Translation of For

Principles of Functional Programming

For-Expressions and Higher-Order Functions

The syntax of `for` is closely related to the higher-order functions `map`, `flatMap` and `filter`.

First of all, these functions can all be defined in terms of `for`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  for x <- xs yield f(x)

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

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

Translation of For (1)

In reality, the Scala compiler expresses for-expressions in terms of `map`, `flatMap` and a lazy variant of `filter`.

Here is the translation scheme used by the compiler (we limit ourselves here to simple variables in generators)

1. A simple for-expression

```
for x <- e1 yield e2
```

is translated to

```
e1.map(x => e2)
```

Translation of For (2)

2. A for-expression

```
for x <- e1 if f; s yield e2
```

where `f` is a filter and `s` is a (potentially empty) sequence of generators and filters, is translated to

```
for x <- e1.withFilter(x => f); s yield e2
```

(and the translation continues with the new expression)

You can think of `withFilter` as a variant of `filter` that does not produce an intermediate list, but instead applies the following `map` or `flatMap` function application only to those elements that passed the test.

Translation of For (3)

3. A for-expression

```
for x <- e1; y <- e2; s yield e3
```

where `s` is a (potentially empty) sequence of generators and filters, is translated into

```
e1.flatMap(x => for y <- e2; s yield e3)
```

(and the translation continues with the new expression)

Example

Take the for-expression that computed pairs whose sum is prime:

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

Applying the translation scheme to this expression gives:

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

This is almost exactly the expression which we came up with first!

Exercise

Translate

```
for b <- books; a <- b.authors if a.startsWith("Bird")
yield b.title
```

into higher-order functions.

Exercise

```
for b <- books; a <- b.authors if a.startsWith("Bird")
yield b.title
```

The expression above expands to which of the following two expressions?

0
```
books.flatMap(b =>
  b.authors.withFilter(a =>
    a.startsWith("Bird")).map(a => b.title))
```

0
```
books.map(b =>
  b.authors.flatMap(a =>
    if a.startsWith("Bird") then b.title))
```

Generalization of `for`

Interestingly, the translation of for is not limited to lists or sequences, or even collections;

It is based solely on the presence of the methods `map`, `flatMap` and `withFilter`.

This lets you use the for syntax for your own types as well – you must only define `map`, `flatMap` and `withFilter` for these types.

There are many types for which this is useful: arrays, iterators, databases, optional values, parsers, etc.

For and Databases

For example, `books` might not be a list, but a database stored on some server.

As long as the client interface to the database defines the methods `map`, `flatMap` and `withFilter`, we can use the `for` syntax for querying the database.

This is the basis of data base connection frameworks such as *Slick* or *Quill*, as well as big data platforms such as *Spark*.