Collections Extra

Effective Programming in Scala

Additional Material on Collections

These optional slides contain additional material on collections. They follow the same structure, covering ways to

- construct;
- query; and
- transform

collections.

We then briefly discuss a few other collection types.

Constructing: Concatenating Collections

If we have two collections we can construct a new collection by concatenating them using the ++ method.

```
List(1, 2, 3) ++ List(4, 5, 6)
mutable.ArrayBuffer("a", "b") ++ mutable.ArrayBuffer("c", "d")
Map("a" -> true, "e" -> true) ++ Map("i" -> true)
```

Constructing: Concatenating Collections

If we have two collections we can construct a new collection by concatenating them using the ++ method.

```
List(1, 2, 3) ++ List(4, 5, 6)
mutable.ArrayBuffer("a", "b") ++ mutable.ArrayBuffer("c", "d")
Map("a" -> true, "e" -> true) ++ Map("i" -> true)
```

As with all methods in Scala we can also call ++ using dot-notation.

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

Aside: Concatenation is Immutable

When we concatenate two collections the original collections are always unchanged.

```
val first = mutable.ArrayBuffer("a", "b")
val second = mutable.ArrayBuffer("c", "d")

first ++ second // ArrayBuffer(a, b, c, d)

first // ArrayBuffer(a, b)
second // ArrayBuffer(c, d)
```

This is a general principle in the collection library: any method that works on both immutable and mutable collections will work the same way on both and will not modify the original collections.

Constructing: Mutable Concatenation

`ArrayBuffer` and other mutable sequences have a method `++=` that performs a mutable concatenation. The original sequence—the one on which we call the `++=` method—is changed after calling this method. Note the second sequence—the parameter we pass to `++=`—remains unchanged.

```
val first = mutable.ArrayBuffer(1, 2)
val second = mutable.ArrayBuffer(3, 4)

first ++= second

first // ArrayBuffer(1, 2, 3, 4)
second // ArrayBuffer(3, 4)
```

Constructing: Mutable Prepend and Append

Mutable sequences, such as `ArrayBuffer`, have additional methods that prepend and append in a way that modifies the original collection.

```
val data = mutable.ArrayBuffer(1, 2, 3)
data += 4
0 +=: data
data // ArrayBuffer(0, 1, 2, 3, 4)
```

Constructing: Mutably Removing Elements

`ArrayBuffer` and other mutable sequences have additional methods to remove elements. The `-=` method removes the first element equal to the given parameter.

```
val a = mutable.ArrayBuffer(1, 2, 3, 3, 4)
a -= 3
a  // ArrayBuffer(1, 2, 3, 4)
```

The `--=` method removes all the elements found in another collection (which itself does not have to be mutable).

```
val a = mutable.ArrayBuffer(1, 2, 3, 3, 4)
a --= List(3, 4, 5)
a // ArrayBuffer(1, 2, 3)
```

Aside: Symbolic and Alphabetic Names

All symbolic method names, like += and ++, have alphabetic equivalents. For example, if you prefer you can use `addOne` instead of +=. Here's a list of the most common symbolic operators and their alphabetic equivalent:

- `+` is `updated`
- `++` is `concat`
- `-` is `removed`
- `--` is `removedAll`
- `+:` is `prepended`
- `:+` is `appended`
- `+=` is `addOne`
- `++=` is `addAll`

Use the Right Tool for the Job

Which should we choose when we have the choice of mutable and immutable operations? It depends on context but, in general, if we are using a mutable collection it should be because we specifically require mutable operations. Thus using immutable methods on a mutable collection is generally not idiomatic code. However using a mutable collection is itself generally not idiomatic.

Constructing: Summary

We have seen the following methods to construct collections:

- `++` to append collections; and
- `+=:`, `+=`, `++=`, `-=`, and `--=` for mutable operations on sequences.

Querying: Checking Predicates

Instead of finding data that matches a predicate (using `filter` or `filterNot`) we might want to know if any or all elements matches a predicate. The `exists` and `forall` methods do this, respectively.

```
val data = mutable.ArrayBuffer(1, 2, 3, 4)

// Is any element negative?
data.exists(x => x < 0) // false

// Are all elements even?
data.forall(x => x % 2 == 0) // false
```

Querying: Access by Index

`List` and `ArrayBuffer` are both examples of sequences. The elements of a sequence have a defined order and can be accessed by index. Indices start at 0.

```
val data = List(1, 2, 3)
data(0) // 1
data(2) // 3
data(3) // raises an exception
```

Aside: Apply Syntax

Calling an object as if it was a function is a shorthand for calling the `apply` method on that object. Instead of

```
data(0) // 1
```

we could write

```
data.apply(0)
```

Querying: Summary

We have seen the following methods to query properties of a collection

- `forall` to check if all elements match a predicate; and
- `exists` to check if at least one element matches a predicate.

We can also access elements of a sequence by index, using the `apply` method.

Reasoning about Transformations

When first encountering `map`, `flatMap`, and `foldLeft` it can be difficult to choose between them. It can help to understand the types of these operations as algebraic equations that we can use to reason about the right choice to make.

Transforming: Map Type Equation

The general type equation for `map` is shown below.

```
F[A].map(A => B) == F[B]
```

- `F` is the container type, such as `List` or `ArrayBuffer`;
- `A` is the type of the elements in the container, such as `Int` or `Boolean`; and
- `B` is the type to which we transform the elements.

Transforming: Reasoning about Map

```
F[A].map(A => B) == F[B]
```

The type equation for `map` allows us to reason about it using just types. For example, suppose we have a `List[Int]` and a function `Int => String`, and we want to produce a `List[String]`.

Then,

- `F` is `List`;
- `A` is `Int`; and
- `B` is `String`

Substituting these into the equation tells us that using `map` will produce `List[String]`, which is the type that we want.

Transforming: FlatMap Type Equation

We can write a general type equation for `flatMap`, just as we did for `map`.

```
F[A].flatMap(A => F[B]) == F[B]
```

- `F` is the container type, such as `List` or `ArrayBuffer`;
- `A` is the type of the elements in the container, such as `Int` or `Boolean`; and
- `B` is the type to which we transform the elements.

For example, if we have a `List[Int]` and a function `Int => List[String]`, the type signature tells us that we'll end up with a `List[String]` if we use `flatMap`.

Transforming: FoldLeft Type Equation

The type equation for `foldLeft` is

```
F[A].foldLeft(B)((B, A) => B) == B
```

- `F` is the container type, such as `List` or `ArrayBuffer`;
- `A` is the type of the elements in the container, such as `Int` or `Boolean`; and
- `B` is the type to which we transform the container and its elements.

Transforming: Reasoning Algebraically

How we should use type equations to reason depends on the situation. We have given a few examples above. Here are a few more general principles we can extract from the equations.

- If the input has type `F[A]` and the output has type `F[B]` then we should consider `map` and `flatMap` before we look at `foldLeft`.
- If the output does not have the same `F` collection type as the input we have to use `foldLeft`.
- We can use the type of transformation function to choose between `map` and `flatMap`.

Transformation: Summary

We've seen that we can reason about transformations using algebraic equations. The equations are:

- `F[A].map(A => B) == F[B]` for `map`;
- `F[A].flatMap(A => F[B]) == F[B]` for `flatMap`; and
- `F[A].foldLeft(B)((B, A) => B) == B` for `foldLeft`

Other Collections: Mutable Maps

There are mutable maps in the `scala.collection.mutable` package. They have additional methods that are similar to those on mutable sequences.

```
val data = scala.collection.mutable.HashMap.empty[String, Int]
data += ("a" -> 0)
data += ("b" -> 1)
data += ("c" -> 2)
data += ("d" -> 3)
data
// HashMap(a -> 0, b -> 1, c -> 2, d -> 3)
```

Other Collections: Miscellaneous

The collections library contains many other data structures. Here is a quick overview of the more commonly used types:

- `Set` is a collection that contains only one instance of a given element;
- `BitSet` is a mutable collection containing a set of bits;
- `LazyList` is a type of sequence that can have infinite length.

Summary

In this section we have covered additional material about collections:

- concatenating collections using `++`;
- operations on mutable collections;
- querying using `forall` and `exists`;
- accessing elements by index;
- reasoning using type equations; and
- briefly looked at other collection types like `Set`.