

Comp 311

Functional Programming

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

October 18, 2018

Announcement

Midterm exam is on **Tuesday** (Oct 23)
in DH 1075 (this room) from 7pm – 10pm

Call-By-Value
and
Call-By-Name

Call-By-Value

- Thus far, the evaluation semantics we have studied (both with the substitution and environment models) is known as call-by-value:
- To evaluate a function application, we first evaluate the arguments and then evaluate the function body

Call-By-Value

- We have seen several “special forms” where this evaluation semantics is not what we want:

&& || if-else

Call-By-Value

- We could delay evaluation in these cases by wrapping arguments in function literals that take no parameters

```
def myOr(left: Boolean, right: () => Boolean) =  
  if (left) true  
  else right()
```

Call-By-Value

- We could delay evaluation in these cases by wrapping arguments in function literals that take no parameters

```
myOr(true, () => 1/0 == 2) ↪ true
```

- Functions that take no arguments are referred to as *thunks*

Call-By-Name

- Scala provides a way that we can pass arguments as thunks without having to wrap them explicitly

```
def myOr(left: Boolean, right: => Boolean) =  
  if (left) true  
  else right
```

*We simply leave off the parentheses
in the parameter's type*



Call-By-Name

- Now we can call our function without wrapping the second argument in an explicit thunk:

```
myOr(true, 1/0 == 2) ↦ true
```

- The thunk is applied (to nothing) the first time that the argument is evaluated in a function

Call-By-Name

- We can use by-name parameters to define new *control abstractions*:

```
def myAssert(predicate: => Boolean) =  
  if (assertionsEnabled && !predicate)  
    throw new AssertionError
```

Syntactic Sugar: Braces for Passing Arguments

- Any function that takes a single argument can be applied by passing the argument enclosed in braces instead of parentheses

```
myAssert {  
    2 + 2 == 4  
}
```

Syntactic Sugar: Braces for Passing Arguments

- Any function that takes a single argument can be applied by passing the argument enclosed in braces instead of parentheses

```
myAssert {  
  def double(n: Int) = 2 * n  
  double(2) == 4  
}
```

Scala Immutable Collections

Immutable Lists

- Behave much like the lists we have defined in class
- Lists are covariant
- The empty list is written `Nil`
- `Nil` extends `List[Nothing]`

Immutable Lists

- The list constructor takes a variable number of arguments:

```
List(1,2,3,4,5,6)
```

Immutable Lists

- Non-empty lists are built from Nil and Cons (written as the right-associative operator ::)

`1 :: 2 :: 3 :: 4 :: Nil`

List Operations

- `head` returns the first element
- `tail` returns a list of elements but the first
- `isEmpty` returns true if the list is empty
- Many of the methods we have defined are available on the built-in lists

FoldLeft and FoldRight Written as Operators

- foldLeft:

`(zero /: xs) (op)`

- foldRight:

`(xs :\ zero) (op)`

FoldLeft and FoldRight Written as Operators

- foldLeft:

```
(xs foldLeft zero) (op)
```

- foldRight:

```
(xs foldRight zero) (op)
```

FoldLeft and FoldRight Written as Methods

- foldLeft:

```
xs.foldLeft(zero) { op }
```

- foldRight:

```
xs.foldRight(zero) { op }
```

SortWith

```
List(1,2,3,4,5,6) sortWith (_ > _)
```

↳

```
List(6, 5, 4, 3, 2, 1)
```

Range

`List.range(1, 5)`

↳

`List(1, 2, 3, 4)`

Using Fill for Uniform Lists

```
List.fill(10)(0) ⇨  
List(0,0,0,0,0,0,0,0,0,0)
```

Using Fill for Uniform Lists

```
List.fill(3,3)(0) ⇨
```

```
List(List(0,0,0),  
      List(0,0,0),  
      List(0,0,0))
```


Tabulating Lists

```
List.tabulate(3,3) { (m,n) =>  
  if (m == n) 1 else 0  
}
```

↳

```
List(List(1,0,0),  
      List(0,1,0),  
      List(0,0,1))
```

Immutable Sets

Immutable Sets

- Sets are unordered, unrepeated collections of elements
- `Set[T]` extends the function type `T ⇒ Boolean`
- Sets are parametric and *invariant* in their element type

Why *in*-variant?



Set Factory

Set(1,2,3,4,5)

Set Element Addition

$\text{Set}(1, 2, 3) + 4 \mapsto$
 $\text{Set}(1, 2, 3, 4)$

Set Element Subtraction

$$\text{Set}(1, 2, 3) - 2 \mapsto \text{Set}(1, 3)$$

$$\text{Set}(1, 2, 3) - 4 \mapsto \text{Set}(1, 2, 3)$$

Set Intersection

Set(1,2,3) intersect Set(2,4,5,3) \mapsto
Set(2,3)

Set(1,2,3) & Set(2,4,5,3) \mapsto
Set(2,3)

Set Union

`Set(1,2,3) union Set(2,4,5) ↪
Set(1,2,3,4,5)`

`Set(1,2,3) | Set(2,4,5) ↪
Set(1,2,3,4,5)`

`Set(1,2,3) ++ Set(2,4,5) ↪
Set(1,2,3,4,5)`

Set Difference

Set(1,2,3) diff Set(2,4,5,3) \mapsto
Set(1)

Set(1,2,3) - - Set(2,4,5,3) \mapsto
Set(1)

Set Cardinality

`Set(1,2,3).size` \mapsto
3

Set Membership

`Set(1,2,3).contains(2) ⇨
true`

`Set(1,2,3)(2) ⇨
true`

*The `apply` method on sets is
equivalent to the `contains` method.*

Immutable Maps

Immutable Maps

- Maps are collections of key/value pairs
- They are parametric in both the key and value type
 - Covariant in their value type
 - Invariant in their key type

← Why *in*-variant?

The -> Operator

- The infix operator -> returns a pair of its arguments:

$$\begin{array}{c} 1 \ -> \ 2 \\ \mapsto \\ (1, 2) \end{array}$$

- Note: Scala also allows *Unicode Operators*, and the infix “→” operator is one such example:

$$\begin{array}{c} 1 \ \rightarrow \ 2 \\ \mapsto \\ (1, 2) \end{array}$$

The \rightarrow Operator is Left Associative

> 1 \rightarrow 2 \rightarrow 3 \rightarrow 4

res8: ((Int, Int), Int), Int) = (((1,2),3),4)

The Map Factory

Map("a" → 1, "b" → 2, "c" → 3)

↪

Map(a -> 1, b -> 2, c -> 3)

Map Addition

Map("a" → 1, "b" → 2, "c" → 3) + ("d" → 4)
↳
Map(a -> 1, b -> 2, c -> 3, d -> 4)

Map Operations

The operators/methods are defined in the expected way:

- -
- ++
- --
- size

Map Membership

```
Map("a" → 1, "b" → 2, "c" → 3).contains("b")  
    ↪  
true
```

Map Lookup

Map("a" → 1, "b" → 2, "c" → 3)("c")
↳
3

Map("a" → 1, "b" → 2, "c" → 3).get("c")
↳
Some(3)

Map Keys

```
Map("a" → 1, "b" → 2, "c" → 3).keys  
  ↪  
Set(a, b, c): Iterable[String]
```

```
Map("a" → 1, "b" → 2, "c" → 3).keySet  
  ↪  
Set(a, b, c): Set[String]
```

Map Values

Map("a" → 1, "b" → 2, "c" → 3).values
↳
Set(1,2,3)

Map Empty

```
Map("a" → 1, "b" → 2, "c" → 3).isEmpty  
  ↪  
false
```