Chapter 23

For Expressions Revisited

Chapter 16 demonstrated that higher-order functions, such as map, flatMap, and filter, provide powerful constructions for dealing with lists. But sometimes the level of abstraction required by these functions makes a program a bit hard to understand.

Here's an example. Say you are given a list of persons, each defined as an instance of a classPerson. Class Person has fields indicating the person's name, whether he or she is male, and his or her children.

Here's the class definition:

```
scala> case class Person(name: String,
                         isMale: Boolean,
                         children: Person*)
```

Here's a list of some sample persons:

```
val lara = Person("Lara", false)
val bob = Person("Bob", true)
val julie = Person("Julie", false, lara, bob)
val persons = List(lara, bob, julie)
```

Now, say you want to find out the names of all pairs of mothers and their children in that list. Using map, flatMap and filter, you can formulate the following query:

```
scala> persons filter (p => !p.isMale) flatMap (p =>
           (p.children map (c => (p.name, c.name))))
res0: List[(String, String)] = List((Julie,Lara),
    (Julie,Bob))
```

You could optimize this example a bit by using a withFilter call instead of filter. This would avoid the creation of an intermediate data structure for female persons:

```
scala> persons withFilter (p => !p.isMale) flatMap (p =>
           (p.children map (c => (p.name, c.name))))
res1: List[(String, String)] = List((Julie,Lara),
    (Julie,Bob))
```

These queries do their job, but they are not exactly trivial to write or understand. Is there a simpler way? In fact, there is. Remember the for expressions in Section 7.3? Using a forexpression, the same example can be written as follows:

```
scala> for (p <- persons; if !p.isMale; c <- p.children)
       yield (p.name, c.name)
res2: List[(String, String)] = List((Julie,Lara),
    (Julie,Bob))
```

The result of this expression is exactly the same as the result of the previous expression. What's more, most readers of the code would likely find the for expression much clearer than the previous query, which used the higher-order functions, map, flatMap, and withFilter.

However, the last two queries are not as dissimilar as it might seem. In fact, it turns out that the Scala compiler will translate the second query into the first one. More generally, all forexpressions that yield a result are translated by the compiler into combinations of invocations of the higher-order methods map, flatMap, and withFilter. All for loops without yield are translated into a smaller set of higher-order functions: just withFilter and foreach.

In this chapter, you'll find out first about the precise rules of writing for expressions. After that, you'll see how they can make combinatorial problems easier to solve. Finally, you'll learn how for expressions are translated, and how as a result, for expressions can help you "grow" the Scala language into new application domains.

23.1 FOR EXPRESSIONS

Generally, a for expression is of the form:

```
for ( seq ) yield expr
```

Here, seq is a sequence of generators, definitions, and filters, with semicolons between successive elements. An example is the for expression:

```
for (p <- persons; n = p.name; if (n startsWith "To"))
yield n
```

This for expression contains one generator, one definition, and one filter. As mentioned inSection 7.3 here, you can also enclose the sequence in braces instead of parentheses. Then the semicolons become optional:

```
for {
  p <- persons              // a generator
  n = p.name                // a definition
  if (n startsWith "To")    // a filter
} yield n
```

A generator is of the form:

```
pat <- expr
```

The expression expr typically returns a list, even though you will see later that this can be generalized. The pattern pat gets matched one-by-one against all elements of that list. If the match succeeds, the variables in the pattern get bound to the corresponding parts of the element, just the way it is described in Chapter 15. But if the match fails, no MatchError is thrown. Instead, the element is simply discarded from the iteration.

In the most common case, the pattern pat is just a variable x, as in x <- expr. In that case, the variable x simply iterates over all elements returned by expr.

A definition is of the form:

```
pat = expr
```

This definition binds the pattern pat to the value of expr, so it has the same effect as a valdefinition:

```
val x = expr
```

The most common case is again where the pattern is a simple variable x (*e.g.*, x = expr). This defines x as a name for the value expr.

A filter is of the form:

```
if expr
```

Here, expr is an expression of type Boolean. The filter drops from the iteration all elements for which expr returns false.

Every for expression starts with a generator. If there are several generators in a forexpression, later generators vary more rapidly than earlier ones. You can verify this easily with the following simple test:

```
scala> for (x <- List(1, 2); y <- List("one", "two"))
         yield (x, y)
res3: List[(Int, String)] =
  List((1,one), (1,two), (2,one), (2,two))
```

23.2 THE N-QUEENS PROBLEM

A particularly suitable application area of for expressions are combinatorial puzzles. An example of such a puzzle is the 8-queens problem: Given a standard chess-board, place eight queens such that no queen is in check from any other (a queen can check another piece if they are on the same column, row, or diagonal). To find a solution to this problem, it's actually simpler to generalize it to chess-boards of arbitrary size. Hence, the problem is to place *N*queens on a chess-board of *N* x *N* squares, where the size *N* is arbitrary. We'll start numbering cells at one, so the upper-left cell of an *N* x *N* board has coordinate (1, 1) and the lower-right cell has coordinate (*N*, *N*).

To solve the N-queens problem, note that you need to place a queen in each row. So you could place queens in successive rows, each time checking that a newly placed queen is not in check from any other queens that have already been placed. In the course of this search, it might happen that a queen that needs to be placed in row *k* would be in check in all fields of that row from queens in row 1 to *k*-1. In that case, you need to abort that part of the search in order to continue with a different configuration of queens in columns 1 to *k*-1.

An imperative solution to this problem would place queens one by one, moving them around on the board. But it looks difficult to come up with a scheme that really tries all possibilities. A more functional approach represents a solution directly, as a value. A solution consists of a list of coordinates, one for each queen placed on the board. Note, however, that a full solution can not be found in a single step. It needs to be built up gradually, by occupying successive rows with queens.

This suggests a recursive algorithm. Assume you have already generated all solutions of placing *k* queens on a board of size *N* x *N*, where *k* is less than *N*. Each such solution can be presented by a list of length *k* of coordinates (row, column), where both row and column numbers range from 1 to *N*. It's convenient to treat these partial solution lists as stacks, where the coordinates of the queen in row *k* come first in the list, followed by the coordinates of the queen in row *k*-1, and so on. The bottom of the stack is the coordinate of the queen placed in the first row of the board. All solutions together are represented as a list of lists, with one element for each solution.

Now, to place the next queen in row *k*+1, generate all possible extensions of each previous solution by one more queen. This yields another list of solution lists, this time of length *k*+1. Continue the process until you have obtained all solutions of the size of the chess-board *N*.

This algorithmic idea is embodied in function placeQueens below:

```
def queens(n: Int): List[List[(Int, Int)]] = {
  def placeQueens(k: Int): List[List[(Int, Int)]] =
    if (k == 0)
      List(List())
    else
      for {
        queens <- placeQueens(k - 1)
        column <- 1 to n
        queen = (k, column)
        if isSafe(queen, queens)
      } yield queen :: queens

  placeQueens(n)
}
```

The outer function queens in the program above simply calls placeQueens with the size of the board n as its argument. The task of the function application placeQueens(k) is to generate all partial solutions of length k in a list. Every element of the list is one solution, represented by a list of length k. So placeQueens returns a list of lists.

If the parameter k to placeQueens is 0, this means that it needs to generate all solutions of placing zero queens on zero rows. There is only one such solution: place no queen at all. This solution is represented by the empty list. So if k is zero, placeQueens returns List(List()), a list consisting of a single element that is the empty list. Note that this is quite different from the empty list List().
If placeQueens returns List(), this means no solutions, instead of a single solution consisting of no placed queens.

In the other case, where k is not zero, all the work of placeQueens is done in a for expression. The first generator of that for expression iterates through all solutions of placing k - 1 queens on the board. The second generator iterates through all possible columns on which the k'th queen might be placed. The third part of the for expression defines the newly considered queenposition to be the pair consisting of row k and each produced column. The fourth part of the forexpression is a filter which checks with isSafe whether the new queen is safe from check by all previous queens (the definition of isSafe will be discussed a bit later).

If the new queen is not in check from any other queens, it can form part of a partial solution, so placeQueens generates with queen :: queens a new solution. If the new queen is not safe from check, the filter returns false, so no solution is generated.

The only remaining bit is the isSafe method, which is used to check whether a given queen is in check from any other element in a list of queens. Here is its definition:

```
def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
  queens forall (q => !inCheck(queen, q))

def inCheck(q1: (Int, Int), q2: (Int, Int)) =
  q1._1 == q2._1 ||  // same row
  q1._2 == q2._2 ||  // same column
  (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // on diagonal
```

The isSafe method expresses that a queen is safe with respect to some other queens if it is not in check from any other queen. The inCheck method expresses that queens q1 and q2 are mutually in check.

It returns true in one of three cases:

1. If the two queens have the same row coordinate,
2. If the two queens have the same column coordinate,
3. If the two queens are on the same diagonal (*i.e.*, the difference between their rows and the difference between their columns are the same).

The first case—that the two queens have the same row coordinate—cannot happen in the application because placeQueens already takes care to place each queen in a different row. So you could remove the test without changing the functionality of the program.

23.3 QUERYING WITH FOR EXPRESSIONS

The for notation is essentially equivalent to common operations of database query languages. For instance, say you are given a database named books, represented as a list of books, whereBook is defined as follows:

```
case class Book(title: String, authors: String*)
```

Here is a small example database represented as an in-memory list:

```
val books: List[Book] =
  List(
    Book(
      "Structure and Interpretation of Computer Programs",
      "Abelson, Harold", "Sussman, Gerald J."
    ),
    Book(
      "Principles of Compiler Design",
      "Aho, Alfred", "Ullman, Jeffrey"
    ),
    Book(
      "Programming in Modula-2",
      "Wirth, Niklaus"
    ),
```

```
    Book(
      "Elements of ML Programming",
      "Ullman, Jeffrey"
    ),
    Book(
      "The Java Language Specification", "Gosling, James",
      "Joy, Bill", "Steele, Guy", "Bracha, Gilad"
    )
  )
```

To find the titles of all books whose author's last name is "Gosling":

```
scala> for (b <- books; a <- b.authors
            if a startsWith "Gosling")
       yield b.title
res4: List[String] = List(The Java Language Specification)
```

Or to find the titles of all books that have the string "Program" in their title:

```
scala> for (b <- books if (b.title indexOf "Program") >= 0)
       yield b.title
res5: List[String] = List(Structure and Interpretation of
  Computer Programs, Programming in Modula-2, Elements of ML
    Programming)
```

Or to find the names of all authors who have written at least two books in the database:

```
scala> for (b1 <- books; b2 <- books if b1 != b2;
           a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
       yield a1
res6: List[String] = List(Ullman, Jeffrey, Ullman, Jeffrey)
```

The last solution is still not perfect because authors will appear several times in the list of results. You still need to remove duplicate authors from result lists. This can be achieved with the following function:

```
scala> def removeDuplicates[A](xs: List[A]): List[A] = {
         if (xs.isEmpty) xs
         else
           xs.head :: removeDuplicates(
             xs.tail filter (x => x != xs.head)
           )
       }
removeDuplicates: [A](xs: List[A])List[A]

scala> removeDuplicates(res6)
res7: List[String] = List(Ullman, Jeffrey)
```

It's worth noting that the last expression in method removeDuplicates can be equivalently expressed using a for expression:

```
xs.head :: removeDuplicates(
  for (x <- xs.tail if x != xs.head) yield x
)
```

23.4 TRANSLATION OF FOR EXPRESSIONS

Every for expression can be expressed in terms of the three higher-order functions map,flatMap, and withFilter. This section describes the translation scheme, which is also used by the Scala compiler.

Translating for expressions with one generator

First, assume you have a simple for expression:

```
for (null <- expr_1) yield expr_2
```

where *x* is a variable. Such an expression is translated to:

```
expr_1.map(null => expr_2)
```

Translating for expressions starting with a generator and a filter

Now, consider for expressions that combine a leading generator with some other elements.
Afor expression of the form:

```
for (null <- expr_1 if expr_2) yield expr_3
```

is translated to:

```
for (null <- expr_1 withFilter (null => expr_2)) yield expr_3
```

This translation gives another for expression that is shorter by one element than the original, because an if element is transformed into an application of withFilter on the first generator expression. The translation then continues with this second expression, so in the end you obtain:

```
expr_1 withFilter (null => expr_2) map (null => expr_3)
```

The same translation scheme also applies if there are further elements following the filter. Ifseq is an arbitrary sequence of generators, definitions, and filters, then:

```
for (null <- expr_1 if expr_2; seq) yield expr_3
```

is translated to:

```
for (null <- expr_1 withFilter expr_2; seq) yield expr_3
```

Then translation continues with the second expression, which is again shorter by one element than the original one.

Translating for expressions starting with two generators

The next case handles for expressions that start with two generators, as in:

```
for (null <- expr_1; null <- expr_2; seq) yield expr_3
```

Again, assume that seq is an arbitrary sequence of generators, definitions, and filters. In fact,seq might also be empty, and in that case there would not be a semicolon after expr_2. The translation scheme stays the same in each case. The for expression above is translated to an application of flatMap:

```
expr_1.flatMap(null => for (null <- expr_2; seq) yield expr_3)
```

This time, there is another for expression in the function value passed to flatMap. That forexpression (which is again simpler by one element than the original) is in turn translated with the same rules.

The three translation schemes given so far are sufficient to translate all for expressions that contain just generators and filters, and where generators bind only simple variables. Take, for instance, the query, "find all authors who have published at least two books," from Section 23.3:

```
for (b1 <- books; b2 <- books if b1 != b2;
     a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
yield a1
```

This query translates to the following map/flatMap/filter combination:

```
books flatMap (b1 =>
  books withFilter (b2 => b1 != b2) flatMap (b2 =>
    b1.authors flatMap (a1 =>
      b2.authors withFilter (a2 => a1 == a2) map (a2 =>
        a1))))
```

The translation scheme presented so far does not yet handle generators that bind whole patterns instead of simple variables. It also does not yet cover definitions. These two aspects will be explained in the next two sub-sections.

Translating patterns in generators

The translation scheme becomes more complicated if the left hand side of generator is a pattern, pat, other than a simple variable. The case where the for expression binds a tuple of variables is still relatively easy to handle. In that case, almost the same scheme as for single variables applies.

A for expression of the form:

```
for ((null, ..., null) <- expr_1) yield expr_2
```

translates to:

```
expr_1.map { case (null, ..., null) => expr_2 }
```

Things become a bit more involved if the left hand side of the generator is an arbitrary pattern pat instead of a single variable or a tuple.

In this case:

```
for (pat <- expr_1) yield expr_2
```

translates to:

```
expr_1 withFilter {
  case pat => true
  case _ => false
} map {
  case pat => expr_2
}
```

That is, the generated items are first filtered and only those that match pat are mapped. Therefore, it's guaranteed that a pattern-matching generator will never throw a MatchError.

The scheme here only treated the case where the for expression contains a single pattern-matching generator. Analogous rules apply if the for expression contains other generators, filters or definitions. Because these additional rules don't add much new insight, they are omitted from discussion here. If you are interested, you can look them up in the *Scala Language Specification* [Ode11].

Translating definitions

The last missing situation is where a for expression contains embedded definitions. Here's a typical case:

```
for (null <- expr_1; null = expr_2; seq) yield expr_3
```

Assume again that seq is a (possibly empty) sequence of generators, definitions, and filters. This expression is translated to this one:

```
for ((null, null) <- for (null <- expr_1) yield (null, expr_2); seq)
yield expr_3
```

So you see that expr_2 is evaluated each time there is a new *x* value being generated. This re-evaluation is necessary because expr_2 might refer to *x* and so needs to be re-evaluated for changing values of *x*. For you as a programmer, the conclusion is that it's probably not a good idea to have definitions embedded in for expressions that do not refer to variables bound by some preceding generator, because re-evaluating such expressions would be wasteful. For instance, instead of:

```
for (x <- 1 to 1000; y = expensiveComputationNotInvolvingX)
yield x * y
```

it's usually better to write:

```
val y = expensiveComputationNotInvolvingX
for (x <- 1 to 1000) yield x * y
```

Translating for loops

The previous subsections showed how for expressions that contain a yield are translated.What about for loops that simply perform a side effect without returning anything? Their translation is similar, but simpler than for expressions. In principle, wherever the previous translation scheme used a map or a flatMap in the translation, the translation scheme for forloops uses just a foreach.

For instance, the expression:

```
for (null <- expr_1) body
```

translates to:

```
expr_1 foreach (null => body)
```

A larger example is the expression:

```
for (null <- expr_1; if expr_2; null <- expr_3) body
```

This expression translates to:

```
expr_1 withFilter (null => expr_2) foreach (null =>
  expr_3 foreach (null => body))
```

For example, the following expression sums up all elements of a matrix represented as a list of lists:

```
var sum = 0
for (xs <- xss; x <- xs) sum += x
```

This loop is translated into two nested foreach applications:

```
var sum = 0
xss foreach (xs =>
  xs foreach (x =>
    sum += x))
```

23.5 GOING THE OTHER WAY

The previous section showed that for expressions can be translated into applications of the higher-order functions map, flatMap, and withFilter. In fact, you could equally go the other way: Every application of a map, flatMap, or filter can be represented as a for expression.

Here are implementations of the three methods in terms of for expressions. The methods are contained in an object Demo to distinguish them from the standard operations on Lists. To be concrete, the three functions all take a List as parameter, but the translation scheme would work just as well with other collection types:

```
object Demo {
  def map[A, B](xs: List[A], f: A => B): List[B] =
    for (x <- xs) yield f(x)

  def flatMap[A, B](xs: List[A], f: A => List[B]): List[B] =
    for (x <- xs; y <- f(x)) yield y

  def filter[A](xs: List[A], p: A => Boolean): List[A] =
    for (x <- xs if p(x)) yield x
}
```

Not surprisingly, the translation of the for expression used in the body of Demo.map will produce a call to map in class List. Similarly, Demo.flatMap and Demo.filter translate to flatMap andwithFilter in class List. So this little demonstration shows that for expressions really are equivalent in their expressiveness to applications of the three functions map, flatMap, andwithFilter.

23.6 GENERALIZING FOR

Because the translation of for expressions only relies on the presence of methods map, flatMap, and withFilter, it is possible to apply the for notation to a large class of data types.

You have already seen for expressions over lists and arrays. These are supported because lists, as well as arrays, define operations map, flatMap, and withFilter. Because they define aforeach method as well, for loops over these data types are also possible.

Besides lists and arrays, there are many other types in the Scala standard library that support the same four methods and therefore allow for expressions. Examples are ranges, iterators, streams, and all implementations of sets. It's also perfectly possible for your own data types to support for expressions by defining the necessary methods. To support the full range of forexpressions and for loops, you need to define map, flatMap, withFilter, and foreach as methods of your data type. But it's also possible to define a subset of these methods, and thereby support a subset of all possible for expressions or loops.

Here are the precise rules:

- If your type defines just map, it allows for expressions consisting of a single generator.
- If it defines flatMap as well as map, it allows for expressions consisting of several generators.
- If it defines foreach, it allows for loops (both with single and multiple generators).
- If it defines withFilter, it allows for filter expressions starting with an if in the forexpression.

The translation of for expressions happens before type checking. This allows for maximum flexibility because the only requirement is that the result of expanding a for expression type checks. Scala defines no typing rules for the for expressions themselves, and does not require that methods map, flatMap, withFilter, or foreach have any particular type signatures.

Nevertheless, there is a typical setup that captures the most common intention of the higher order methods to which for expressions translate. Say you have a parameterized class, C, which typically would stand for some sort of collection. Then it's quite natural to pick the following type signatures for map, flatMap, withFilter, and foreach:

```
abstract class C[A] {
  def map[B](f: A => B): C[B]
  def flatMap[B](f: A => C[B]): C[B]
  def withFilter(p: A => Boolean): C[A]
  def foreach(b: A => Unit): Unit
}
```

That is, the map function takes a function from the collection's element type A to some other type B. It produces a new collection of the same kind C, but with B as the element type. TheflatMap method takes a function f from A to some C-collection of Bs and produces a C-collection of Bs. The withFilter method takes a predicate function from the collection's element type A toBoolean. It produces a collection of the same type as the one on which it is invoked. Finally, theforeach method takes a function from A to Unit and produces a Unit result:

In class C above, the withFilter method produces a new collection of the same class. That means that every invocation of withFilter creates a new C object, just the same as filter would work. Now, in the translation of for expressions, any calls to withFilter are always followed by calls to one of the other three methods. Therefore, the object created by withFilter will be taken apart by one of the other methods immediately afterwards. If objects of class C are large (think long sequences), you might want to avoid the creation of such an intermediate object. A standard technique is to let withFilter return not a C object but just a wrapper object that "remembers" that elements need to be filtered before being processed further.

Concentrating on just the first three functions of class C, the following facts are noteworthy. In functional programming, there's a general concept called a monad, which can explain a large number of types with computations, ranging from collections, to computations with state and I/O, backtracking computations, and transactions, to name a few. You can formulate functions map, flatMap, and withFilter on a monad, and, if you do, they end up having exactly the types given here.

Furthermore, you can characterize every monad by map, flatMap, and withFilter, plus a "unit" constructor that produces a monad from an element value. In an object-oriented language, this "unit" constructor is simply an instance constructor or a factory method. Therefore, map,flatMap, and withFilter can be seen as an object-oriented version of the functional concept of monad. Because for expressions are equivalent to applications of these three methods, they can be seen as syntax for monads.

All this suggests that the concept of for expression is more general than just iteration over a collection, and indeed it is. For instance, for expressions also play an important role in asynchronous I/O, or as an alternative notation for optional values. Watch out in the Scala libraries for occurrences of map, flatMap, and withFilter—when they are present, for expressions suggest themselves as a concise way of manipulating elements of the type.

23.7 CONCLUSION

In this chapter, you were given a peek under the hood of for expressions and for loops. You learned that they translate into applications of a standard set of higher-order methods. As a result, you saw that for expressions are really much more general than mere iterations over collections, and that you can design your own classes to support them.