Transforming Collections

Effective Programming in Scala

Transforming Collections

This section is about transforming collections. We'll learn about a handful of methods that allow us to transform the elements of a collection to produce a new collection or, in the case of `foldLeft`, anything at all.

Transforming: Map

Using `map` we can transform every element into a new element. The new element can have the same or different type.

```
val list = List(1, 2, 3, 4)
list.map(x => x + 1) // List(2, 3, 4, 5)

val buff = mutable.ArrayBuffer(1, 2, 3, 4)
buff.map(x => x % 2 == 0) // mutable.ArrayBuffer(false, true, false, true)

val map = Map(0 -> "No", 1 -> "Yes")
map.map((key, value) => key -> (value * 2) ) // Map(0 -> NoNo, 1 -> YesYes)
```

Transforming: Map

Using `map` we can transform every element into a new element. The new element can have the same or different type.

```
val list = List(1, 2, 3, 4)
list.map(x => x + 1) // List(2, 3, 4, 5)

val buff = mutable.ArrayBuffer(1, 2, 3, 4)
buff.map(x => x % 2 == 0) // mutable.ArrayBuffer(false, true, false, true)

val map = Map(0 -> "No", 1 -> "Yes")
map.map((key, value) => key -> (value * 2) ) // Map(0 -> NoNo, 1 -> YesYes)
```

Note: after calling `map` the number of elements is unchanged. For sequences the order is also unchanged.

Transforming: FlatMap

We can also transform a collection with `flatMap`, which allows us to change the number of elements in the collection.

```
// Remove all the elements
List(1, 2, 3).flatMap(x => List())

// Double the number of elements
mutable.ArrayBuffer(1, 2).flatMap(x => mutable.ArrayBuffer(x, x * 2))

// Keep number of elements the same but change type
Map(0 -> "zero", 1 -> "one").flatMap((key, _) =>
  Map(key.toString -> key)
)
```

map vs flatMap

Implement a program that returns all the phone numbers of a contact list.

```
case class Contact(name: String, phoneNumbers: List[String])
val contacts: List[Contact] = ...
```

First attempt:

```
val allPhoneNumbers = contacts.map(contact => contact.phoneNumbers)
// allPhoneNumbers = List(List(), List("+41787829420"))
```

map vs flatMap

Implement a program that returns all the phone numbers of a contact list.

```
case class Contact(name: String, phoneNumbers: List[String])
val contacts: List[Contact] = ...
```

First attempt:

```
val allPhoneNumbers = contacts.map(contact => contact.phoneNumbers)
// allPhoneNumbers = List(List(), List("+41787829420"))
```

We get back a `List[List[String]]`, but ideally we would like to flatten the nested lists at the top-level. We can do so with the `flatMap` operation:

```
val allPhoneNumbers = contacts.flatMap(contact => contact.phoneNumbers)
// allPhoneNumbers = List("+41787829420")
```

Transforming: FoldLeft

The `foldLeft` method allows us to transform a collection into *anything* else. Let's see some examples.

```
// Sum the elements of the list (in practice we would use the sum method)
List(1, 2, 3).foldLeft(0)((accum, elt) => accum + elt) // 6

// Reverse the list (in practice we would use the reverse method)
List(1, 2, 3).foldLeft(List.empty[Int])((accum, elt) =>
  elt +: accum
)

// True if the last element is even or the list is empty
List(1, 2, 3).foldLeft(true)((accum, elt) => elt % 2 == 0)
```

Aside: Multiple Parameters Lists

The method `foldLeft` has two parameter lists.

Generally, methods can have more than one parameter list.

```
def foo(x: Int, y: Int)(f: Int => Int): Int
```

We call it as follows:

```
foo(0, 10)(i => i * i)
```

Or, with braces instead of parentheses:

```
foo(0, 10) { i =>
  i * i
}
```

Understanding FoldLeft: 1

We're going to look at two ways to understand `foldLeft`. Let's start with this example:

```
// Sum the elements of the list
List(1, 2, 3).foldLeft(0)((accum, elt) => accum + elt) // 6
```

We can think of the first argument, `0`, as an accumulator. The accumulator is the value we've computed over the portion of the list we've seen so far. The initial value of the accumulator is `0`, meaning the sum of the empty list is `0`.

The second argument, the function, is how we combine the accumulator with the current element of the list to create a new accumulator. In this example we add the element to the accumulator. Since the accumulator is the sum so far, the result will be the sum of all the elements.

Understanding FoldLeft: 2

Here's our second way of understanding `foldLeft`. We will see that `foldLeft` processes the list is the reverse order to which it is constructed, and we use the initial value of the accumulator at the start insteed of `Nil` at the end.

Let's start with

```
val data = List(1, 2, 3)
```

Remeber that we can write this list as

```
val data = 1 :: 2 :: 3 :: Nil
```

Lists are constructed from right to left. Brackets make this explicit.

```
val data = 1 :: (2 :: (3 :: Nil))
```

Understanding FoldLeft: 2

`foldLeft` processes the list from left to right, which is opposite to the construction order. Let's define a function `add`.

```
val add = (x: Int, y: Int) => x + y
```

Then

```
(1 :: 2 :: 3 :: Nil).foldLeft(0)(add)
```

is equivalent to

```
add(add(add(0, 1), 2), 3)
```

Notice that `foldLeft` goes from left-to-right, and instead of the list finishing with `Nil`, `foldLeft` starts with the initial accumulator `0`.

Understanding FoldLeft: 2

We can extend this way of understanding `foldLeft` to the general case:

```
xs.foldLeft(z)(f) == f(f(f(f(z, xs(0)), xs(1)), ...), xs(n - 1))
```

In words it says we start at the left of the list, combining the initial element with the accumulator, and then proceed from left to right combining partial results with the next element we encounter.

Understanding FoldLeft: Beyond Lists

The explanations above both used `List`. We can generalize to any collection by assuming some (possibly arbitrary) ordering on the elements. For example, we could write

```
mutable.ArrayBuffer(1, 2, 3)
```

as

```
1 +: 2 +: 3 +: mutable.ArrayBuffer.empty[Int]
```

and then the second explanation of `foldLeft` can be transferred to `ArrayBuffer`.

Transforming: GroupBy

`groupBy` groups the elements of a collection according to a partition function. For instance, to group email addresses by domains:

```
val emails = List("alice@sca.la", "bob@sca.la", "carol@earth.world")

// Function extracting the domain name from an email address
val domain: String => String =
  email => email.dropWhile(c => c != '@').drop(1)

val emailsByDomain = emails.groupBy(domain)
// emailsByDomain: Map[String, List[String]] = Map(
//   "sca.la"      -> List("alice@sca.la", "bob@sca.la"),
//   "earth.world" -> List("carol@earth.world")
// )
```

Transforming: Summary

We have seen the following methods for transforming collections:

- `map` for transforming the elements while keeping the general structure of the collection;
- `flatMap` for transforming elements and possibly transforming some part of the structure of the collection; and
- `foldLeft` for transforming a collection in any way; and
- `groupBy` for grouping elements by a partition function.