Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC

Translating For Expressions

- It turns out that for expressions are translated to maps, flatMaps, and filters!
- 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

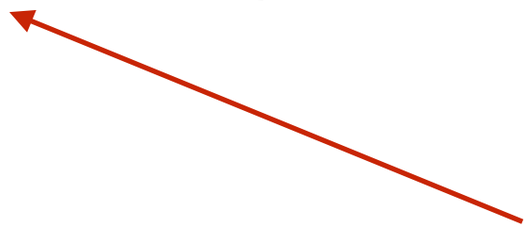
```
for (x <- expr1 if expr2) yield expr3
```

↦

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

↦

```
expr1 withFilter (x => expert) 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 {
  case pat => expr2
}
```

Translating Patterns in Generators

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

    - Because translation occurs before type checking, there is no particular type that our collection must subtype

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, 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` or

- 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 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 Type Checking

The Environment Model of Type Checking

- We have used environments in type checking to hold the bounds on type parameters

- They can also be used to record the types of names and function parameters

- Rather than thinking of typing rules as substitutions, we can think of them directly as assertions on expressions that we can reason with according to a logic

The Environment Model of Type Checking

- As a convenient notation, we express subtyping rules in the context of an environment by placing an environment to the left of a “turnstile” and a typing judgement to the right

$$\frac{}{\{T <: \mathtt{Any}\} \vdash T <: T}\ \texttt{[S-Refl1]}$$

The Environment Model of Type Checking

- As a convenient notation, we express subtyping rules in the context of an environment by placing an environment to the left of a “turnstile” and a typing judgement to the right

$$\frac{}{\{T <: N\} \vdash T <: T}\ \texttt{[S-Refl2]}$$

The Environment Model of Type Checking

- As a convenient notation, we express subtyping rules in the context of an environment by placing an environment to the left of a “turnstile” and a typing judgement to the right

$$\frac{}{\Delta \vdash T <: T}\ \texttt{[S-Refl]}$$

The Environment Model of Type Checking

- We express typing rules in the context of
  - a type parameter environment and
    - a type environment (mapping names to types)

- We place both environments to the left of the "turnstile" (separated by a semicolon) and a typing judgement to the right:

$$\frac{}{\Delta; \Gamma + \{\texttt{x:T}\} \vdash \texttt{x:T}}\ \texttt{[T-Var]}$$

The Environment Model of Type Checking

- Some typing judgements require assumptions
- We place assumed judgements above a horizontal bar (above the resulting type judgement)

$$\frac{\Delta; (\Gamma + \texttt{x:N}) \vdash \texttt{e:M}}{\Delta; \Gamma \vdash \texttt{((x:N) => e):(N => M)}}\ \texttt{[T-Arrow]}$$

The Environment Model of Type Checking

- Function applications involve checking the function and the arguments:

$$\frac{\Delta;\Gamma \vdash e_0 : \texttt{R => S};\ \Delta;\Gamma \vdash e_1 : \texttt{T};\ \Delta \vdash \texttt{T <: R};}{\Delta;\Gamma \vdash e_0\ e_1 : \texttt{S}}\ \texttt{[T-App]}$$

The Environment Model of Type Checking

- To type check an expression in a pair of environments:
  - Form a proof tree, where each node is the application of an inference rule
  - The root of the tree is the typing judgement we are trying to prove
  - Each premise in a given rule is the root of a subtree proving that premise

The Environment Model of Type Checking

- For each form of expression there is exactly one inference rule

- Therefore, proving a typing judgement is a simple recursive descent over the structure of an expression

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 *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

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
```