Basics of Functional Programming

A Motivating Example: Cafe

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

Bad because card is charged as a side effect.

Mockable Payments

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

Better because we can now supply a mock `Payments` object, but

- mocking is tedious,
- function still has a side effect (does more than one thing), and
- hard to reuse `buyCoffee` – if we buy 2 coffees we're charged twice rather than once.

Georgia Tech

Functional Cafe

```scala
class FunctionalCafe {

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

Now separating concern of creating a charge from processing a charge

Composable Charges

```
class FunctionalCafe {

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

  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)))
  }
}
```

Georgia Tech

Composable Charges

By adding a combining operator to `Charge`:

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

we can easily compose multiple purchases into one:

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

Pure Functions

A **pure function** is simply a computational representation of a mathematical function.

In Scala, a function is represented by a type such as `A => B`. The function `f: A => B` is pure iff:

- `f` relates every value `a` in `A` to exactly one value `b` in `B`, and
- the computation of `b` is determined only by the value of `a`.

We also say that a pure funciton has no *side effects*, that is, no observable effects on the program's state.

Referential Transparency

We can operationalize the concept of function purity with referential transparency.

> *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.*

The substitution model of function evaluation relies on referential transparency.

Referential Transparency and Side Effects

Remember `buyCoffee`:

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

Since `buyCoffee` returns a `new Coffee()` then `p(buyCoffee(aliceCreditCard))` would have to be equivalent to `p(new Coffee())` for any `p`. But that's not the case, because `p(buyCoffee(aliceCreditCard))` also results in a state change to `aliceCreditCard`.

Referential Transparency and Mutable Data

```
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
```

Now replace `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
```

`r1` and `r2` no longer equal.

Referential Transparency and Immutable Data

```
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
r1: String = dlroW ,olleH
```

Now replace `x` with 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` still equal.

Closing Thoughts

Functional programming means programming with immutable data and pure functions. FP gives us:

- *composability*
  - the meaning of the whole depends only on the meaning of the components and the rules governing their composition
- *equational reasoning*
  - we can substitute values for the expressions that compute them, enabling local reasoning about expressions

Georgia Tech