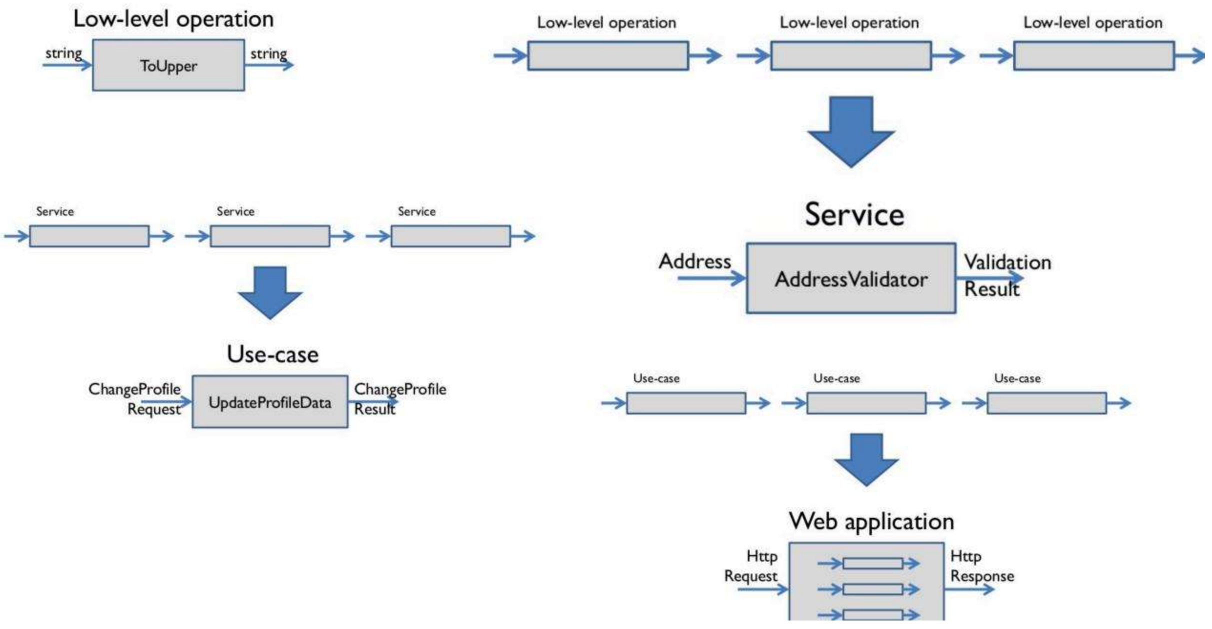
Functions and Types (Morphisms and Sets)



$$g \circ f : A \longrightarrow C$$

Modular and Scale

Composition is everywhere!



Side Effects

- Modifying a variable
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error
- Printing to the console or reading user input
- Reading from or writing to a file
- Drawing on the screen

etc.

A monad is just a monoid in the category of endofunctors.

What's the problem?

Functional Programming

- Functional programming is a restriction on *"how" we write programs*, but not on what programs we can express.
- Over the course of this tutorial, we'll learn how to express all of our programs without side effects, and that includes programs that perform I/O, handle errors, and modify data.
- **General guideline**: Implement programs with a pure core and a thin layer on the outside that handles effects.

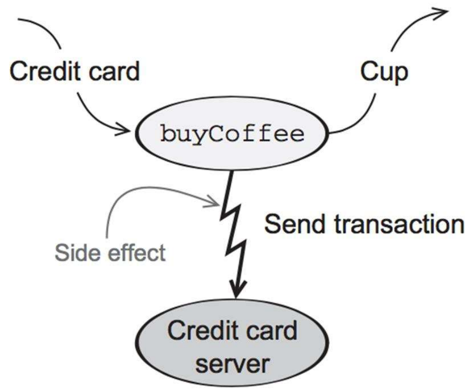
A Scala Program with Side Effects

Side effect.
Actually charges the credit card.

```
class Cafe {
  def buyCoffee(cc: CreditCard): Coffee = {
    val cup = Coffee()
    cc.charge(cup.price)
    cup
  }
}
```

Charging a credit card involves some interaction with the outside world.

Problem Due to Side Effects



Can't test buyCoffee without credit card server. Can't combine two transactions into one.

The code is difficult to test.

It's difficult to reuse **buyCoffee**.

Adding a Payment Object … but …

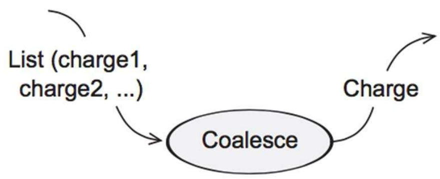
Side effect.

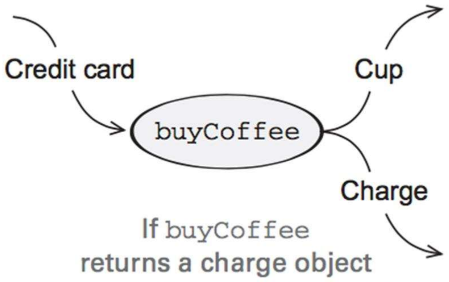
```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payments): Coffee = {
    val cup = Coffee()
    p.charge(cc, cup.price)
    cup
  }
}
```

It's still difficult to reuse **buyCoffee**.

Though side effects still occur when we call **p.charge(cc, cup.price)**, at least regained some testability via **Payment** *interface* or mock objects.

Functional Solution





If `buyCoffee` returns a charge object instead of performing a side effect, a caller can easily combine several charges into one transaction. (and can easily test the `buyCoffee` function without needing a payment processor).

The functional solution is to eliminate side effects and have **buyCoffee** *return the charge as a value* in addition to returning the **Coffee**.

Separated the concern of *creating* a charge from the *processing* or *interpretation* of that charge.

Functional Solution

```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = Coffee()
    (cup, Charge(cc, cup.price))
  }
}
```

`buyCoffee` now returns a pair of a `Coffee` and a `Charge`, indicated with the type `(Coffee, Charge)`. Whatever system processes payments is not involved at all here.

Charge as a First-Class Value

```
case class Charge(cc: CreditCard, amount: Double) {
  def combine(other: Charge): Charge =
    if cc == other.cc then
      Charge(cc, amount + other.amount)
    else
      throw new Exception("Can't combine charges to different cards")
}

class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = ???

  def buyCoffees(cc: CreditCard, n: Int): (List[Coffee], Charge) = {
      val purchases: List[(Coffee, Charge)] = List.fill(n)(buyCoffee(cc))
      val (coffees, charges) = purchases.unzip
      (coffees, charges.reduce((c1,c2) => c1.combine(c2)))
  }
}
```

Additional Benefit of First-Class Value

Making **Charge** into a first-class value has other benefits:

➔ more easily assemble business logic for working with these charges

```
def coalesce(charges: List[Charge]): List[Charge] =
  charges.groupBy(_.cc).values.map(_.reduce(_ combine _)).toList
```


Referential Transparency and Purity

An expression ***e*** is referentially transparent
if, for all programs ***p***, all occurrences of ***e*** in ***p***
can be replaced by the result of evaluating ***e***
without affecting the meaning of ***p***.

A function ***f*** is **pure** if the expression ***f(x)*** is referentially transparent
for all referentially transparent ***x***.

```
scala> val x = "Hello, World"
x: java.lang.String = Hello, World

scala> val r1 = x.reverse
r1: String = dlroW ,olleH

scala> val r2 = x.reverse
r2: String = dlroW ,olleH
```

r1 and r2 are the same.

Suppose we replace all occurrences of x with the expression referenced by x.

```
scala> val r1 = "Hello, World".reverse
r1: String = dlroW ,olleH

scala> val r2 = "Hello, World".reverse
r2: String = dlroW ,olleH
```

r1 and r2 are still the same.

What's more, r1 and r2 are referentially transparent as well.


```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val y = x.append(", World")
y: java.lang.StringBuilder = Hello, World

scala> val r1 = y.toString
r1: java.lang.String = Hello, World

scala> val r2 = y.toString
r2: java.lang.String = Hello, World
```

r1 and r2 are the same.

Replacing all occurrences of y with the expression referenced by y:

```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val r1 = x.append(", World").toString
r1: java.lang.String = Hello, World

scala> val r2 = x.append(", World").toString
r2: java.lang.String = Hello, World, World
```

We conclude that **StringBuilder.append** is *not* a pure function.

r1 and r2 are no longer the same.

Non RT Example

For **buyCoffee** to be pure, it must be the case that

$$p(\texttt{buyCoffee(aliceCreditCard)}) == p(\texttt{Coffee()})$$

for *any p*.

```
class Cafe {
  def buyCoffee(cc: CreditCard): Coffee = {
    val cup = Coffee()
    cc.charge(cup.price) // return value is ignored here
    cup
  }
}
```


RT and Substitution Model

- Referential transparency forces the invariant that **everything a function *does* is represented by the *value* that it returns**, according to the result type of the function.
- This constraint enables a ***simple and natural mode of reasoning*** about program evaluation called the **substitution model**.

Benefits of Purity, Substitution Model

- **A pure function** is **modular**.
  - because it separates the logic of the computation itself from "*what to do with the result*" and "*how to obtain the input*"; it's a black box.
- Modular programs allows **local reasoning** and **independent reuse**.
  - the meaning of the whole depends only on the meaning of the components and the rules governing their composition
- Hence, pure functions are **composable**.

Benefits of FP

- More modular
- Easier to test
- Easier to reuse
- Easier to parallelize
- Easier to generalize
- Easier to reason about
- Less prone to bugs