Comp 311
Functional Programming

Eric Allen, Two Sigma Investments
Robert “Corky” Cartwright, Rice University
Sağnak Taşırlar, Two Sigma Investments
Partially Applied Functions

• If we want to pass a function as an argument, but supply some of the arguments to the function ourselves, we can wrap an application to the function in a function literal:

```javascript
map(x => x + 1, xs)
```
Partially Applied Functions

• If we want to pass a function as an argument, but supply some of the arguments to the function ourselves, we can wrap an application to the function in a function literal:

\[
\text{map}(x \Rightarrow x + 1, \, xs)
\]

which is equivalent to

\[
\text{map}(_ + 1, \, xs)
\]
Partially Applied Functions

- **Eta Expansion**: Wrapping a function in function literal that takes all of the arguments of f and immediately calls f with those arguments

```
(x:Int) => square(x)
```

is equivalent to

```
square
```
Mapping a Computation Over a List

We can use eta expansion to pass operators as arguments:

```
map(x => -x, xs)
```
Mapping a Computation Over a List

We can use eta expansion to pass operators as arguments:

\[
\text{map}(-\_\_, \text{xs})
\]
Returning Functions as Values
We Can Define Functions That Return Other Functions as Values

def add(x: Int): Int => Int = {
  def addX(y: Int) = x + y
  addX
}

We Can Define Functions That Return Other Functions as Values

def add(x: Int): Int => Int = {
    def addX(y: Int) = x + y
    addX
}

The explicit return type is needed because Scala type inference assumes an unapplied function is an error
We Can Define Functions That Return Other Functions as Values

def add(x: Int) = {
  def addX(y: Int) = x + y
  addX _
}

Alternatively, we can eta-expand addX to assure the type checker that we really do intend to return a function
We Can Define Functions That Return Other Functions as Values

```scala
def add(x: Int) = {
  def addX(y: Int) = x + y
  addX _
}
```

An underscore outside of parentheses in a function application denotes the entire tuple of arguments passed to the function.
We Can Define Functions That Return Other Functions as Values

```
def add(x: Int) = x + (_: Int)
```

We can instead define `add` by *partially* eta-expanding the `+` operator. But then we need to annotate the second operand with a type.

We can instead define `add` by *partially* eta-expanding the `+` operator. But then we need to annotate the second operand with a type.
Aside: Type Annotations

• In general, an expression annotated with a type is itself an expression:

\[
\text{expr: Type}
\]

• If the static type of \texttt{expr} is a subtype of \texttt{Type}, then the type of \texttt{expr: Type} is \texttt{Type}
Partial Eta-Expansion

- We can partially eta-expand any function, but we need to annotate the argument types:

```scala
def reduce0 =
  reduce(∅, _: (Int, Int) => Int, _: List)
```
def derivative(f: Double => Double, dx: Double) =
  (x: Double) =>
    (f(x + dx) - f(x)) /
    dx
Derivatives

def f(x: Double) = x * x
def Df = derivative(f, 0.00001)

f(4) ↦ 16
Df(4) ↦ 8.000009999952033
Encapsulating dx

```scala
def D(f: Double => Double) = {
  val dx = 0.00001
  (x: Double) =>
    (f(x + dx) - f(x)) / dx
}
```
Encapsulating dx

```scala
def D(f: Double => Double) = {
  val dx = 0.00001
  (x: Double) =>
    (f(x + dx) - f(x)) / dx
}
```

Our returned function “remembers” these values
Applying a Derivative

```scala
def D(f: Double => Double) = {
  val dx = 0.00001
  (x: Double) =>
    (f(x + dx) - f(x)) / dx
}

D(f)(4) ↦
D((x: Double) => x * x)(4) ↦
```
Applying a Derivative

\[ D((x: \text{Double}) \Rightarrow x \times x)(4) \rightarrow \]

\{val \quad dx = 0.00001
  (x: \text{Double}) \Rightarrow
  ((x: \text{Double}) \Rightarrow x \times x)(x + dx) -
  (x: \text{Double}) \Rightarrow x \times x)(x)) /
  dx \}

\}(4) \rightarrow
Applying a Derivative

\[
\begin{align*}
(x: \text{Double}) & \Rightarrow \\
((x: \text{Double}) & \Rightarrow x \times x)(x + 0.00001) - \\
((x: \text{Double}) & \Rightarrow x \times x)(x)) / \\
0.00001 & \rightarrow \\

((x: \text{Double}) & \Rightarrow x \times x)(4 + 0.00001) - \\
((x: \text{Double}) & \Rightarrow x \times x)(4)) / \\
0.00001 & \rightarrow
\end{align*}
\]

We must be careful to substitute only corresponding occurrences of \(x\)
Applying a Derivative

\[
\frac{(x: \text{Double}) \Rightarrow x \times x)(4 + 0.00001) - (x: \text{Double}) \Rightarrow x \times x)(4)}{0.00001} \mapsto
\]

\[
\frac{(x: \text{Double}) \Rightarrow x \times x)(4.00001) - (x: \text{Double}) \Rightarrow x \times x)(4)}{0.00001} \mapsto
\]

\[
\frac{(4.00001 \times 4.00001) - (4 \times 4)}{0.00001} \mapsto
\]
Applying a Derivative

\[
\frac{(4.00001 \times 4.00001) - (4 \times 4)}{0.00001} \Rightarrow
\]

\[
\frac{16.000080000099995 - 16}{0.00001} \Rightarrow
\]

\[
8.00000999952033E-5 / 0.00001 \Rightarrow
\]

\[
8.00000999952033
\]
Safe Substitution
Applying a Derivative

\[
{(x: \text{Double}) \mapsto \frac{(x: \text{Double}) \Rightarrow x \times x)(x + 0.00001) - (x: \text{Double}) \Rightarrow x \times x)(x)} / 0.00001}(4) \rightarrow
\]

\[
((x: \text{Double}) \Rightarrow x \times x)(4 + 0.00001) - (x: \text{Double}) \Rightarrow x \times x)(4)) / 0.00001
\]

In cases like this one, we can avoid accidental variable capture by selective renaming.
Safe Substitution
(a.k.a. Alpha Renaming)

• We can ensure we never accidentally substitute the wrong parameters by automatically renaming constants, functions, and parameters with fresh names

• A fresh name must not capture a name referred to in the scope of a parameter

• A fresh name must not be captured by a name in an enclosing scope
Applying a Derivative

\{(x: Double) =>
   ((y: Double) => y * y)(x + 0.00001) -
   (z: Double) => z * z)(x)) /
   0.00001\}(4) \mapsto

((y: Double) => y * y)(4 + 0.00001) -
   (z: Double) => z * z)(4)) /
   0.00001
Function Equivalence

• Now we have seen the three forms of function equivalence stipulated by the Lambda Calculus:

  • Alpha Renaming: Changing the names of a function’s parameters does not affect the meaning of the function

  • Beta Reduction: To apply a function to an argument, reduce to the body of the function, substituting occurrences of the parameter with the corresponding argument

  • Eta Equivalence: Two functions are equivalent iff they are *extensionally equivalent*: They give the same results for all arguments
Parametric Types
Parametric Types

• We have defined two forms of lists: lists of ints and lists of shapes

• Many computations useful for one are useful for the other:
  • Map, reduce, filter, etc.

• It would be better to define lists and their operations once for all of these cases
Parametric Types

• Higher-order functions take functions as arguments and return functions as results

• Likewise, *parametric types*, a.k.a., *generic types*, takes types as arguments and return types as results
Parametric Lists

• Every application of this parametric type to an argument yields a new type:

```scala
abstract class List[T] {
  def ++(ys: List[T]): List[T]
}
```
Parametric Lists

• Every application of this parametric type to an argument yields a new type:

```scala
abstract class List[T <: Any] {
  def ++(ys: List[T]): List[T]
}
```

• We augment the declarations of type parameters to permit an upper bound on all instantiations of a parameter

• By default, the bound is Any
Syntax of Parametric Class Definitions

```java
<modifiers> class C[T1 <: N,..,TN <: N] extends N {
    <ordinary class body>
}
```

- We denote “naked” type parameters as $T_1$, $T_2$, etc.
- We denote all other types with $N$, $M$, etc.
Syntax of Parametric Class Definitions

<modifiers> class C[T1 <: N,..,TN <: N] extends N {
    <ordinary class body>
}

- Declared type parameters T1, ..., TN are in scope throughout the entire class definition, including:
  - The bounds of type parameters
  - The extends clause
  - Object definitions must not be parametric
Parametric Lists

- Every application of this parametric type yields a new type:

  List[Int]
  List[String]
  List[List[Double]]
  etc.
Parametric Lists

• Every application (a.k.a., instantiation) of this parametric type yields a new type:

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

Note that our parametric type can be instantiated with type parameters, including its own!
Parametric Lists

case class Empty[S]() extends List[S] {
  def ++(ys: List[S]) = ys
}

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

```scala
case class Empty[S]() extends List[S] {
  def ++(ys: List[S]) = ys
}

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

Our definition requires a separate type `Empty[S]` for every instantiation of `S`. Thus we must define `Empty` as a class rather than an object.
Type Environments

• To explain how to type check expressions in the context of parametric types, we must introduce the notion of *environments*

• We define a type parameter environment to hold a collection of zero or more type parameter declarations with their bounds

• Type environments can be extended with more declarations
Type Checking a Class Definition

• To type check a parametric class definition:

  • Check the declarations of the class in a new type parameter environment that extends the enclosing environment with all its type parameters
Type Checking a Function Definition

- To type check a function definition in environment E:
  - Check that the types of all parameters are well-formed
  - Find the type of the body of the function, substituting occurrences of parameters with their types
  - Ensure that the type of the body is a subtype of the declared return type (in environment E)
Well-Formedness of Types

• A type is well-formed in environment E iff:
  • If it is a well-defined non-parametric type
  • It is a type parameter T in environment E
  • It is an instantiation of a defined parametric type and:
    • All of its type arguments are well-formed types in E
    • All of its type arguments respect the bounds on their corresponding type parameters
Subtyping With Environments

• It is non-sensical to compare types in separate type environments:

```scala
case class Empty[S]() extends List[S] {
  def ++(ys: List[S]) = ys
}

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

• Is $S$ a subtype of $T$?
Subtyping With Environments

• We must modify our subtyping rules to refer to an environment E:

  • $S <: S$ in E

  • If $S <: T$ in E and $T <: U$ in E then $S <: U$ in E
Subtyping With Environments

- If:
  - class $C[T_1, \ldots, T_N]$ extends $D[U_1, \ldots, U_M]$
  - and $X_1, \ldots, X_N$ are well-formed in $E$
  - then $C[X_1, \ldots, X_N] <: D[U_1, \ldots, U_M][T_1 \mapsto X_1, \ldots, T_N \mapsto X_N]$ in $E$
Subtyping With Environments

• If:

  • class C[T1,..,TN] extends D[U1,...UM]

  • and X1,…,XN are well-formed in E

  • then C[X1,…XN] <: D[U1,...,UM][T1↦X1,...,TN↦XN] in E

We use this notation to indicate safe substitution of T1 for X1, … TN for XN in D[U1,...,UM]
Covariance

• Can one instantiation of a parametric type be a subtype of another?

• Currently our rules allow this only in the reflexive case:

\[
\text{List[Int]} <: \text{List[Int]} \text{ in } E
\]
Covariance

• It would be useful to allow some instantiations to be subtypes of another

• For example, we would like it to be the case that:

$$\text{List[Int]} <: \text{List[Any]}$$
Covariance

• In general, we say that a parametric type $C$ is covariant with respect to its type parameter $S$ if:

$$S <: T \text{ in } E$$

implies

$$C[S] <: C[T] \text{ in } E$$

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

- For a parametric type such as:

  ```scala
  abstract class List[T <: Any] {
    def +(ys: List[T]): List[T]
  }
  ```

- And types $S$ and $T$, such that $S <: T$ in some environment $E$:

  - What must we check about the body of class `List` to allow for `List[S] <: List[T]` in $E$?
Covariance

• Consider instantiations for types \texttt{String} and \texttt{Any}:

abstract class List[\texttt{Any}] {
    def ++(ys: List[\texttt{Any}]): List[\texttt{Any}]
}
abstract class List[\texttt{String}] {
    def ++(ys: List[\texttt{String}]): List[\texttt{String}]
}
Covariance

• If these were ordinary classes connected by an extends class:

  • We would need to ensure that the overriding definition of ++ in class List[String] was compatible with the overridden definition in List[Any]
Covariance

abstract class List[Any] {
    def ++(ys: List[Any]): List[Any]
}
abstract class List[String] extends List[Any] {
    def ++(ys: List[String]): List[String]
}
Covariance

abstract class List[Any] {
    def ++(ys: List[Any]): List[Any]
}
abstract class List[String] extends List[Any] {
    def ++(ys: List[String]): List[String]
}

But if List[String] <: List[Any] in E
then this is not a valid override
abstract class List[Any] {
    def ++(ys: List[Any]): List[Any]
}
abstract class List[String] extends List[Any] {
    def ++(ys: List[String]): List[String]
}

On the other hand, the return types are not problematic
Covariance

• From our example, we can glean the following rule:

  • We allow a parametric class $C$ to be covariant with respect to a type parameter $T$ so long as $T$ does not appear in the types of the method parameters of $C$. 
Covariance

abstract class List[+T] {}

• We stipulate that a parametric type is covariant in a parameter T by prefixing a + at the definition of T

• (We will return to our definition of append later)
Covariance

case object Empty extends List[Nothing] {
}

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

```scala
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:

\[ T <: S \]

• They can also include lower bounds:

\[ T >: U \]

• Or they can include both:

\[ T >: S <: U \]
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
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)
}
abstract class List[+T] {
    ...
    def map[U](f: T => U): List[U]
}
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:

  • $(\text{String} \Rightarrow U) \, >: \, (\text{Any} \Rightarrow U)$ in $E$ provided:
    • $\text{String} <: \, \text{Any}$ in $E$
    • $U <: \, U$ in $E$

• So subtype $\text{List}[	ext{String}] <: \text{List}[	ext{Any}]$ 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
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:

$$S <: T \text{ in } E$$

implies

$$C[T] <: C[S] \text{ in } E$$

• 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
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.

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