Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

October 17, 2017

Filters in For Expressions

This is a filter

```
for (x <- xs if x >= 0)
  yield square(x) + 1
```

Filters in For Expressions

- Filters are attached to generators
- A given generator can have zero or more filters

Filters in For Expressions

```
for (
  x <- xs
  if x >= 0
  if x % 2 == 0
) yield square(x) + 1
```

Clauses Can Be Enclosed in Braces Instead of Parentheses

```
for {
  x <- xs
  if x >= 0
  if x % 2 == 0
} yield square(x) + 1
```

For Expressions Can Include Multiple Generators

```
for {
  x <- xs
  if x >= 0
  y <- ys
  if y % 2 == 0
} yield x * y
```

For Expressions Can Include Local Bindings

```
for {
  x <- xs
  if x >= 0
  square = x * x
  y <- ys
  if y % square == 0
} yield x * y
```

Generators Can Specify Arbitrary Patterns

```
val xs = Cons(Square(4),
           Cons(Circle(3),
             Cons(Rectangle(2,3),
               Empty)))

for {
  Rectangle(x,y) <- xs
} yield x * y
↦*
Cons(6.0, Empty)
```

Generators Can Specify Arbitrary Patterns

- Elements of the collection that do not match the pattern are filtered
- Effectively, a pattern in a `for` expression serves as part of a generator and a filter

Guidelines on Using For Expressions

- Prefer `for` expressions to maps and filters
- They tend to be easier to read:
  - All bindings and collections iterated over are listed up front
- Prefer {...} to (...) around non-trivial *for* clauses

For vs Map

- Compare:

```
for (x <- xs if x >= 0)
  yield square(x) + 1
```

- To:

```
xs.filter(_ >= 0).map(square(_) + 1)
```

For Expressions and Database Queries

- `for` expressions are similar to standard database queries

- Consider a simple in-memory database of books, represented as a list of Book instances *(Odersky et al 2012)*:

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

For Expressions and Database Queries

```
val books: List[Book] =
  Cons(
    Book(
      “Structure and Interpretation of Computer Programs”,
      “Abelson, Harold”, “Sussman, Gerald J.”
    ),
    Book(
      “How to Design Programs”,
      “Felleisen, Matthias”, “Findler, Robert Bruce”,
      “Flatt, Mathew”, “Krishnamurthi, Shriram”
    ),
    Book(
      “Programming in Scala”,
      “Odersky, Martin”, “Spoon, Lex”, “Venners, Bill”
    )
    …
  )
```

Finding All Books Whose Author Has Last Name “Sussman”

```
for {
  b <- books
  a <- b.authors
  if a startsWith “Sussman,”
} yield b.title
```

Finding All Books That Have The String “Program” In the Title

```
for {
  b <- books
  if b.title indexOf “Program” >= 0
} yield b.title
```

Finding All Authors That Have Written More Than One Book in the Database

```
for {
  b1 <- books
  b2 <- books if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
} yield a1
```

Translating For Expressions

- For expressions are simply translated to maps, flatMaps, and filters by the Scala compiler
- Translation occurs *before* type checking
    - Why is this preferable?
- We start by considering only for expressions with generators that bind simple names (no patterns)

Translating For Expressions With A Single Generator

```
for (x <- expr1) yield expr2
              ↦
  expr1.map(x => expr2)
```

Translating For Expressions With a Generator and a Filter

```
for (x <- expr1 if expr2) yield expr3
↦
for (x <- expr1 withFilter (x => expr2)) yield expr3
```

Translating For Expressions With a Generator and a Filter

```
for (x <- expr1 if expr2) yield expr3
↦
for (x <- expr1 withFilter (x => expr2)) yield expr3
↦
expr1 withFilter (x => expr2) map (x => expr3)
```

For now, read this as "filter". We will return to it.

Translating For Expressions Starting With a Generator and a Filter

```
for (x <- expr1 if expr2; seq) yield expr3
↦
for (x <- expr1 withFilter (x => expr2); seq)
  yield expr3
```

Translating For Expressions Starting With Two Generators

```
for (x <- expr1; y <- expr2; seq) yield expr3
↦
expr1.flatMap(x => for (y <- expr2; seq) yield expr3)
```

Translating For Expressions Example

```
for (b1 <- books; b2 <- books if b1 != b2;
     a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
yield a1
↦
books flatMap (b1 =>
  books withFilter (b2 => b1 != b2) flatMap (b2 =>
    b1.authors flatMap (a1 =>
      b2.authors withFilter (a2 => a1 == a2)
        map (a2 => a1))))
```

Translating Patterns in Generators

```
for (pat <- expr1) yield expr2
↦
expr1 withFilter (_ match {
  case pat => true
  case _ => false
}) map (_ match {
  case pat => expr2
})
```

Other cases with patterns and for expressions are similar

Generalizing For Expressions

- Because for expressions are simply translated to expressions involving `map`, `flatMap`, and `withFilter`, we can use `for` expressions over our own collections
- We need only define: `map`, `flatMap` and `withFilter`
    - Because translation occurs before type checking, there is no particular type that a collection must subtype to be compatible with for-expressions

Generalizing For Expressions

- We can even define a subset of these methods and use our collection only in `for` expressions that translate to our subset!
  - For example, if we do not define `withFilter`, we cannot use our collection in a for expression with a filter

Generalizing For Expressions

- Because translation occurs before type checking, there is no particular signature that our methods map, `flatMap` and `withFilter` must satisfy!
  - All that is required is that the resulting, translated program passes type checking

The WithFilter Function

- In our own List implementation, we could simply define `withFilter` as filter, and our collection would work with for expressions

- The idea behind `withFilter` is that it is often advantageous to simply wrap the collection in a view that performs the given filter on the next `map`

- Because no particular type signature is required, we need only define `map` and `flatMap` on our wrapper

The WithFilter Function

```
abstract class List[+T] {
  …
  def withFilter[S >: T, U](p: S => Boolean) =
    WithFilter[S](p,this)
}
```

The WithFilter Function

```
case class WithFilter[T](p: T => Boolean, xs: List[T]) {
  def map[U](f: T => U): List[U] = {
    xs match {
      case Empty => Empty
      case Cons(y,ys) => {
        val rest = WithFilter(p,ys) map f
        if (p(y)) Cons(f(y), rest)
        else rest
      }
    }
  }
  …
}
```

The WithFilter Function

- Because results of withFilter are immediately taken apart by a `map` or a `flatMap`, we can still think of the result of a `withFilter` as being an instance of the original collection

Typical Structure of a Class That Works With For Expressions

```
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]
}
```

Monads

- In functional programming, a *monad* can be defined as a type for which we can formulate
  - The functions `map`, `flatMap`, and `withFilter`
  - A “unit constructor” which produces a monad from an element value
    - In an object-oriented language, we can think of the “unit constructor” simply as a constructor or a factory method

Monads

Because `for` expressions work over precisely those datatypes for which we can formulate the functions that characterize monads, we can think of `for` expressions as syntax for computing with monads

Monads

- But monads are able to characterize far more than just collections:
  - I/O
  - State
  - Transactions
  - Options
  - etc.

Monads

- Thus, `for` expressions can be used in far more general contexts than simply walking over collections
- When looking at library classes, watch for implementations of `map, flatMap, withFilter`
- When these functions are defined, consider expressing your computation with `for` expressions

The Environment Model of Reduction

Limitations of the Substitution Model of Reduction

- Consider the following function definition:

```
def makeOddBooster(n: Int) = {
  require(n >= 0)
  def isEven(n: Int): Boolean = {
    (n == 0) || isOdd(n - 1)
  }
  def isOdd(n: Int): Boolean = {
    !isEven(n)
  }
  (m: Int) => if (isEven(m)) m else m + n
}
```

Limitations of the Substitution Model of Reduction

- Our `makeOddBooster` function cannot be expanded before it is returned
- It must remember the context in which it was formed

The Environment Model of Reduction

- *Name environments* map names to values
- Every expression is evaluated in the context of a *name environment*

The Environment Model of Reduction

- To evaluate a *name*, simply reduce to the value it is mapped to in the environment

The Environment Model of Reduction

- To evaluate a function, reduce it to a *lexical closure,* which consists of two parts:
  - The body of the function
  - The environment in which the body occurs

The Environment Model of Reduction

- To evaluate an application of a closure
  - Extend the environment of the closure, mapping the function's parameters to argument values
  - Evaluate the body of the closure in this new environment

Constructs that Add New Names to the Environment

- `val`
- `def`
- `object`
- `case class`
- `import`
- Bindings in patterns
- Function applications

Example Evaluation

```
             makeOddBooster(3)(1); ENV ↦
(m: Int) => if (isEven(m)) m else m + n)(1);
                  {n: Int = 3,
              isEven = Closure(…),
          isOdd = Closure(…)} ∪ ENV ↦
          if (isEven(m)) m else m + n;
    {m: Int = 1, n: Int = 3, …} ∪ ENV ↦*
            if (false) m else m + n;
     {m: Int = 1, n: Int = 3, …} ∪ ENV ↦
                     m + n;
     {m: Int = 1, n: Int = 3, …} ∪ ENV ↦
                    4; ENV
```

Takeaways

- Use Scala's *for-expressions* liberally
- Define `map`, `flatMap` and/or `withFilter` on your own monad-like data-structures to use them with *for*
- To evaluate non-trivial expressions, we need to keep track of an *environment*
- *Lexical closures* are used to store function values in an environment