

Chapter 2: The Functional Approach to Collections

Sergei Winitzki

Academy by the Bay

November 12, 2017

Tuples

- Pair of values:

```
val a: (Int, String) = (123, "xyz")
```

- Triple of values:

```
val b: (Boolean, Int, Int) = (true, 3, 4)
```

- Tuples can be nested:

```
val c: (Boolean, (String, Int), Boolean) =  
  (true, ("abc", 3), false)
```

- Parts of the tuple can be accessed by number:

```
val x: (String, Int) = c._2
```

- Functions on tuples:

```
def f(p: (Boolean, Int), q: Int): Boolean = p._1 && (p._2 > q)
```

Pattern-matching syntax for tuples

Scala allows pattern matching in two places:

- `val pattern = ...` (value assignment)
- `case pattern ⇒ ...` (partial function)

Examples:

- `val a = (1, 2, 3); val (x, y, z) = a`
- `val f: ((Int, Int, Int)) ⇒ Int = { case (x, y, z) ⇒ x + y + z }; f(a)`

Combining tuple types with other types

We can use tuple types anywhere:

- Tuple of functions:

```
val q: (Int ⇒ Int, Int ⇒ Int) = (x ⇒ x + 1, x ⇒ x - 1)
```

- Sequence of tuples:

```
val s: Seq[(String, Int)] =  
  Seq(("apples", 3), ("oranges", 2), ("pears", 0))
```

- Tuples are used a lot in the Scala standard library...

- ▶ `zip: (Seq[A], Seq[B]) ⇒ Seq[(A, B)]`

- ▶ `map: (Map[K, V], (K, V) ⇒ R) ⇒ Seq[R]`

- ★ Note: the syntax `(a → b)` means the same as the tuple `(a, b)`

- ▶ `toMap: Seq[(K, V)] ⇒ Map[K, V]`

Worked examples

- 1 for a given sequence a_i , compute the sequence of pairs $b_i = (\cos a_i, \sin a_i)$ – use `.map`, assume `Seq[Double]`
- 2 in a given sequence a_i , count how many times $\cos a_i > \sin a_i$ occurs – use `.count`, assume `Seq[Double]`
- 3 for given sequences a_i and b_i , compute the sequence of differences $c_i = a_i - b_i$ (use `.zip`, `.map`, assume `Seq[Double]`)
- 4 in a given sequence a_i , count how many times $a_i > a_{i+1}$ occurs
- 5 for a given $k > 0$, compute the sequence $b_i = \max(a_{i-k}, \dots, a_{i+k})$ – use `.sliding`
- 6 create a multiplication table as a value of type `Map[(Int, Int), Int]` – use `.flatMap`
- 7 for a given sequence a_i , compute the combined set of the numbers a_i , $\cos a_i$, $\sin a_i$ and find its maximum value – use `.map`, `.flatMap`, `.max`
- 8 from a `Map[String, String]` mapping names to addresses, and assuming that the addresses do not repeat, compute a `Map[String, String]` mapping addresses to names – use `.toMap`, `.map`
 - Write this as a function with type parameters `Name` and `Address` instead of the fixed type `String`

Exercises I

- ❶ Find all i, j within $(0, 1, \dots, 9)$ such that $i + 4 * j > i * j$ (use `.flatMap`)
 - ▶ Same task for i, j, k and the condition $i + 4 * j + 9 * k > i * j * k$
- ❷ Given two sequences `a: Seq[String]` and `b: Seq[Boolean]` of equal length, compute a `Seq[String]` with those elements of `a` for which the corresponding element of `b` is `true` – use `.zip`, `.map`, `.filter`
- ❸ Convert a `Seq[Int]` into a `Seq[(Int, Boolean)]` where the Boolean value is `true` when the element is followed by a larger value; e.g. `Seq(1,3,2,4)` is to be converted into `Seq((1,true), (3,false), (2,true))`
- ❹ Given `a: Seq[String]` and `b: Seq[Int]` of equal length, and assuming that elements of `b` do not repeat, compute a `Map[Int, String]` that maps numbers from `b` to their corresponding strings from `a`
 - ▶ Write this as a function with type parameters `S` and `I` instead of the fixed types `String` and `Int`
- ❺ Given `a: Seq[String]` and `b: Seq[Int]` of equal length, compute a `Seq[String]` that contains the strings from `a` ordered according to the corresponding numbers from `b` – use `.sortBy`
 - ▶ Write this as a function with type parameter `S` instead of `String`

Exercises II

- ① Given a `Seq[(String, Int)]` showing a list of purchased items (names may repeat), compute `Map[String, Int]` showing the total counts: e.g. given a `Seq(("apple", 2), ("pear", 3), ("apple", 5))`, compute `Map("apple" → 7, "pear" → 3)` – use `.groupBy`, `.map`, `.sum`
 - ▶ Write this as a function with type parameter `S` instead of `String`
- ② Given a `Seq[List[Int]]`, compute a `Seq[List[Int]]` where each new inner list contains the three largest elements from the initial inner list – use `.sortBy`, `.take`, `.map`
- ③ Given two sets `a`, `b` of type `Set[Int]`, compute a `Set[(Int, Int)]` representing the Cartesian product of the sets `a` and `b` – use `.flatMap`
 - ▶ Write this as a function with type parameters `I`, `J` instead of `Int`
- ④ * Given a `Seq[Map[Person, Amount]]`, showing the amounts various people paid on each day, compute a `Map[Person, Seq[Amount]]`, showing the sequence of payments for each person (assume `Person` and `Amount` are type parameters and use `.flatMap`, `.toSeq`, `.groupBy`)

Mathematical induction I

Computing a number from a sequence

Typical problem:

- Compute an integer value from the sequence of its decimal digits

```
def fromDigits(digits: Seq[Int]): Int = ???  
fromDigits(Seq(1, 3, 0, 0)) == 1300
```

Mathematical formulation uses *induction*

- base case: empty sequence: `fromDigits(Seq()) = 0`
- induction step: if `fromDigits` is already computed for a sequence *previous...*, how to compute it for a sequence with one more element:
`fromDigits(Seq(previous..., x)) = 10 * fromDigits(previous...) + x`
 - ▶ the result still needs to be divided by 10

Translating mathematical induction into code:

- use recursion
- use standard library functions `fold`, `scan`, etc.

Mathematical induction II

Writing a recursive function by hand

- base case vs. inductive step needs to be decided in the code
- the function calls itself recursively

```
def fromDigits(digits: Seq[Int]): Int = {  
  digits match {  
    case Seq() => 0  
    case _ =>  
      val x = digits.head  
      val rest = digits.tail  
      10 * fromDigits(rest) + x  
  }  
}
```

- lots of code...
 - ▶ not very different from writing a loop

Tail recursion

The “accumulator” technique

The code of `fromDigits` calls itself *in the middle* of an expression

```
def fromDigits(...) = if (...) 0
else f(..., fromDigits(...), ...)
```

- the intermediate expression grows, causing stack overflow
- to remedy this: use **tail recursion**

```
@tailrec def fromDigits(...) = if (...) 0 else fromDigits(...)
@tailrec def fromDigits(...) = if (...) 0 else {
  val x = ...
  fromDigits(... x ...)
}
```

- the “accumulator technique” makes *some* programs tail-recursive
 - ▶ add another argument that accumulates the correct result

Mathematical induction III

Computing a number from a sequence

The standard function `foldLeft` implements general induction:

- base case is the first argument to `foldLeft`
- induction step is represented by a function $(\text{previous}, x) \Rightarrow \text{next}$

```
def fromDigits(digits: Seq[Int]): Int =  
  digits.foldLeft(0){ case (prev, x)  $\Rightarrow$  prev * 10 + x }
```

- see other library functions: `.foldRight`, `.fold`, `.reduce`

Mathematical induction IV

Computing a sequence from a number (iterate)

Typical problem:

- Compute the sequence of decimal digits of a given integer
 - ▶ we cannot solve this with `.map`, `.zip`, `.fold` etc., because the length of the resulting sequence is unknown
 - ▶ we need to “unfold” into a sequence of unknown length, and terminate it when some condition holds
- Inductive definition: given $n > 0$, build sequence (m_k, d_k) until $(0, 0)$:

$$(m_0, d_0) = (n, 0)$$

$$(m_k, d_k) = \left(\frac{m_{k-1}}{10}, (m_{k-1} \bmod 10) \right) \text{ for } k > 0$$

The `Iterator.iterate` method can do this:

```
Iterator.iterate((n, 0)) { case (m, _) => (m / 10, m % 10) }  
.takeWhile{case (m, d) => m > 0 || d > 0 }  
.drop(1).map(_._2) // extract sequence of digits
```

Mathematical induction V

Computing a sequence from another sequence (scan)

Typical problem:

- Compute partial sums of the given sequence: $b_k = \sum_{i=0}^k a_i$
- Definition by induction:

$$b_0 = 0;$$

$$b_k = a_k + b_{k-1} \text{ for } k > 0$$

- Example code:

```
val a = Seq(1, 2, 3, 4)
val b = a.scan(0) { (x, y) => x + y } // yields Seq(0, 1, 3, 6, 10)
```

Summary

What problems can we solve now?

- Compute mathematical expressions involving arbitrary recursion
- Create sequences from inductive definitions
- Process sequences using inductive definitions

Worked examples

- 1 Compute the smallest n such that $f(f(f(\dots f(1)\dots)) > 1000$, where the function f is applied n times (use `iterate`)
 - ▶ Write this as a function taking f , 1, and 1000 as arguments
- 2 Find the k -th largest element in an (unsorted) sequence of integers – use `foldLeft`
- 3 Find the last element of a (nonempty) sequence – use pattern matching, `.drop`, and tail recursion
- 4 Implement binary search over a sorted `Array[Int]` – use tail recursion

Exercises III

- 1 Compute the sum of squared digits of a given integer; e.g.,
`dsumsq(123) = 14`