

# Chapter 2: The Functional Approach to Collections

Sergei Winitzki

Academy by the Bay

November 22, 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 (esp. pairs) 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 pair `(a, b)`

- ▶ `toMap: Seq[(K, V)] ⇒ Map[K, V]`

- ▶ `toSeq: Map[K, V] ⇒ Seq[(K, V)]`

# Worked examples

- ❶ for a given sequence  $a_i$ , compute the sequence of pairs  $b_i = (\cos a_i, \sin a_i)$  – use `.map`, assume `a: Seq[Double]`
- ❷ in a given sequence  $a_i$ , count how many times  $\cos a_i > \sin a_i$  occurs – use `.count`, assume `a: Seq[Double]`
- ❸ for given sequences  $a_i$  and  $b_i$ , compute the sequence of differences  $c_i = a_i - b_i$  (use `.zip`, `.map`, assume `a, b: Seq[Double]`)
- ❹ in a given sequence  $a_i$ , count how many times  $a_i > a_{i+1}$  occurs
- ❺ for a given  $k > 0$ , compute the sequence  $b_i = \max(a_{i-k}, \dots, a_{i+k})$  – use `.sliding`
- ❻ create a multiplication table as a value of type `Map[(Int, Int), Int]` – use `.flatMap`
- ❼ 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`
- ❽ 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`; test it with `S=Boolean` and `I=Set[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; 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 =  
  if (digits.isEmpty) 0  
  else {  
    val x = digits.head  
    val rest = digits.drop(1)  
    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 expression overflow
  - ▶ this crashes our program with a “stack overflow” error
- To remedy this: use **tail recursion** (`fromDigits` is called at the “tail”)

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

- The “accumulator technique” makes *some* functions tail-recursive
  - ▶ add another argument that accumulates the final result

```
@tailrec def fromDigits(digits: Seq[Int], res: Int) =
  if (digits.isEmpty) res
  else fromDigits(digits.drop(1), 10 * res + digits.head)
```

# Mathematical induction III

## Computing a number from a sequence

The library 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) => prev * 10 + x }
```

- see other library functions: `.foldRight`, `.fold`, `.reduce`
- most of these functions are tail-recursive

# 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)
```

- `scanLeft` implements general induction:
  - ▶ base case is the first argument to `scanLeft`
  - ▶ induction step is represented by a function `(previous, x) ⇒ next`

# Summary

What problems can we solve now?

- Compute mathematical expressions involving arbitrary recursion
- Use tail recursion when possible
- Use arbitrary inductive (i.e. recursive) formulas to:
  - ▶ convert sequences to numbers (“aggregate”)
  - ▶ create new sequences from scratch
  - ▶ transform existing sequences

What problems are not solved with these tools?

- Compute non-tail recursive functions without expression overflow

# 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
- 5 For a given `n: Int`, compute the sequence  $(s_0, s_1, s_2, \dots)$  defined by  $s_0 = SD(n)$  and  $s_k = SD(s_{k-1})$  for  $k > 0$ , where  $SD(p)$  is the sum of the decimal digits of the integer  $p$ , e.g.  $SD(123) = 6$  (use `iterate`)
- 6 For a given sequence  $(s_0, s_1, s_2, \dots)$  of type `Iterator[T]`, compute the “half-speed” sequence  $(s_0, s_0, s_1, s_1, s_2, s_2, \dots)$  – use `.flatMap`
- 7 Cut off a given sequence  $(s_0, s_1, s_2, \dots)$  at a place  $k$  where an element  $s_k$  equals some earlier element  $s_i$  with  $i < k$  – use `.zip`, `.takeWhile`

# Exercises III

- 1 Compute the sum of squared digits of a given integer; e.g., `dsq(123)=14`
  - ▶ Same task for an arbitrary function `f: Int⇒Int` instead of squaring
- 2 For a given integer  $n$ , compute the sum of cubed digits, then the sum of cubed digits of the result, etc.; determine whether the resulting sequence starts repeating itself, and if so, whether it ever reaches 1
- 3 For a given integer  $n$ , compute the Collatz sequence:  $c_0 = n$  and

$$c_{k+1} = \begin{cases} c_k/2 & \text{if } c_k \text{ is even,} \\ 3c_k + 1 & \text{if } c_k \text{ is odd} \end{cases}$$

- ▶ Stop the sequence when it reaches 1
- 4 For `a,b,c` of type `Set[Int]`, compute the set of all sets of the form `Set(x,y,z)` where `x` is from `a`, `y` from `b`, and `z` from `c` (use `.flatMap`)
  - 5 \* Same task for a `Set[Set[Int]]` instead of just three sets `a,b,c` – use `.foldLeft`