Comp 311
Functional Programming

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

October 19, 2017

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
```

Lexical vs Dynamic Scoping

- The semantics of function application that we have outlined is referred to as *lexical scoping*
- Early versions of Lisp avoided the need for closures:
    - They reduced function applications by extending the environment in which the application *occurred*
        - This semantics of function application is known as dynamic scoping
        - Why is dynamic scoping problematic?

Call-By-Value
and
Call-By-Name

Call-By-Value

- Thus far, the evaluation semantics we have studied (both with the substitution and environment models) is known as call-by-value:
  - To evaluate a function application, we first evaluate the arguments and then evaluate the function body

Call-By-Value

- We have seen several “special forms” where this evaluation semantics is not what we want:

`&&`   `||`   `if-else`

Call-By-Value

- We could delay evaluation in these cases by wrapping arguments in function literals that take no parameters

```
def myOr(left: Boolean, right: () => Boolean) =
  if (left) true
  else right()
```

Call-By-Value

- We could delay evaluation in these cases by wrapping arguments in function literals that take no parameters

```
myOr(true, () => 1/0 == 2) ↦ true
```

- Functions that take no arguments are referred to as *thunks*

Call-By-Name

- Scala provides a way that we can pass arguments as thunks without having to wrap them explicitly

```
def myOr(left: Boolean, right: => Boolean) =
  if (left) true
  else right
```

We simply leave off the parentheses in the parameter's type

Call-By-Name

- Now we can call our function without wrapping the second argument in an explicit thunk:

```
myOr(true, 1/0 == 2) ↦ true
```

- The thunk is applied (to nothing) the first time that the argument is evaluated in a function

Call-By-Name

- We can use by-name parameters to define new *control abstractions:*

```
def myAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

Syntactic Sugar: Braces for Passing Arguments

- Any function that takes a single argument can be applied by passing the argument enclosed in braces instead of parentheses

```
myAssert {
  2 + 2 == 4
}
```

Syntactic Sugar: Braces for Passing Arguments

- Any function that takes a single argument can be applied by passing the argument enclosed in braces instead of parentheses

```
myAssert {
  def double(n: Int) = 2 * n
  double(2) == 4
}
```

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

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

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:

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

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

Scala Immutable Collections

Immutable Lists

- Behave much like the lists we have defined in class
- Lists are covariant
- The empty list is written `Nil`
- Nil extends `List[Nothing]`

Immutable Lists

- The list constructor takes a variable number of arguments:

```
List(1,2,3,4,5,6)
```

Immutable Lists

- Non-empty lists are built from Nil and Cons (written as the right-associative operator `::`)

```
1 :: 2 :: 3 :: 4 :: Nil
```

List Operations

- `head` returns the first element
- `tail` returns a list of elements but the first
- `isEmpty` returns true if the list is empty
- Many of the methods we have defined are available on the built-in lists

FoldLeft and FoldRight Written as Operators

- foldLeft:

```
(zero /: xs)(op)
```

- foldRight:

```
(xs :\ zero)(op)
```

FoldLeft and FoldRight Written as Operators

- foldLeft:

```
(xs foldLeft zero)(op)
```

- foldRight:

```
(xs foldRight zero)(op)
```

FoldLeft and FoldRight Written as Methods

- foldLeft:

```
xs.foldLeft(zero) { op }
```

- foldRight:

```
xs.foldRight(zero) { op }
```

SortWith

```
List(1,2,3,4,5,6) sortWith (_ > _)
↦
List(6, 5, 4, 3, 2, 1)
```

Range

```
List.range(1,5)
↦
List(1, 2, 3, 4)
```

Using Fill for Uniform Lists

```
List.fill(10)(0) ↦
List(0,0,0,0,0,0,0,0,0,0)
```

Using Fill for Uniform Lists

```
List.fill(3,3)(0) ↦

List(List(0,0,0),
     List(0,0,0),
     List(0,0,0))
```

Tabulating Lists

```
List.tabulate(3,3) { (m,n) =>
  if (m == n) 1 else 0
}
↦
List(List(1,0,0),
     List(0,1,0),
     List(0,0,1))
```

Immutable Sets

Immutable Sets

- Sets are unordered, unrepeated collections of elements
- Set[T] extends the function type T ⇒ Boolean
- Sets are parametric and *invariant* in their element type

Set Factory

```
Set(1,2,3,4,5)
```

Set Element Addition

```
Set(1,2,3) + 4 ↦
  Set(1,2,3,4)
```

Set Element Subtraction

```
Set(1,2,3) - 2 ↦
Set(1,3)
```

```
Set(1,2,3) - 4 ↦
Set(1,2,3)
```

Set Union

```
Set(1,2,3) ++ Set(2,4,5) ↦
Set(1,2,3,4,5)
```

Set Difference

```
Set(1,2,3) -- Set(2,4,5,3) ↦
Set(1)
```

Set Intersection

```
Set(1,2,3) & Set(2,4,5,3) ↦
Set(2,3)
```

Set Cardinality

```
Set(1,2,3).size ↦
3
```

Set Membership

```
Set(1,2,3).contains(2) ↦
true
```

```
Set(1,2,3)(2) ↦
true
```

The *apply* method on sets is equivalent to the *contains* method.

Immutable Maps

Immutable Maps

- Maps are collections of key/value pairs
- They are parametric in both the key and value type
    - Invariant in their key type
    - Covariant in their value type

The -> Operator

- The infix operator `->` returns a pair of its arguments:

```
1 -> 2
↦
(1,2)
```

- Note: Scala also allows *Unicode Operators*, and the infix "⟶" operator is one such example:

```
1 → 2
↦
(1,2)
```

The → Operator is Left Associative

```
> 1 → 2 → 3 → 4
res8: (((Int, Int), Int), Int) = (((1,2),3),4)
```

The Map Factory

```
Map("a" → 1, "b" → 2, "c" → 3)
                ↦
  Map(a -> 1, b -> 2, c -> 3)
```

Map Addition

```
Map("a" → 1, "b" → 2, "c" → 3) + ("d" → 4)
↦
Map(a -> 1, b -> 2, c -> 3, d -> 4)
```

Map Operations

The operators/methods are defined in the expected way:

- `-`
- `++`
- `--`
- `size`

Map Membership

```
Map("a" → 1, "b" → 2, "c" → 3).contains("b")
↦
true
```

Map Lookup

```
Map("a" → 1, "b" → 2, "c" → 3)("c")
                ↦
                3
```

Map Keys

```
Map("a" → 1, "b" → 2, "c" → 3).keys
↦
Set(a, b, c): Iterable[String]
```

```
Map("a" → 1, "b" → 2, "c" → 3).keySet
↦
Set(a, b, c): Set[String]
```

Map Values

```
Map("a" → 1, "b" → 2, "c" → 3).values
↦
Set(1,2,3)
```

Map Empty

```
Map("a" → 1, "b" → 2, "c" → 3).isEmpty
↦
false
```

Traits

Traits

Traits provide a way to factor out common behavior among multiple classes and mix it in where appropriate

Trait Definitions

Syntactically, a trait definition looks like an abstract class definition, but with the keyword “trait”:

```
trait Echo {
  def echo(message: String) =
    message
}
```

Trait Definitions

- Traits can declare fields and full method definitions
- They must not include constructors

```
trait Echo {
  val language = “Portuguese"
  def echo(message: String) =
    message
}
```

Using Traits

- Classes "mix in" traits using either the `extends` or `with` keywords

```
class Parrot extends Echo {
  def fly() = {
    // forget to fly and talk instead
    echo("poly wants a cracker")
  }
}
```

Using Traits

- Classes "mix in" traits using either the `extends` or `with` keywords

```
class Parrot extends Bird with Echo {
  def fly() = {
    // forget to fly and talk instead
    echo("poly wants a cracker")
  }
}
```

Using Traits

- Classes "mix in" traits using either the `extends` or `with` keywords

```
trait Smart {
  def somethingClever() =
    "better a witty fool than a foolish wit"
}
```

Using Traits

- Classes can mix in multiple traits via multiple `with`s:

```
class Parrot extends Bird with Echo
with Smart {
  def fly() = {
    // forget to fly and talk instead
    echo(somethingClever())
  }
}
```

Using Traits

Classes can mix in multiple traits via multiple `with`s:

```
trait X
case class Foo()

new Foo() with X
```

*Must use the **`new`** keyword when creating a new class instance with a mixin trait*