

Comp 311

Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

September 13, 2018

Case 1

**We Expect Few New Functions
But Many New Variants**

Abstract Methods

```
abstract class Shape {  
  def area: Double  
}
```

```
case class Circle(radius: Double) extends Shape {  
  val pi = 3.14  
  def area = pi * radius * radius  
}
```

```
case class Square(side: Double) extends Shape {  
  def area = side * side  
}
```

```
case class Rectangle(length: Double, width: Double)  
extends Shape {  
  def area = length * width  
}
```

Design Templates for Abstract Datatypes (Part 2)

Case Two

**We Expect Many New Functions
But Few New Variants**

One (Pattern Matching) Method to Rule Them All

```
abstract class Shape {  
  val pi = 3.14  
  def area: Double = this match {  
    case Circle(radius) => pi * radius * radius  
    case Square(side) => side * side  
    case Rectangle(width, height) => width * height  
  }  
}
```

Case 2: We Expect Many New Functions But Few New Variants

- This is a case that traditional functional programming handles well
- Classic example domains: Compilers, theorem provers, numeric algorithms, machine learning
- Declare a top-level function with cases for each data variant

a.k.a., The Visitor Pattern

We Can Define Arbitrary Functions Without Modifying Data Definitions

```
def makeLikeFirst(shape0: Shape, shape1: Shape) = {  
  (shape0, shape1) match {  
    case (Circle(r), Square(s)) => Circle(s)  
    case (Circle(r), Rectangle(l,w)) => Circle((l+w)/2)  
  
    case (Square(s), Circle(r)) => Square(r)  
    case (Square(s), Rectangle(l,w)) => Square((l+w)/2)  
  
    case (Rectangle(l,w), Circle(r)) => Rectangle(r,r)  
    case (Rectangle(l,w), Square(s)) => Rectangle(s,s)  
  
    case _ => shape1  
  }  
}
```


But A New Data Variant Requires Us To Modify All Functions Over the Datatype

```
val pi = 3.14

def area(shape: Shape) = {
  shape match {
    case Circle(r) => pi * r * r
    case Square(x) => x * x
    case Rectangle(x,y) => x * y
    case Triangle(b,h) => b*h/2
  }
}
```

But A New Data Variant Requires Us To Modify All Functions Over the Datatype

```
def makeLikeFirst(shape0: Shape, shape1: Shape) = {
  (shape0, shape1) match {
    case (Circle(r), Square(s)) => Circle(s)
    case (Circle(r), Rectangle(l,w)) => Circle((l+w)/2)
    case (Circle(r), Triangle(b,h)) => Circle(b)

    case (Square(s), Circle(r)) => Square(r)
    case (Square(s), Rectangle(l,w)) => Square((l+w)/2)
    case (Square(s), Triangle(b,h)) => Square(b+h/2)

    case (Rectangle(l,w), Circle(r)) => Rectangle(r,r)
    case (Rectangle(l,w), Square(s)) => Rectangle(s,s)
    case (Rectangle(l,w), Triangle(b,h)) => Rectangle(b,h)

    // plus all the cases for Triangle on the left (omitted)
    case _ => shape1
  }
}
```

Sealed Data Types

- Adding the **sealed** keyword to an abstract type indicates that all subclasses of that type are declared in the current compilation unit.
- Provides extra information to the compiler for optimizations and diagnostics

```
sealed abstract class Shape
case class Square(length: Double) extends Shape
case class Circle(radius: Double) extends Shape
case class Triangle(base: Double, height: Double)
      extends Shape
```

Sealed Data Types

```
object Math {  
  val pi = 3.141592653589793  
}
```

```
sealed abstract class Shape {  
  def area: Double = this match {  
    // case Square(x) => x * x  
    case Circle(r) => Math.pi * r * r  
    case Triangle(b, h) => 0.5 * b * h  
  }  
}
```

warning: match may not be exhaustive.
It would fail on the following input: Square(_)
def area: Double = this match {

Recursively Defined Datatypes

Recursively Defined Datatypes

- Case classes allow us to combine multiple pieces of a data into a single object
- But sometimes we don't know how many things we wish to combine
- We can use recursion to define datatypes of unbounded size
- This case corresponds to the Composite Design Pattern

Backus-Naur Form For Lists of Ints

```
List ::= Empty  
      | Cons(Int, List)
```

Examples of Lists

Empty

Cons(3, Empty)

Cons(3, Cons(1, Empty))

Cons(3, Cons(1, Cons(4, Empty)))

Defining Lists With Scala Case Classes

```
abstract class List
case object Empty extends List
case class Cons(head: Int, tail: List) extends List
```

Where Do We Put Functions Over Lists?

- We do not expect to define new subtypes of lists
- We do expect to define many new functions over lists
- Similar to our Case Two Design Template for Abstract Datatypes
- Thus, we will start with our pattern matching template

An Example Function for Lists

```
def containsZero(xs: List): Boolean = {  
  xs match {  
    case Empty => false  
    case Cons(n, ys) => {  
      if (n == 0) true  
      else containsZero(ys)  
    }  
  }  
}
```

An Example Function for Lists

```
def containsZero(xs: List): Boolean = {  
  xs match {  
    case Empty => false  
    case Cons(n, ys) => (n == 0) || containsZero(ys)  
  }  
}
```

Generalizing to Our First Template Function for Lists

```
def ourFunction(xs: List): Boolean = {  
  xs match {  
    case Empty => ...  
    case Cons(n, ys) => ... n ... ourFunction(ys) ...  
  }  
}
```


Generalizing to Our First Template Function for Lists

```
def ourFunction(xs: List): Boolean = {  
  xs match {  
    case Empty => ...  
    case Cons(n, ys) => ... n ... ourFunction(ys) ...  
  }  
}
```

We need to determine our *base case*

Generalizing to Our First Template Function for Lists

```
def ourFunction(xs: List): Boolean = {  
  xs match {  
    case Empty => ...  
    case Cons(n, ys) => ... n ... ourFunction(ys) ...  
  }  
}
```



We must determine how to combine these values

Generalizing to Our First Template Function for Lists

```
def ourFunction(xs: List): Boolean = {  
  xs match {  
    case Empty => ...  
    case Cons(n, ys) => ... n ... ourFunction(ys) ...  
  }  
}
```

This template is an example of *natural recursion* or *structural recursion*: We recursively decompose and then recombine a computation according to the natural structure of the data.


Filling in the Template

```
def containsZero(xs: List): Boolean = {  
  xs match {  
    case Empty => false  
    case Cons(n, ys) => (n == 0) || containsZero(ys)  
  }  
}
```

Here the base case is easy:
An empty list does not contain zero
(or anything else)

Filling in the Template

```
def containsZero(xs: List): Boolean = {  
  xs match {  
    case Empty => false  
    case Cons(n, ys) => (n == 0) || containsZero(ys)  
  }  
}
```



We break into cases based on the pieces from match: Either our first element n is zero or the answer lies with the rest of the list

Another Example: How Many Elements?

```
def length(xs: List): Int = {  
  xs match {  
    case Empty => 0  
    case Cons(n, ys) => 1 + length(ys)  
  }  
}
```

Another Example: The Sum of the Elements

```
def sum(xs: List): Int = {  
  xs match {  
    case Empty => 0  
    case Cons(n, ys) => n + sum(ys)  
  }  
}
```

Another Example: The Product of the Elements

```
def product(xs: List): Int = {  
  xs match {  
    case Empty => 1  
    case Cons(n, ys) => n * product(ys)  
  }  
}
```

Converting Hours to Seconds

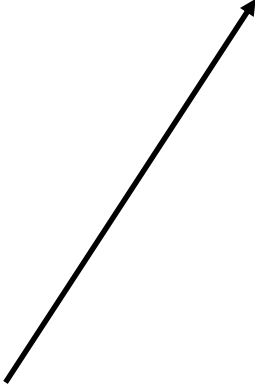
Problem Statement: Given a list of times measured in hours, we want to construct a list of corresponding times measured in seconds

Converting Hours to Seconds

```
def hoursToSeconds(xs: List): List = {  
  xs match {  
    case Empty => Empty  
    case Cons(n, ys) => Cons(seconds(n), hoursToSeconds(ys))  
  }  
}  
  
def seconds(hours: Int) = 3600 * hours
```

Generalizing to a Template

```
def ourFunction(xs: List): List = {  
  xs match {  
    case Empty => ...  
    case Cons(n, ys) => Cons(...n...,  
                             ourFunction(ys))  
  }  
}
```



Really, this is the same template as before, but now Cons is our combining operation

The Natural Numbers

```
Nat ::= 0
      | Next(Nat)
```

The Natural Numbers

```
Nat ::= 0
      | Next(Nat)
```

Here we are between Cases One and Two for Abstract Datatypes:

- No new variants expected
- Many new functions expected
- But some basic functions are intrinsic to the type

Defining The Natural Numbers in Scala

```
abstract class Nat  
case object Zero extends Nat  
case class Next(n: Nat) extends Nat
```

Defining The Natural Numbers in Scala

```
abstract class Nat {  
  def +(n: Nat): Nat  
  def *(n: Nat): Nat  
}
```

Defining The Natural Numbers in Scala

```
case object Zero extends Nat {  
  def +(n: Nat) = n  
  def *(n: Nat) = Zero  
}
```

```
case class Next(n: Nat) extends Nat {  
  def +(m: Nat) = Next(n + m)  
  def *(m: Nat) = m + (n * m)  
}
```

Defining The Natural Numbers in Scala

```
case object Zero extends Nat {  
  def +(n: Nat) = n  
  def *(n: Nat) = Zero  
}
```

Again we have natural
recursion: base case,
recursion, combination

```
case class Next(n: Nat) extends Nat {  
  def +(m: Nat) = Next(n + m)  
  def *(m: Nat) = m + (n * m)  
}
```

Example Reduction

(3 + 2)

Next (Next (Next (Zero)) + Next (Next (Zero))) \mapsto
Next (Next (Next (Zero)) + Next (Next (Zero))) \mapsto
Next (Next (Next (Zero) + Next (Next (Zero)))) \mapsto
Next (Next (Next (Zero + Next (Next (Zero))))) \mapsto
Next (Next (Next (Next (Next (Zero)))))

Factorial

```
def factorial(n: Nat): Nat = {  
  n match {  
    case Zero => Next(Zero)  
    case Next(m) => n * factorial(m)  
  }  
}
```


Transferring The Pattern To Ints

```
def factorial(n: Int): Int = {  
  require (n >= 0)  
  
  if (n == 0) 1  
  else n * factorial(n - 1)  
  
} ensuring (_ > 0)
```

Combining Via Auxiliary Functions

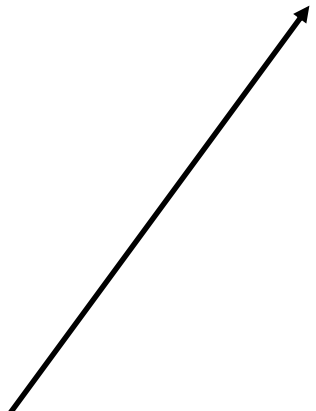
Combining Via Auxiliary Functions

- As our examples with natural numbers shows, it is often desirable to define the *combining operation* of a natural recursion as an auxiliary function
- We can apply this insight to lists and use our template to cover yet more cases

Sorting Lists

```
def sort(xs: List): List = {  
  xs match {  
    case Empty => Empty  
    case Cons(n, ys) => insert(n, sort(ys))  
  }  
}
```

We need to explain how to
insert into a sorted list



Insertion

```
def insert(n: Int, xs: List): List = {  
  xs match {  
    case Empty => Cons(n, Empty)  
    case Cons(m, ys) => {  
      if (n <= m) Cons(n, xs)  
      else Cons(m, insert(n, ys))  
    }  
  }  
}
```

Insertion

```
def insert(n: Int, xs: List): List = {  
  xs match {  
    case Empty => Cons(n, Empty)  
    case Cons(m, ys) => {  
      if (n <= m) Cons(n, xs)  
      else Cons(m, insert(n, ys))  
    }  
  }  
}
```

This parameter is not traversed,
but is used for combination and comparison
Other functions follow this pattern.

Appending Two Lists

```
abstract class List {  
  /**  
   * Returns a new list with the elements of  
   * this list appended to the given list.  
   */  
  def ++(ys: List): List  
}
```

Appending Two Lists

```
case object Empty extends List {  
  def ++(ys: List) = ys  
}
```


Appending Two Lists

```
case class Cons(first: Int, rest: List) extends List {  
  def ++(ys: List) = Cons(first, rest ++ ys)  
}
```