@AndrzejWasowski
Andrzej Wąsowski

Advanced Programming

Partial Computations Advanced Programming

IT UNIVERSITY OF COPENHAGEN

PROCESS AND SYSTEM MODELS GROUP

- Traits
- Algebraic Data Types
- Variance of Type Parameters
- Fold functions
- Primary Constructors
- In the next episode ...

AGENDA

Traits: Rich or Fat Interfaces

Scala idiom: decompose classes into traits

```scala
// A class with a final property 'name' and
// a constructor. You can still add
// more members like in Java in braces.
abstract class Animal (val name :String)

// concrete methods
trait HasLegs {
  def run  () :String = "after you!"
  def jump () :String = "hop!"
}
// abstract method
trait Audible { def makeNoise () :String }
// field
trait Registered { var id :Int = 0 }

// multiple traits mixed in
class Frog (name:String) extends
  Animal(name) with HasLegs with Audible {
  def makeNoise () :String = "croak!"
} // Frog concrete, so provide makeNoise
```

```scala
// Mix directly into an object
val f = new Frog ("Kaj") with
          Registered
// f: Frog with Registered =
//              $anon$1@88f0bea
f.id = 42
println ( s"My name is ${f.name}")
println ( "I'm running " + f.run )
println( "I'm saying " + f.makeNoise)

}
```

	class	abstr. class	trait
mult. inheritance	−	−	+
data	+	+	+
concr. methods	+	+	+
abstr. methods	−	+	+
constr. params.	+	+	−

[Horstmann 2012, chpt. 10], [Odersky et al. 2014, chpt. 12] have more info than [Chiusano, Bjarnason 2014]

Algebraic Data Types (ADTs)

Def. Algebraic Data Type

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

Sets generated by constructos 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

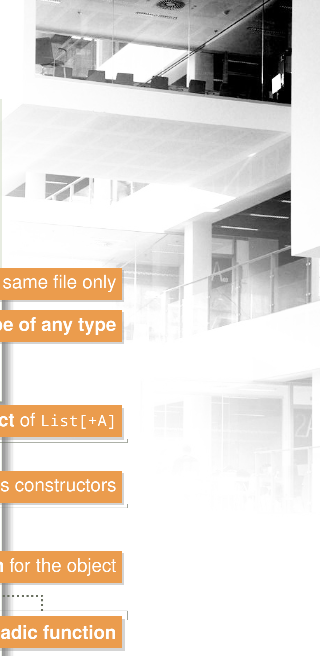
```
object List {
  def sum(ints :List[Int]) :Int =
    ints match { case Nil => 0
                 case Cons(x,xs) => x + sum(xs) }
  def apply[A](as :A*): List[A] =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

companion object of `List[+A]`

pattern matching uses case class constructors

overload function application for the object

variadic function



Mentimeter: Dynamic Virtual Dispatch

```
class Printable                  { void hello() { print ("printable "); }}
class Triangle extends Printable{ void hello() { print ("triangle ");}}
class Square extends Printable  { void hello() { print ("square "); }}
...
Square x = new Square ()
Printable y = new Triangle ()
x.hello ();
((Printable)x).hello ();
y.hello ();
((Printable)y).hello ();
```

1. printable printable printable printable
2. square printable triangle printable
3. square printable printable printable
4. square square triangle triangle
5. square square printable printable
6. The program will crash, or fail to type check

Variance of Type Parameters

- Write `A <: B` to say that `A` is a **subtype of** `B` (values of `A` fit where `B`s are expected)
- **Example**: if class `A` extends a class `B` then `A <: B`. Same for traits.
- Assume a generic type `T[B]`;
  `B` is a **covariant** parameter of `T` if for each `A <: B` we have that `List[A] <: List[B]`
  So we can use `List[A]` values where lists of `B`s are expected
- In Scala write `T[+B]` to specify that B is a covariant type parameter.
- Covariance common in pure programs. Scala lists are covariant (`List[+B]`).
- `A` is a **contravariant** parameter of `T` if whenever `A <: B` we have that `T[B] <: T[A]`
- Contravariance is needed if `A` is a return type, and in some impure situations.
  In Scala, write `T[-A]` to specify contravariance
- **Invariance** means that there is no automatic subtypes of generic type T;
  Invariance is default in Scala (when you omit the -/+)
- Recall that Java and C# generics **also** support variance of type parameters.
- Java has covariant Arrays (unsafe), Scala's arrays are invariant.

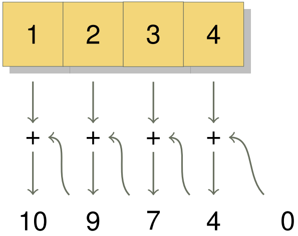
Why arrays shouldn't be covariant: http://stackoverflow.com/questions/6684493/why-are-arrays-invariant-but-lists-covariant
[Odersky et al. 2014, Chpt. 19] explains variance annotations in Scala in detail

Quiz: Variance of Type Parameters

```
abstract class A

abstract class B extends A

// Will the following code type check if T is
// (a) invariant,
// (b) covariant,
// (c) contravariant ?

val T[A] = new T[B]
```

[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

The Primary Constructor

```scala
class Person (val name: String, val age: Int) {
  println ("Just constructed a person")
  def description = s"$name is $age years old"
}
```

```java
class Person {
  private String name;
  private int age;
  public String name() { return name; }
  public int age() { return age; }

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
    System.out.println("Just constructed a person");
  }

  public String description ()
  { return name + "is " + age + " years old"; }
}
```

- Parameters become fields
- 'val' parameters become values, 'var' become variables
- If no parameter list, primary constructor takes none
- Constructor initializes fields and executes top-level statements of the class
- Like for all functions, parameters can take default values, reducing the need for overloading
- Note: primary constructors are used with case classes

[Horstmann 2012, Chpt. 5.7] explains the primary constructors in Scala

Scala: Summary

- **Basics** (objects, modules, functions, expressions, values, variables, operator overloading, infix methods, interpolated strings.)
- **Pure functions** (referential transparency, side effects)
- **Loops and recursion** (tail recursion)
- **Functions as values** (higher-order functions)
- **Parametric polymorphism** (monomorphic functions, dynamic and static dispatch)
- **Standard HOFs** in Scala's library
- **Anonymous functions** (currying, partial function application)
- **Traits** (fat interfaces, multiple inheritance, mixins)
- **Algebraic Data Types** (pattern matching, case classes)
- **Variance** of type parameters (covariance, contravariance, invariance)
- **Folding**
- **Primary constructors** (default parameter values)

In the next episode ...

- Basics of functional design: exceptions vs values, partial functions, the Option data type, exception oriented API of Option, for comprehensions, Either
- We will experience the first monadic computation (but refrain from defining monads yet)
- The reading should be relatively easy, so you should really try it!