Comp 311 Functional Programming

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

Covariance

```
case object Empty extends List[Nothing] {
}

case class Cons[+T](head: T, tail: List[T])
extends List[T] {
}
```

Now we can define Empty as an object that extends the bottom of the List types

Covariance and Append

- The problem with our original declaration of append was that it was not general enough:
  - There is no reason to require that we always append lists of identical type
  - Really, we can append a `List[S]` for any supertype of our `List[T]`
  - The result will be of type `List[S]`

Lower Bounds on Type Parameters

- Thus far, we have allowed type parameters to include upper bounds:

```
S <: T
```

- They can also include lower bounds:

```
S >: T
```

- Or they can include both:

```
S >: T <: U
```

Parametric Functions

- Just as we can add type parameters to a class definition, we can also add them to a function definition
- The type parameters are in scope in the header and body of the function

Covariance and Append

```
abstract class List[+T] {
  def ++[S >: T](ys: List[S]): List[S]
}

case object Empty extends List[Nothing] {
  def ++[S](ys: List[S]) = ys
}

case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  def ++[S >: T](ys: List[S]) = Cons(head, tail ++ ys)
}
```

Map Revisited

```
abstract class List[+T] {
  …
  def map[U](f: T => U): List[U]
}
```

Why is this occurrence of T acceptable?

We Consider Specific Instantiations

```
abstract class List[Any] {
  …
  def map[U](f: Any => U): List[U]
}
abstract class List[String] {
  …
  def map[U](f: String => U): List[U]
}
```

Then List[String] is an acceptable subtype of List[Any]
provided that (String => U) >: (Any => U)
which requires that String <: Any.

Generalizing Our Rules

- In our example, type parameter `T` occurs as the parameter of an arrow type:
  - `(String => U)  >: (Any => U)` in `E` provided:
    - `String <: Any` in `E`
    - `U <: U` in `E`
  - So subtype `List[String] <: List[Any]` is permitted

To Check Variance, We Annotate Each Type Position With A *Polarity*

- Recursively descend a class definition:
  - At top level, all positions are positive
  - Polarity is flipped at method parameter positions
  - Polarity is flipped at method type parameter positions
  - Polarity is flipped at arrow type parameter positions

Annotating Polarity

```
abstract class List[+T] {
  def ++[S^- >: T^+](ys: List[S^-]): List[S^+]
  def map[U^-](f: T^+ => U^-): List[U^+]
}
```

We Generalize Our Rules for Checking Variance As Follows

- Covariant type parameters (declared with +) are allowed to occur only in positive locations
- Type parameters with no annotation are allowed to be used in all locations

Contravariance

Contravariance

- In general, we say that a parametric type `C` is contravariant with respect to its type parameter `S` if:

```
S <: T in E
```

implies

```
C[T] <: C[S] in E
```

- We must be careful that such relationships do not break the soundness of our type system

Contravariance

- Syntactically, contravariant type parameter declarations are annotated with a minus sign:

```
case class F[-A,+B]
```

To Check Variance, We Annotate Each Type Location With A *Polarity*

- Recursively descend a class definition:
  - At top level, all locations are positive
  - Polarity is flipped at method parameter positions
  - Polarity is flipped at method type parameter positions
  - Polarity is flipped at arrow type parameter positions
  - Polarity is flipped at positions of contravariant type parameters

Annotating Polarity

$$\begin{array}{l}
\texttt{abstract class List[+T] \{} \\
\quad\texttt{def ++[S}^{-}\texttt{ >: T}^{+}\texttt{](ys: List[S}^{-}\texttt{]): List[S}^{+}\texttt{]} \\
\quad\texttt{def map[U}^{-}\texttt{](f: T}^{+}\texttt{ => U}^{-}\texttt{): List[U}^{+}\texttt{]} \\
\texttt{\}}
\end{array}$$

We Generalize Our Rules for Checking Variance As Follows

- Covariant type parameters (declared with +) are allowed to occur only in positive locations
- Type parameters with no annotation are allowed to be used in all locations
- Contravariant type parameters are allowed to occur only in negative locations

An Example of How We Might Use Contravariant Type Parameters

```
abstract class Function1[-S,+T] {
  def apply(x:S): T
}
```

Map Revisited

```
case object Empty extends List[Nothing] {
  …
  def map[U](f: Nothing => U) = Empty
}
```

Map Revisited

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  …
  def map[U](f: T => U) =
    Cons(f(head), tail.map(f))
}
```

Syntactic Sugar: Currying

- Scala provides special syntax for defining a function that immediately returns another function:

```
def f(x0:T1,…,xN:TN) = (y0:U1,…,yM:UM) => expr
```

can be written as:

```
def f(x0:T1,…,xN:TN) (y0:U1,…,yM:UM) = expr
```

- Defining a function in this way is called “currying” after the computer scientist Haskell Curry

Reduce Revisited

```
abstract class List[+T] {
  …
  def foldLeft[S >: T](x: S)(f: (S, S) => S): S
  def foldRight[S >: T](x: S)(f: (S, S) => S): S
}
```

Note that these functions are curried

Reduce Revisited

```scala
case object Empty extends List[Nothing] {
  …
  def foldLeft[S](x: S)(f: (S, S) => S) = x
  def foldRight[S](x: S)(f: (S, S) => S) = x
}
```

Reduce Revisited

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  …
  def foldLeft[S >: T](x: S)(f: (S, S) => S) =
    tail.foldLeft(f(x, head), f)

  def foldRight[S >: T](x: S)(f: (S, S) => S) =
    f(tail.foldRight(x, f), head)
  }
}
```

Reduce Revisited

```
def foldLeft[S >: T](x: S)(f: (S, S) => S) =
  tail.foldLeft(f(x, head), f)
```

```
Cons(1,Cons(2,Cons(3,Empty))).foldLeft(0)(_+_) ↦
Cons(2,Cons(3,Empty)).foldLeft(0 + 1, _+_) ↦
Cons(2,Cons(3,Empty)).foldLeft(1, _+_) ↦
Cons(3,Empty).foldLeft(1 + 2, _+_) ↦
Cons(3,Empty).foldLeft(3, _+_) ↦
Empty.foldLeft(3 + 3, _+_) ↦
Empty.foldLeft(6, _+_) ↦
6
```

Reduce Revisited

```
  def foldRight[S >: T](x: S)(f: (S, S) => S) =
    f(tail.foldRight(x, f), head)

Cons(1,Cons(2,Cons(3,Empty))).foldRight(0)(_+_) ↦
Cons(2,Cons(3,Empty)).foldRight(0, _+_) + 1 ↦
Cons(3,Empty).foldLeft(0, _+_) + 2 + 1 ↦
Empty.foldLeft(0, _+_) + 3 + 2 + 1 ↦
0 + 3 + 2 + 1 ↦
6
```

Reduce Revisited

```
abstract class List[+T] {
  …
  def reduce[S >: T](f: (S, S) => S): S
}
```

We can elide a zero element for the reduction provided that the list is non-empty

Reduce Revisited

```
case object Empty extends List[Nothing] {
  …
  def reduce[S](f: (S, S) => S) =
    throw ReduceError
}
```

case object ReduceError extends Error

Reduce Revisited

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  …
  def reduce[S >: T](f: (S, S) => S) =
    tail.foldLeft[S](head)(f)
}
```

We explicitly instantiate the type parameter to foldLeft. Without this, type inference will instantiate the type parameter based on the static type of head (which is T) and then signal an error that f is not of type (T, T) => T.

Forall and Exists

```
abstract class List[+T] {
  …
  def forall(p: T => Boolean) =
    map(p).foldLeft(true, _&&_)

  def exists(p: T => Boolean) =
    map(p).foldLeft(false, _||_)
}
```

Length

```
abstract class List[+T] { …
  def length: Int
}
case object Empty extends List[Nothing] { …
  def length = 0
}
case class Cons[+T](head: T, tail: List[T])
extends List[T] { …
  def length = map((_:T) => 1).reduce(_+_)
}
```

In what real contexts could we justify this definition of length?

Pointwise Addition

```
def pointwiseAdd(xs: List[Int], ys: List[Int]): List[Int] = {
  require (xs.length == ys.length)

  (xs, ys) match {
    case (Empty, Empty) => Empty
    case (Cons(x1, xs1), Cons(y1, ys1)) =>
      Cons(x1 + y1, pointwiseAdd(xs1,ys1))
  }
}
```

Generalizing to ZipWith

```scala
// in class List:
def zipWith[U,V](f: (T, U) => V)(that: List[U]): List[V] = {
  require (this.length == that.length)

  (this, that) match {
    case (Empty, Empty) => Empty
    case (Cons(x1,xs1), Cons(y1,ys1)) =>
      Cons(f(x1,y1), xs1.zipWith(f)(ys1))
  }
}
```

Defining The Zip Function

```
// in class List:
def zip[U](that: List[U]) = zipWith((_, _: U))(that)
```

Defining Flatten

```
def flatten[S](xs: List[List[S]]) = {
  xs.foldLeft(Empty)(_++_)
}
```

Defining Flatten

```
def flatten[S](xs: List[List[S]]) = {
  xs.foldLeft(Empty)(_++_)
}
```

Because of the specific type of List needed, we define as a top level function

Defining FlatMap

```
abstract class List[+T] {
  …
  def flatMap[S](f: T => List[S]) =
    flatten(this.map(f))
  }
}
```

Defining FlatMap

```
abstract class List[+T] {
  …
  def flatMap[S](f: T => List[S]) =
    flatten(this.map(f))
  }
}
```

In contrast to flatten, our flatMap function can be defined on arbitrary lists

Defining FlatMap

- These definitions suggest that flatMap is the best thought of as the more primitive notion
- We can define flatMap as a method on lists directly and then define flatten in terms of it

Defining FlatMap

```
abstract class List[+T] { …
  def flatMap[S](f: Nothing => List[S]): List[S]
}

case object Empty extends List[Nothing] { …
  def flatMap[S](f: Nothing => List[S]) = Empty
}

case class Cons[+T](head: T, tail: List[T])
extends List[T] { …
  def flatMap[S](f: T => List[S]) =
    f(head) ++ tail.flatMap(f)
}
```

Defining Filter

```
abstract class List[+T] {
  …
  def filter[U](p: T => Boolean): List[T]
}
```

Defining Filter

```
case object Empty extends List[Nothing] {
  …
  def filter[U](p: T => Boolean) = Empty
}
```

Defining Filter

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  …
  def filter[U](p: T => Boolean) = {
    if (p(head)) Cons(head, tail.filter(p))
    else tail.filter(p)
  }
}
```

For Expressions

For Expressions

- As with all expressions, for expressions reduce to a value
- The value reduced to is a collection
- The type of collection produced depends on the types of collections iterated over
- Each iteration produces a value to include in the resulting collection

Many Maps and Filters Can Be Expressed Using For Expressions

```
for (x <- xs) yield square(x) + 1
```

Many Maps and Filters Can Be Expressed Using For Expressions

```
for (x <- xs) yield square(x) + 1
```

We call this a generator

Many Maps and Filters Can Be Expressed Using For Expressions

```
for clauses yield body
```

Many Maps and Filters Can Be Expressed Using For Expressions

```
for (i <- 1 to 10) yield square(i) + 1
```

Many Maps and Filters Can Be Expressed Using For Expressions

```
for (i <- 0 until 10) yield square(i) + 1
```

Does not include 10

Many Maps and Filters Can Be Expressed Using For Expressions

```
// BAD FORM
for (i <- 0 until xs.length)
  yield square(xs.nth(i)) + 1
```

Many Maps and Filters Can Be Expressed Using For Expressions

```
// Write this instead
for (x <- xs)
  yield square(x) + 1
```

For Expressions Can Also Include Filters

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

This is a filter

Filters in For Expressions

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

For Expressions Can Also Include Filters

```
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

For vs Map

- Compare:

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

- To:

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

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 z 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
```