Math in Programming

For non-math nerds :)

Math and Programming

- Are related!
- Imperative/procedural code obfuscates this relationship
- Result: Programmers are trained to not think mathematically

New Paradigm

- Functional programming models computations using mathematical functions
- Challenge: Come up with mathematical model for programming
- Category theory to the rescue!

Category Theory Wins

- Provides mathematical model for programming
- Model contains abstractions for common programming challenges
- Mathematical laws eliminate certain categories of bugs
- This presentation focuses on the model

Let's Talk About

- Functions
- Categories
- Functors
- Monads

PHD Not Required

- *This slide intentionally left blank. :)*

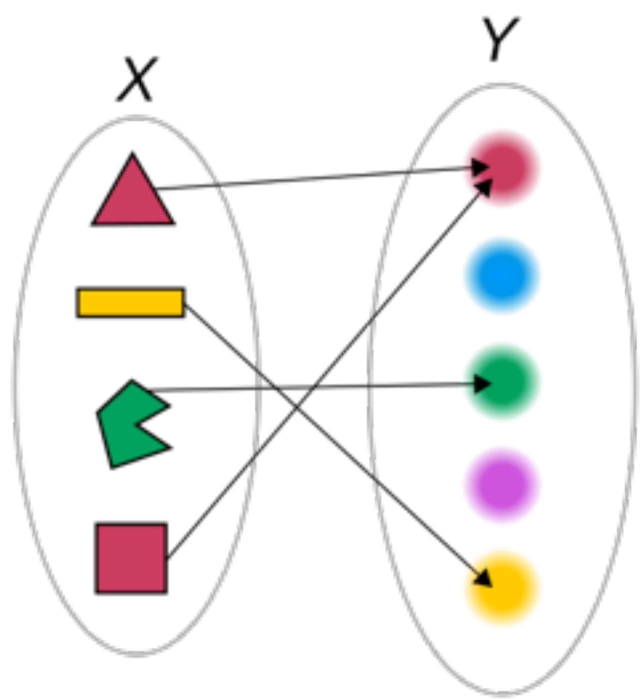
Functions

Functions

- Express a relationship between a set of inputs and outputs
- Can be modeled by maps
- The set of keys is called the domain
- The set of values is called the co-domain
- Restriction that each key points to a single value (map enforces this too)

Functions

- Functions can relate more than just numbers



Functions

- Here's a model of this function in Scala

```scala
abstract class Shape(color: Color)

case class Triangle(color: Color) extends Shape(color)
case class Rectangle(color: Color) extends Shape(color)
case class Hexagon(color: Color) extends Shape(color)
case class Square(color: Color) extends Shape(color)
```

```scala
val color-of-the-shape: Map[Shape, Color] = Map(
  redTriangle -> Red,
  yellowRectangle -> Yellow,
  greenHexagon -> Green,
  redSquare -> Red
)
```

```scala
val redTriangle = Triangle(Red)
val yellowRectangle = Rectangle(Yellow)
val greenHexagon = Hexagon(Green)
val redSquare = Square(Red)
```

```scala
trait Color

case object Red extends Color
case object Blue extends Color
case object Green extends Color
case object Purple extends Color
case object Yellow extends Color
```

Functions

- Functions can also be modeled by an abstract class

```
abstract class UnaryFunction[A, B] {
  def apply(a: A): B
}
```

```
class ColorOfTheShape extends UnaryFunction[Shape, Color] {
  override def apply(shape: Shape): Color = shape.color
}
```

Functions

- Can be composed
- Are pure (side effect free)
- Can be used as inputs...
- And outputs

Categories

Categories

- Categories are composed of objects and arrows
- Objects: Things in the category
- Arrows: Morphisms (functions) between things in the category

Categories

- Let's define a category in Scala
- Objects: All types in Scala
- Arrows: All functions in Scala
- This category contains others (as we'll see)

Functors

Functors

- Functors are functions that map between categories such that
- Every object in category A is mapped to a object in category B
- Every morphism in category A is mapped to a morphism in category B

Functors

- Example: List
- First: Map all objects (types) from category "All Scala Types & Functions" to category "List"
- This actually is a lot easier than you think

Functors

- To map the types, use a type constructor (generic method)

```
object List {
  def apply[A](a: A*): List[A] = a.foldLeft(List.empty[A]){(acc, a) => acc :+ a}
}
```

```
List(1,2,3,)          //Maps Int to List[Int]
List("a", "b", "c")  //Maps String to List[String]
List(List(1, 2), List(3, 4), List(5, 6)) //Maps List[Int] to List[List[Int]]
```

Functors

- Next: Map all morphisms (functions) from from category “All Scala Types & Functions” to category “List”
- Again, as we’ll see, not so scary

Functors

```scala
def map[A, B](xs: List[A], f: A => B): List[B] = xs match {
  case Nil => Nil
  case h :: t => f(h) :: map(t, f)
}

map(List(1,2,3), (x: Int) => x + 1)
map(List("a", "bc", "def"), (x: String) => x.length)
map(List(List(1, 2), List(3, 4), List(5,6)), (x: List[Int]) => x.last)
```

Functors

- Mapping objects handled by "constructor"
- Mapping morphisms handled by higher order function
- Let's see another example

Functors

- Example: Option
- First: Map all objects (types) from category "All Scala Types & Functions" to category "Option"

Functors

```
object Option {
  def apply[A](a: A): Option[A] = if(a == null) None else Some(a)
}

Option(1)             //Maps Int to Option[Int]
Option("abc")         //Maps String to Option[String]
Option(Option(12l)) //Maps Option[Long] to Option[Option[Long]]
```

Functors

- Next: Map all morphisms (functions) from from category “All Scala Types & Functions” to category “List”

Functors

```scala
def map[A, B](x: Option[A], f: A => B): Option[B] = x match {
  case Some(a) => Some(f(a))
  case None => None
}

map(Option(1), (x: Int) => x + 1)
map(Option("abc"), (x: String) => x.length)
map(Option(Option(2)), (x: Option[Int]) => map(x, (y: Int) => y + 1))
```

Monads

Monads

- Enhances Functors with extra goodies
- Consider the following morphism

```
def half(x: Int): Option[Int] = if(x % 2 == 0) Some(x / 2) else None

map(Option(10), half)   //yields Some(Some(5))
map(Option(11), half)   //yields Some(None)
```

Monads

- We've now buried our Option in another Option
- Monad to the rescue
- Adds third operation to the functor duo

Monads

- Enter bind (or flatMap in Scala land)
- If Map takes morphism A => B and “lifts” it to F[A] => F[B]
- FlatMap (bind) takes morphism A => F[B] and “lifts” it to F[A] => F[B]

Monads

```
def flatMap[A, B](x: Option[A], f: A => Option[B]): Option[B] = x match {
  case Some(a) => f(a)  //Note no functor use here
  case None => None
}

flatMap(Option(10), half)   //yields Some(5)
flatMap(Option(11), half)   //yields None
```

Conclusion

- Just what I know so far, lots more to learn
- No PHD required, as promised
- This info not 100% perfect; starting point for more learning
- Categories and Monads (even Functors) have some laws you should research

Questions

?