Principles of Programming Languages (Lecture 8)

COMP 3031, Fall 2025

Lionel Parreaux

Combinatorial Search and For-Expressions

Handling Nested Sequences

We can extend the usage of higher order functions on sequences to many calculations which are usually expressed using nested loops.

Example: Given a positive integer `n`, find all pairs of positive integers `i` and `j`, with `1 <= j < i < n` such that `i + j` is prime.

For example, if `n = 7`, the sought pairs are

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

Algorithm

A natural way to do this is to:

- Generate the sequence of all pairs of integers `(i, j)` such that `1 <= j < i < n`.
- Filter the pairs for which `i + j` is prime.

Algorithm

A natural way to do this is to:

- Generate the sequence of all pairs of integers `(i, j)` such that `1 <= j < i < n`.
- Filter the pairs for which `i + j` is prime.

One natural way to generate the sequence of pairs is to:

- Generate all the integers `i` between `1` and `n` (excluded).
- For each integer `i`, generate the list of pairs `(i, 1), ..., (i, i-1)`.

Algorithm

A natural way to do this is to:

- Generate the sequence of all pairs of integers `(i, j)` such that `1 <= j < i < n`.
- Filter the pairs for which `i + j` is prime.

One natural way to generate the sequence of pairs is to:

- Generate all the integers `i` between `1` and `n` (excluded).
- For each integer `i`, generate the list of pairs `(i, 1), ..., (i, i-1)`.

This can be achieved by combining `until` and `map`:

```
(1 until n).map(i =>
  (1 until i).map(j => (i, j)))
```

Generate Pairs

The previous step gave a sequence of sequences, let's call it `xss`.

We can combine all the sub-sequences using `foldRight` with `++`:

Note: `++` is like `:::` but for aribitrary sequences.

```
xss.foldRight(Seq[(Int, Int)]())(_ ++ _)
```

Or, equivalently, we use the built-in method `flatten`

```
xss.flatten
```

Generate Pairs

The previous step gave a sequence of sequences, let's call it `xss`.

We can combine all the sub-sequences using `foldRight` with `++`:

Note: `++` is like `:::` but for aribitrary sequences.

```
xss.foldRight(Seq[(Int, Int)]())(_ ++ _)
```

Or, equivalently, we use the built-in method `flatten`

```
xss.flatten
```

This gives:

```
((1 until n).map(i =>
  (1 until i).map(j => (i, j)))).flatten
```

Generate Pairs (2)

Here's a useful law:

```
xs.flatMap(f)  =   xs.map(f).flatten
```

Hence, the above expression can be simplified to

```
(1 until n).flatMap(i =>
    (1 until i).map(j => (i, j)))
```

Assembling the pieces

By reassembling the pieces, we obtain the following expression:

```
(1 until n)
  .flatMap(i => (1 until i).map(j => (i, j)))
  .filter((x, y) => isPrime(x + y))
```

This works, but is a bit heavyweight and hard to read.

Is there a simpler way?

For-Expressions

Higher-order functions such as `map`, `flatMap` or `filter` provide powerful constructs for manipulating lists.

But sometimes the level of abstraction required by these function make the program difficult to understand.

In this case, Scala's `for` expression notation can help.

For-Expression Example

Let `persons` be a list of elements of class `Person`, with fields `name` and `age`.

```
case class Person(name: String, age: Int)
```

To obtain the names of persons over 20 years old, you can write:

```
for p <- persons if p.age > 20 yield p.name
```

which is equivalent to:

```
persons
  .filter(p => p.age > 20)
  .map(p => p.name)
```

The for-expression is similar to loops in imperative languages, except that it builds a list of the results of all iterations.

Syntax of For

A for-expression is of the form

```
for s yield e
```

where `s` is a sequence of *generators* and *filters*, and `e` is an expression whose value is returned by an iteration.

- A *generator* is of the form `p <- e`, where `p` is a pattern and `e` an expression whose value is a collection.
- A *filter* is of the form `if f` where `f` is a boolean expression.
- The sequence must start with a generator.
- If there are several generators in the sequence, the last generators vary faster than the first.

Use of For

Here are two examples which were previously solved with higher-order functions:

Given a positive integer `n`, find all the pairs of positive integers `(i, j)` such that `1 <= j < i < n`, and `i + j` is prime.

```
for
  i <- 1 until n
  j <- 1 until i
  if isPrime(i + j)
yield (i, j)
```

Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =
```

Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =

  (for (x, y) <- xs.zip(ys) yield x * y).sum
```

Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =

  (for (x, y) <- xs.zip(ys) yield x * y).sum
```

Question: What will the following produce?

```
  (for x <- xs; y <- ys yield x * y).sum
```

Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =

  (for (x, y) <- xs.zip(ys) yield x * y).sum
```

Question: What will the following produce?

```
  (for x <- xs; y <- ys yield x * y).sum
```

Answer: It would multiply *every* element of `xs` with *every* element of `ys` and sum up the results.

Combinatorial Search Example

Sets

Sets are another basic abstraction in the Scala collections.

A set is written analogously to a sequence:

```
val fruit = Set("apple", "banana", "pear")
val s = (1 to 6).toSet
```

Most operations on sequences are also available on sets:

```
s.map(_ + 2)
fruit.filter(_.startsWith("app"))
s.nonEmpty
```

(see Scaladoc for `scala.Set` for a list of all supported operations)

Sets vs Sequences

The principal differences between sets and sequences are:

1. Sets are unordered; the elements of a set do not have a predefined order in which they appear in the set
2. sets do not have duplicate elements:

```
s.map(_ / 2)        // Set(2, 0, 3, 1)
```

3. The fundamental operation on sets is `contains`:

```
s.contains(5)       // true
```

Example: N-Queens

The eight queens problem is to place eight queens on a chessboard so that no queen is threatened by another.

- In other words, there can't be two queens in the same row, column, or diagonal.

We now develop a solution for a chessboard of any size, not just 8.

One way to solve the problem is to place a queen on each row iteratively.

Once we have placed `k - 1` queens, one must place the `k`th queen in a column where it's not "in check" with any other queen on the board.

Algorithm

We can solve this problem with a recursive algorithm:

- Suppose that we have already generated all the solutions consisting of placing `k-1` queens on a board of size `n`.
- Each solution is represented by a list of `k-1` column indices.
- The column index of the queen in the `(k-1)`th row comes first in the list, followed by the column index of the queen in row `k-2`, etc.
- The solution set is thus represented as a set of lists, with one element for each solution.
- Now, to place the `k`th queen, we generate all possible extensions of each solution preceded by a new queen:

Implementation

```scala
def queens(n: Int) =
  def placeQueens(k: Int): Set[List[Int]] =
    if k == 0 then Set(Nil)
    else
      for
        prevSol <- placeQueens(k - 1)
        col <- 0 until n
        if isSafe(col, prevSol)
      yield col :: prevSol
  placeQueens(n)
```

Exercise

Write a function

```
def isSafe(col: Int, queens: List[Int]): Boolean
```

which tests if a queen placed in an indicated column `col` is secure amongst the other placed queens.

It is assumed that the new queen is placed in the next available row after the other placed queens (in other words: in row `queens.length`).

Exercise

```scala
def isSafe(col: Int, queens: List[Int]): Boolean =
  !checks(col, 1, queens)
```

where `checks` takes in an additional parameter `delta`, the distance in rows between the lowest row of `queens` and the row where the current queen is being placed.

```scala
// Does any queen in ‘queens’ check a new queen placed ‘delta’ columns lower?
def checks(col: Int, delta: Int, queens: List[Int]): Boolean = queens match
  case qcol :: others =>
    qcol == col                         // vertical check
    || (qcol - col).abs == delta        // diagonal check
    || checks(col, delta + 1, others)
  case Nil =>
    false
```

How Other Languages Do It

How Other Languages Do It

Many languages feature the idea of declaratively *mapping*, *flattening*, and *filtering* collections while producing new ones. Examples:

How Other Languages Do It

Many languages feature the idea of declaratively *mapping*, *flattening*, and *filtering* collections while producing new ones. Examples:

Python

```
[ (i, j) for i in range(1, n) for j in range(1, i) if is_prime(i+j) ]
```

How Other Languages Do It

Many languages feature the idea of declaratively *mapping*, *flattening*, and *filtering* collections while producing new ones. Examples:

Python

```
[ (i, j) for i in range(1, n) for j in range(1, i) if is_prime(i+j) ]
```

Haskell

```
[ (i, j) | i <- [1..n], j <- [1..i], is_prime (i+j) ]
```

How Other Languages Do It

Many languages feature the idea of declaratively *mapping*, *flattening*, and *filtering* collections while producing new ones. Examples:

Python

```
[ (i, j) for i in range(1, n) for j in range(1, i) if is_prime(i+j) ]
```

Haskell

```
[ (i, j) | i <- [1..n], j <- [1..i], is_prime (i+j) ]
```

F#

```
[for i in [1 .. n] do for j in [1 .. i] do if is_prime i j then yield (i, j)]
```