Lecture 4
Types in Scala
Instructor: Hossein Hojjat

September 6, 2017
There are no statements in Scala: everything must have some value

Value can be of type `Unit`
- A type that allows only one value: `()`
- Indicates no interesting value (similar to Java’s void type)

Type of expression automatically inferred (or can make it explicit)

```scala
scala> var x = 5
x: Int = 5

scala> val r = {val y = 6; y/x}
r: Int = 1

scala> val i = println("hello")
hello
i: Unit = ()
```
### Conditional Expression

- **if (condition) expression [else if (condition) expression] [else expression]**

- Condition is any expression that returns a Boolean

```scala
scala> val x = -6
x: Int = -6

scala> val abs_x = if (x < 0) -x else x
abs_x: Int = 6
```
Conditional Expression

if (condition) expression [else if (condition) expression] [else expression]

- What happens if the else expression is missing?
- Type of the else expression is unspecified so the compiler uses Unit
- Either Int or Unit can be returned, so the compiler choose the root type AnyVal

```scala
scala> val x = -6
x: Int = -6

scala> val abs = if (x < 0) -x
abs: AnyVal = 6
```
Scala Class Hierarchy

scala.Any

- scala.AnyVal
  - scala.Double
  - scala.Float
  - scala.Long
  - scala.Int
  - scala.Short
  - scala.Byte
  - scala.Char
- scala.AnyRef
  - scala.ScalaObject
    - java.lang.Object
- scala.Seq
  - scala.List
- scala.Option

Implicit Conversion

Subtype
• while-loops usually indicate imperative programming style
• They manipulate the content of some mutable variables in each step
• Result type of while is Unit

```
scala> var x = 0
x: Int = 0

scala> while (x < 5) {x = x+1; println(x)}
1
2
3
4
5
```
Function Type

- Type of \texttt{inc} is \texttt{Int => Int}
- When applied to an integer produces an integer
- \texttt{Int => Int} is the set of functions, each of which when applied to an integer produces an integer

\begin{verbatim}
scala> def inc: Int => Int = (_ + 1)
inc: Int => Int

scala> def add: (Int, Int) => Int = (_ + _)
add: (Int, Int) => Int
\end{verbatim}
Currying

- Transform a function that takes multiple parameters into a chain of single parameter functions

```scala
scala> def cat = (s1: String, s2: String) => s1 + s2
cat: (String, String) => String

scala> def catCurried = (s1: String) => (s2: String) => s1 + s2
catCurried: String => (String => String)

scala> cat("Programming", "Languages")
res0: String = Programming Languages

scala> catCurried("Programming")("Languages")
res1: String = Programming Languages
```
What is the most “general type” for `double`?

```scala
def double = (x: Int) => x + x
double: ?
```
What is the most “general type” for `double`?

```scala
def double = (x: Int) => x + x
double: ?
```

**Step 1: Assign preliminary types**

<table>
<thead>
<tr>
<th>Sub-expression</th>
<th>Preliminary Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double</code></td>
<td><code>A</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td><code>Int</code></td>
</tr>
<tr>
<td><code>x + x</code></td>
<td><code>B</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>\forall \alpha. \alpha \Rightarrow \alpha \Rightarrow \alpha</code> (with simplification)</td>
</tr>
</tbody>
</table>
What is the most “general type” for `double`?

```scala
def double = (x: Int) => x + x
double: ?
```

**Step 1: Assign preliminary types**

<table>
<thead>
<tr>
<th>Sub-expression</th>
<th>Preliminary Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double</code></td>
<td><code>A</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td><code>Int</code></td>
</tr>
<tr>
<td><code>x + x</code></td>
<td><code>B</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td>(\forall \alpha. \alpha \Rightarrow \alpha \Rightarrow \alpha) (with simplification)</td>
</tr>
</tbody>
</table>

**Step 2: Generate constraints**

\[
\alpha_1 \Rightarrow \alpha_1 \Rightarrow \alpha_1 = \text{Int} \Rightarrow \text{Int} \Rightarrow B
\]

\[
A = \text{Int} \Rightarrow B
\]
What is the most “general type” for `double`?

```scala
def double = (x: Int) => x + x
double: Int => Int
```

**Step 1: Assign preliminary types**

<table>
<thead>
<tr>
<th>Sub-expression</th>
<th>Preliminary Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double</code></td>
<td><code>A</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td><code>Int</code></td>
</tr>
<tr>
<td><code>x + x</code></td>
<td><code>B</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>∀α. α ⇒ α ⇒ α</code> (with simplification)</td>
</tr>
</tbody>
</table>

**Step 2: Generate constraints**

\[
\alpha_1 \Rightarrow \alpha_1 \Rightarrow \alpha_1 = \text{Int} \Rightarrow \text{Int} \Rightarrow B
\]

\[
A = \text{Int} \Rightarrow B
\]

**Step 3: Solve constraints**

\[
A = \text{Int} \Rightarrow \text{Int}, \quad B = \text{Int}
\]
def f = (x: Boolean) => (if (x) 1 else 0)
f: ?
def f = (x: Boolean) => (if (x) 1 else 0)
f:  Boolean => Int

**Step 1: Assign preliminary types**

<table>
<thead>
<tr>
<th>Sub-expression</th>
<th>Preliminary Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>A</td>
</tr>
<tr>
<td>x</td>
<td>Boolean</td>
</tr>
<tr>
<td>(if (x) 1 else 0)</td>
<td>B</td>
</tr>
<tr>
<td>if</td>
<td>( \forall \alpha. \text{Boolean} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha )</td>
</tr>
<tr>
<td>0, 1</td>
<td>Int</td>
</tr>
</tbody>
</table>

**Step 2: Generate constraints**

\[
\text{Boolean} \Rightarrow \alpha_1 \Rightarrow \alpha_1 \Rightarrow \alpha_1 = \text{Boolean} \Rightarrow \text{Int} \Rightarrow \text{Int} \Rightarrow B \\
A = \text{Boolean} \Rightarrow B
\]

**Step 3: Solve constraints**

\[
A = \text{Boolean} \Rightarrow \text{Int} , \ B = \text{Int}
\]
def twice(f, x) = f(f(x))

- Note: This function does not type check in Scala
- It can be type checked in a language with Hindly/Milner type system
  - (e.g. Haskell)
Exercise

def twice(f,x) = f(f(x))

• Note: This function does not type check in Scala
• It can be type checked in a language with Hindly/Milner type system
  - (e.g. Haskell)

Answer.

twice : (a => a) => a => a
• Allows building new data type by combining other types

abstract class Bool
case object True extends Bool
case object False extends Bool

case class Cons(hd: Int, tl: List) extends List
Algebraic Data Types

- Why **algebraic**?
- "Algebraic structure" is a set closed under one or more operations
  - (e.g., multiplication, addition)
- An algebraic data type is built from products and sums

```scala
// product type: A has a B and C
case class A(b: B, c: C)

// sum type: A is a B or C
abstract class A
case class B() extends A
case class C() extends A
```
Pattern Matching

- Check a given abstract datatype for an exact pattern
- Similar to Java’s Switch/Case statement

```scala
Cons(1, Cons(2, Cons(3, Nil))) match {
  case Cons(1, tail) => println("one")
  case Cons(2, tail) => println("two")
}
```
Exercise

• Write a function reverse that gets a list of integers and returns the reverse of the list

abstract class List
case object Nil extends List
case class Cons(hd: Int, tl: List) extends List
• Write a function `reverse` that gets a list of integers and returns the reverse of the list.

```scala
abstract class List
case object Nil extends List
case class Cons(hd: Int, tl: List) extends List

def reverse(l: List): List = {
  def _rev(res: List, rem: List): List = rem match {
    case Nil => res
    case Cons(hd, tl) => _rev(Cons(hd, res), tl)
  }
  _rev(Nil, l)
}
```