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 $b_i = \max(a_{i-k}, ..., a_{i+k})$ – 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

1. Find all i, j within $(0, 1, ..., 9)$ such that $i + 4 * j > i * j$ (use `.flatMap`)
   - Same task for i, j, k and the condition $i + 4 * j + 9 * k > i * j * k$
2. 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`
3. 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))`
4. 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`
5. 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

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 `(previous, x) ⇒ next`

```
def fromDigits(digits: Seq[Int]): Int =
  digits.foldLeft(0){ case (prev, x) ⇒ 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

The `Iterator` type has methods `iterate`, `fill`, `tabulate` that can help

Mathematical induction V

Computing a sequence from a sequence (`scan`)

Typical problem:

- compute partial sums of the given sequence

Examples

- Using both `def` and `val`, define a function that...
  1. adds 20 to its integer argument
  2. takes an integer `x`, and returns a *function* that adds `x` to *its* argument
  3. takes an integer `x` and returns `true` iff `x + 1` is prime
- What are the types of the functions in Examples 1 - 3?
- Compute the average of all numbers in a sequence of type `Seq[Double]`. Use `sum` and `size` but no loops.
- Given n, compute the Wallis product truncated up to $\frac{2n}{2n+1}$:

$$\frac{2}{1}\frac{2}{3}\frac{4}{3}\frac{4}{5}\frac{6}{5}\frac{6}{7}\dots\frac{2n}{2n+1}.$$

  Use a sequence of `Int` or `Double` numbers, `map`, and `product`.
- Given `s: Seq[Set[Int]]`, compute the sequence containing the sets of size at least 3. Use `map`, `filter`, `size`. The result must be again of type `Seq[Set[Int]]`.

Summary

- What problems can we solve now?
  - Compute mathematical expressions involving sums, products, and quantifiers, based on integer ranges (such as $\sum_{k=1}^{n} f(k)$ etc.)
  - Implement functions that take or return other functions
  - Work on collections using `map` and other library methods
- What kinds of problems are not solved with these tools?
  - Compute the smallest n such that $f(f(f(...f(1)...) > 1000$, where the function f is applied n times.
  - Find the k-th largest element in an (unsorted) array of integers.
  - Perform binary search over a sorted array.
- Why can't we solve such problems yet?
  - Because we can't yet put *mathematical induction* into code

Exercises

1. Define a function of type `Seq[Double] => Seq[Double]` that "normalizes" the sequence: it finds the element having the max. absolute value and, if that value is nonzero, divides all elements by that factor.
2. Define a function of type `Seq[Seq[Int]] => Seq[Seq[Int]]` that adds 20 to every element of every inner sequence.
3. An integer n is called "3-factor" if it is divisible by only three different integers j such that $2 \leq j < n$. Compute the set of all "3-factor" integers n among $n \in [1, ..., 1000]$.
4. Given a function f of type `Int => Boolean`, an integer n is called "3-f" if there are only three different integers $j \in [1, ..., n]$ such that $f(j)$ returns `true`. Define a function that takes f as an argument and returns a sequence of all "3-f" integers among $n \in [1, ..., 1000]$. What is the type of that function? Rewrite Exercise 3 using that function.
5. Define a function that takes two functions `f: Int => Double` and `g: Double => String` as arguments, and returns a new function that computes the functional composition of `f` and `g`.