CSCI 740 - Programming Language Theory

Lecture 4

Types in Scala

Instructor: Hossein Hojjat

September 6, 2017

Types

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

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

Conditional Expression

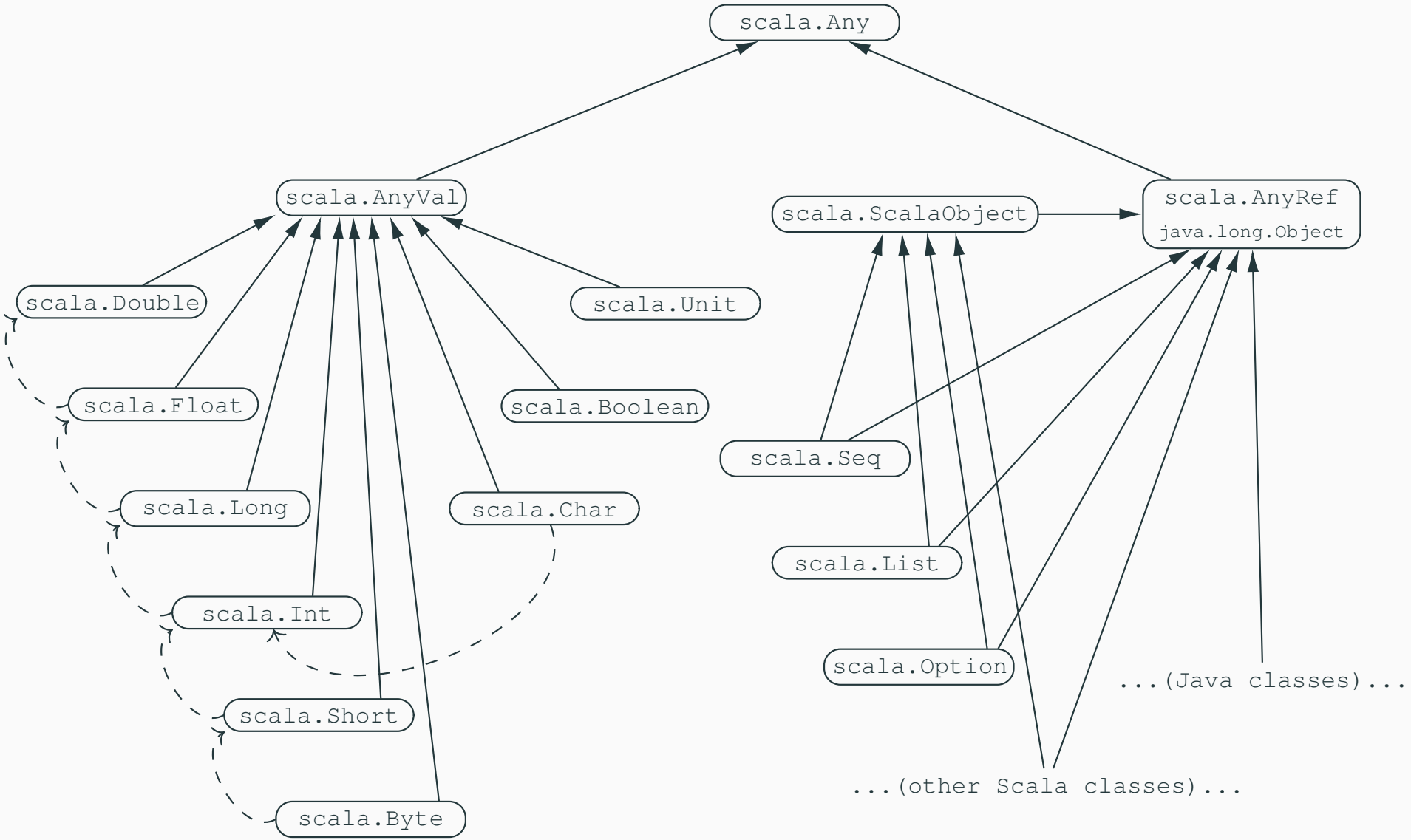
if (*condition*) *expression* [**else if** (*condition*) *expression*] [**else** *expression*]

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

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

Scala Class Hierarchy



while-loop

- `while`-loops usually indicate imperative programing 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 `inc` is `Int => Int`
- When applied to an integer produces an integer
- `Int => Int` is the set of functions, each of which when applied to an integer produces an integer

```
scala> def inc: Int => Int = (_ + 1)
inc: Int => Int

scala> def add: (Int,Int) => Int = (_ + _)
add: (Int, Int) => Int
```

Currying

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

```
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
```

Type Inference - Example 1

What is the most "general type" for `double`?

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

Type Inference - Example 1

What is the most "general type" for `double`?

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

Step 1: Assign preliminary types

Sub-expression	Preliminary Type
`double`	A
`x`	`Int`
`x + x`	B
`+`	$\forall\alpha.\ \alpha \Rightarrow \alpha \Rightarrow \alpha$
	(with simplification)

Type Inference - Example 1

What is the most "general type" for `double`?

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

Step 1: Assign preliminary types

Sub-expression	Preliminary Type
`double`	A
`x`	`Int`
`x + x`	B
`+`	$\forall\alpha.\ \alpha \Rightarrow \alpha \Rightarrow \alpha$
	(with simplification)

Step 2: Generate constraints

$$\begin{aligned} \alpha_1 \Rightarrow \alpha_1 \Rightarrow \alpha_1 &= \texttt{Int} \Rightarrow \texttt{Int} \Rightarrow B \\ A &= \texttt{Int} \Rightarrow B \end{aligned}$$

Type Inference - Example 1

What is the most "general type" for `double`?

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

Step 1: Assign preliminary types

Sub-expression	Preliminary Type
`double`	A
`x`	`Int`
`x + x`	B
`+`	$\forall\alpha.\ \alpha \Rightarrow \alpha \Rightarrow \alpha$
	(with simplification)

Step 2: Generate constraints

$$\alpha_1 \Rightarrow \alpha_1 \Rightarrow \alpha_1 \quad = \quad \texttt{Int} \Rightarrow \texttt{Int} \Rightarrow B$$

$$A \quad = \quad \texttt{Int} \Rightarrow B$$

Step 3: Solve constraints

$$A = \texttt{Int} \Rightarrow \texttt{Int}\ ,\ B = \texttt{Int}$$

Type Inference - Example 2

```
def f = (x: Boolean) => (if (x) 1 else 0)
f: ?
```

Type Inference - Example 2

```
def f = (x: Boolean) => (if (x) 1 else 0)
f:  Boolean => Int
```

Step 1: Assign preliminary types

Sub-expression	Preliminary Type
`f`	A
`x`	`Boolean`
`(if (x) 1 else 0)`	B
`if`	$\forall\alpha.\ \texttt{Boolean} \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha$
`0 , 1`	`Int`

Step 2: Generate constraints

$$\texttt{Boolean} \Rightarrow \alpha_1 \Rightarrow \alpha_1 \Rightarrow \alpha_1 \quad = \quad \texttt{Boolean} \Rightarrow \texttt{Int} \Rightarrow \texttt{Int} \Rightarrow B$$

$$A \quad = \quad \texttt{Boolean} \Rightarrow B$$

Step 3: Solve constraints

$$A = \texttt{Boolean} \Rightarrow \texttt{Int}\ ,\ B = \texttt{Int}$$

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)

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
```

Algebraic Data Types

- Allows building new data type by combining other types

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

```
abstract class List
case object Nil extends List
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

```
//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

```
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
```

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
```

```
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)
}
```