@AndrzejWasowski
Andrzej Wąsowski

@zhoulaifu
Zhoulai Fu

Advanced Programming

1. Functional Programming In A Nutshell

IT UNIVERSITY OF COPENHAGEN

SOFTWARE QUALITY RESEARCH

Basics of Scala

A singleton class and its only instance

object creates a name space; used to build modules. Access the namespace with navigation: `MyModule.abs(42)`

```
1 object MyModule {
2
3   def abs(n: Int): Int = if (n < 0) -n else n
4
5   private def formatAbs(x: Int) =
6     s"The absolute value of $x is ${abs (x)}"
7
8   val magic :Int = 42
9   var result :Option[Int] = None
10
11  def main(args: Array[String]): Unit = {
12    assert (magic - 84 == magic.-(84))
13    println (formatAbs (magic-100))
14  }
15 }
```

def Defines a function (l.3)

A body **expression** (statements secondary in Scala)

Use braces if more expressions needed.

A named **value** declaration (final, immutable). Use this a lot.

A **variable** declaration. Avoid if possible.

Instantiation of a generic type

`None` is a singleton "constructor". Construct case classes without **new**

Operators are functions, can be overloaded:
minus is `Int.-(Int) :Int`
Unary methods can be used infix: `MyModule abs -42` legal

Every value is an object

Line 6 shows an **interpolated** character string

The example adapted from [Chiusano, Bjarnason 2015]

Exercises 1–2

Pure Functions

Def. Referentially transparent expression (e)

Expression e is RT iff replacing e by its value in programs does not change their semantics

(Java) append an element to a list

`a.add(5)` // non RT

value `void`; substitution is pointless; the meaning is in the references reachable from `a` (change over time for the same `a`)

(Scala) append to an immutable list

`val b =Cons(5,a)` // RT

The value is a list `b`, identical to `a`, modulo the added head element

Def. Pure function (f)

Iff every expression $f(x)$ is referentially transparent for all referentially transparent expressions x. Otherwise **impure** or **effectful**.

In practice: **A function is pure if it does not have side effects** (writes/reads variables, files or other streams, modifies data structures in place, sets object fields, throws exceptions, halts with errors, draws on screen)

Pure code shows dependencies in interface, good for mocking, testable

Loops and Recursion

An imperative factorial

```
def factorial (n :Int) :Int = {
  var result = 1
  for (i <- 2 to n)
    result *= i
  return result
}
```

Loops compute with effects;
cannot be used in pure code

Tail recursive, pure factorial

```
def factorial (n :Int) = {
  def f (n :Int, r :Int) :Int =
    if (n<=1) r
    else f (n-1, n*r)
  f (n,1)
}
```

call in tail position

Call tails are automatically compiled to loops with O(1) space overhead

A pure recursive factorial

```
def factorial (n :Int) :Int =
  if (n<=1) 1
  else n * factorial (n-1)
```

call not in tail position

Example execution

$\texttt{factorial(5)}$
$\rightsquigarrow 5 * (\texttt{factorial(4)})$
$\rightsquigarrow 5 * (4 * (\texttt{factorial(3)}))$
$\rightsquigarrow 5 * (4 * (3 * (\texttt{factorial(2)})))$
$\rightsquigarrow 5 * (4 * (3 * (2 * (\texttt{factorial(1)}))))$
$\rightsquigarrow 5 * (4 * (3 * (2 * 1)))$
$\rightsquigarrow 5 * (4 * (3 * 2))$
$\rightsquigarrow 5 * (4 * 6)$
$\rightsquigarrow 5 * 24$
$\rightsquigarrow 120$

Uses O(n) stack space;
Technically exponential
(for this example)!

Def. Call in **tail position**

The caller immediately returns the value of the call

Exercise 3

Algebraic Data Types (ADTs)

Def. Algebraic Data Type

A type generated by one or several constructors, each of which may contain zero or more arguments.

Sets generated by constructors are **summed**, each constructor is a **product** of its arguments; thus **algebraic**.

Example: immutable lists

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head :A, tail :List[A]) extends List[A]
```

sealed: extensible in the same file only

`Nothing`: **subtype of any type**

Example: operations on lists

```
object List {
  def sum(ints :List[Int]) :Int =
    ints match { case Nil => 0
                 case Cons(x,xs) => x + sum(xs) }
  def apply[A](as :A*): List[A] =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

companion object of `List[+A]`

pattern matching uses case class constructors

overload function application for the object

variadic function

Function Values

- In functional programing **functions are values**
- Functions can be **passed to other functions**, composed, etc.
- Functions operating on function values are **higher order** (HOFs)

```
def map (a :List[Int]) (f :Int => Int) :List[Int] =
  a match { case Nil     => Nil
            case h::tail => f(h)::map (tail) (f) }
```

A functional (pure) example

```
val mixed = List(-1, 2, -3, 4)
map (mixed) (abs _)
```

```
map (mixed) ((factorial _) compose (abs _))
```

see method `factorial` as a function value

alternatively type it explicitly:

`(abs :Int => Int)`

An imperative (impure) example

```
val mixed = Array (-1, 2, -3, 4)
for (i <- 0 until mixed.length)
  mixed(i) = abs (mixed(i))
```

```
val mixed1 = Array (-1, 2, -3, 4)
for (i <- 0 until mixed1.length)
  mixed1(i) = factorial(abs(mixed1(i)))
```

© Andrzej Wąsowski, IT University of Copenhagen

Anonymous Functions

Literals

```
val l =List(1, -2, 3)
val a =Array(-1, 2, -3)
```

Function Literals (Anonymous Functions)

We need the same for functions

```
val negative =(x :Int) =>x < 0
negative (-42) ⇝ true
```

Use to create functions in place:

```
l.filter ((x :Int) =>x < 0) ⇝ ?
a.filter ((x :Int) =>x > 0) ⇝ ?
```

Alternative concise syntax

```
(abs _) ⇝ (x :Int) =>MyModule.abs x
```

Scala distinguishes functions and methods.

We used this syntax before to turn a method into a function (like above).

Currying and partial application

```
val add2 =(x :Int, y :Int) =>x + y
val add =(x :Int) =>(y :Int) =>x + y
```

a curried function

What is the type of `add`? What is the value of `add (2) (3) ⇝` ?

Curried functions can be partially applied: `val incr =add (1)`

a partial application

Type of `incr`? Value of `incr (7) ⇝` ?

Methods can also be curried: `def add (x:Int) (y:Int) :Int =x + y`

Parametric Polymorphism

Monomorphic functions operate on fixed types:

A monomorphic map in Scala

```
def map (a :List[Int]) (f :Int => Int) :List[Int] =
  a match { case Nil      => Nil
            case h::tail => f(h)::map (tail) (f) }
```

There is nothing specific here regarding Int.

A polymorphic map in Scala

```
def map[A,B] (a :List[A]) (f :A => B) :List[B] =
  a match { case Nil      => Nil
            case h::tail => f(h)::map (tail) (f) }
```

An example of use (type parameters are inferred):

```
map[Int,String] (mixed_list) { _.toString } compose
  (factorial _) compose (abs _))
```

- A **polymorphic** function operates on values of (m)any types (some restriction possible in Scala)
- A polymorphic type defines a parameterized family of types
- Don't confuse with OO-polymorphism roughly equal to "dynamic method dispatch" (dependent on the inheritance hierarchy)

HOFs in Scala Standard Library

Methods of class `List[A]`, operate on `this` list, type `A` is bound in the class

`map[B](f: A =>B): List[B]`
Translates `this` list of `A`s into a list of `B`s using `f` to convert the values

`filter(p: A =>Boolean): List[A]`
Compute a sublist of `this` by selecting the elements satisfying the predicate `p`

`flatMap[B](f: A =>List[B]): List[B]` *type slightly simplified
Builds a new list by applying `f` to elements of `this`, concatenating results.

`take(n: Int): List[A]`
Selects first n elements.

`takeWhile(p: A =>Boolean): List[A]`
Takes longest prefix of elements that satisfy a predicate.

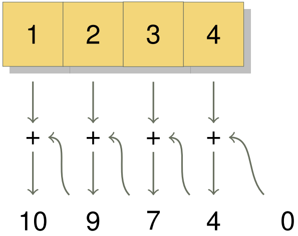
`forall(p: A =>Boolean): Boolean`
Tests whether a predicate holds for all elements of this sequence.

`exists(p: A =>Boolean): Boolean`
Tests whether a predicate holds for some of the elements of this sequence.

More at `http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List`

[Right]Folding: Functional Loops

Compute a sum of list's elements



What characterizes similar computations?

- An **input list** `l = List(1,2,3,4)`
- An **initial value** `z = 0`
- A **binary operation** `f :Int => Int = _ + _`
- An **iteration algorithm** (folding)

```
def foldRight[A,B] (f : (A,B) => B) (z :B) (l :List[A]) :B =
  l match {
    case Cons(x,xs) => f(x, foldRight (f) (z) (xs))
    case Nil => z
  }
val l1 = List (1,2,3,4,5,6)
val sum     = foldRight[Int,Int] (_+_) (0) (l1)
val product = foldRight[Int,Int] (_*_) (1) (l1)
def map[A,B] (f :A=>B) (l: List[A])=
  foldRight[A,List[B]] ((x, z) => Cons(f(x),z)) (Nil) (l)
```

Many HOFs can be implemented as special cases of folding

Exercises