



 @AndrzejWasowski

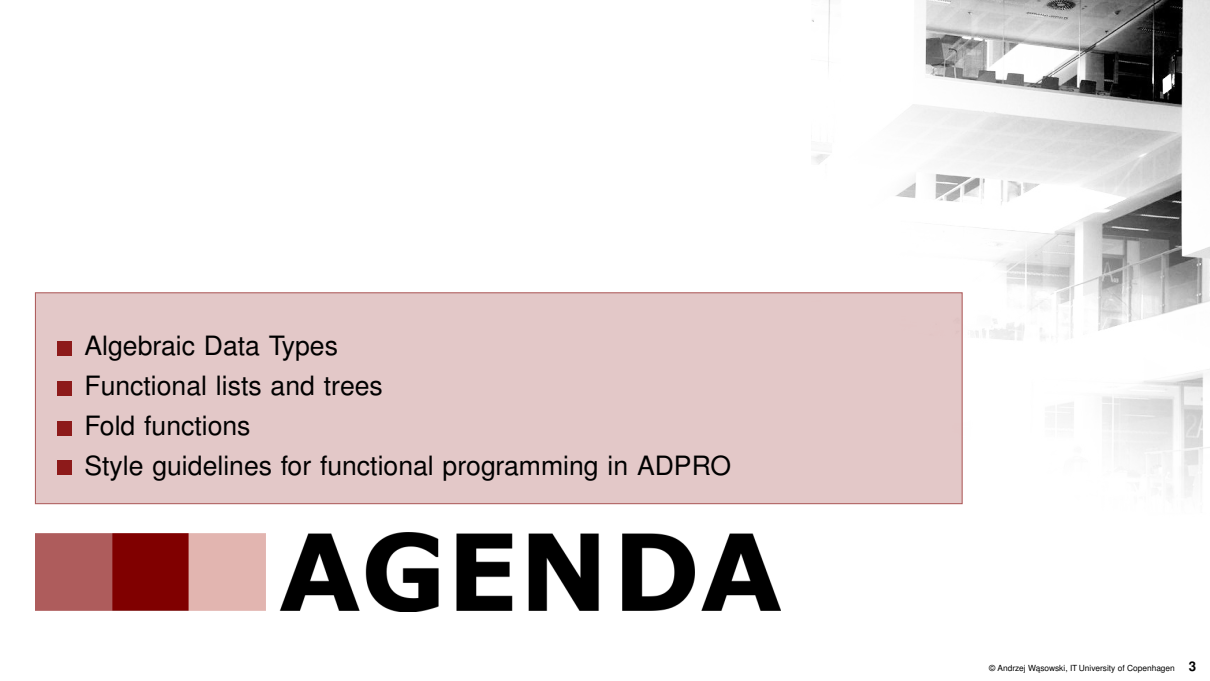
Andrzej Wąsowski
Florian Biermann

Advanced Programming

Algebraic Data Types

IT UNIVERSITY OF COPENHAGEN

S SOFTWARE
Q QUALITY
R RESEARCH

- 
- Algebraic Data Types
 - Functional lists and trees
 - Fold functions
 - Style guidelines for functional programming in ADPRO



AGENDA

Algebraic Data Types (ADTs) as Enums and Classes

Enums less general, but used more often

Def. Algebraic Data Type

A type generated by one or more constructors, each taking zero or more arguments.

The sets of objects generated by each constructor are **summed** (unioned), each constructor can be seen as a representation of a Cartesian **product** (tuple) of its arguments; thus the name **algebraic**.

ADT as enum

```
1 enum List[+A]:  
2   case Nil  
3   case Cons(head: A, tail: List[A])
```

ADT as case class hierarchy

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

sealed: extensible in the same file only

Nothing: **subtype of any type**

Algebraic Data Types (ADTs)

Def. Algebraic Data Type

A type generated by one or more constructors, each taking zero or more arguments.

The sets of objects generated by each constructor are **summed** (unioned), each constructor can be seen as a representation of a Cartesian **product** (tuple) of its arguments; thus the name **algebraic**.

Example: lists

```
1 enum List[+A]:  
2   case Nil  
3   case Cons(head: A, tail: List[A])
```

operations on lists

```
1 object List:  
2   def sum(ints: List[Int]): Int = ints match  
3     case Nil => 0  
4     case Cons(xxs) => x + sum(xs)  
5   def apply[A](as: A*): List[A] =  
6     if as.isEmpty then Nil  
7     else Cons(as.head, apply(as.tail*))
```

companion object of List[+A]

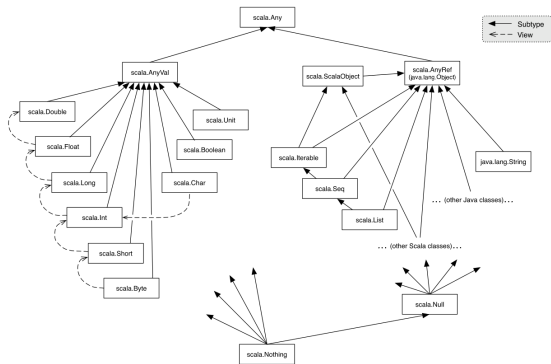
pattern matching against case constructors

overloading function application for the object

variadic function

Lists are covariant

All share the same tail!



For any type A we have that

`Nil <: List[Nothing] <: List[A]`

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

Poll: How is your recursion?

Mentimeter: 2820 9992

```
1 def f (a: List[Int]): Int = a match
2   case Nil => 0
3   case Cons(h, t) => h + f(t)
```

What is `f(List(42, -1, 1, -1, 1, -1))` ?



Function Values

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

```
1 def map(a: List[Int])(f: Int => Int): List[Int] = a match
2   case Nil => Nil
3   case Cons(h, tail) => Cons(f(h), map(tail)(f))
```

A functional (pure) example

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

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

An imperative (impure) example

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

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

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 Cons(h, tail) => Cons(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 Cons(h, tail) => Cons(f(h), map(tail)(f))
```

An example of use:

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

- A **polymorphic** function operates on values of (m)any types
- A polymorphic **type constructor** defines a parameterized family of types
- Don't confuse with inheritance-based polymorphism AKA "**dynamic dispatch**"

HOFs in the 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]`

Translate `this` list of `As` into a list of `Bs` using `f` to convert the values

`filter(p: A => Boolean): List[A]`

A sublist of `this` containing elements satisfying predicate `p`

`flatMap[B](f: A => List[B]): List[B]`

**type slightly simplified*

Apply `f` to elements of `this` and concatenate the produced lists

`take(n: Int): List[A]`

A list of first `n` elements of `this`.

`takeWhile(p: A => Boolean): List[A]`

A prefix of `this` containing elements satisfying `p`

`forall(p: A => Boolean): Boolean`

True iff `p` holds for all elements of `this`

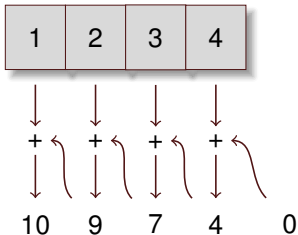
`exists(p: A => Boolean): Boolean`

True iff `p` holds for at least one element of `this`

More at <https://www.scala-lang.org/api/current/scala/collection/immutable/List.html>

Folds: Functional Loops

Sum of a list



What characterizes folds ?

- An **input list** $l = \text{List}(1, 2, 3, 4)$
- An **initial value** $z = 0$
- A **binary operation** $f: (\text{Int}, \text{Int}) \Rightarrow \text{Int} = _ + _$
- An **iteration algorithm**

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

Many HOFs are special cases of folding

Preferred Programming Style in ADPRO

Always choose the best possible style for an exercise and your abilities

Condemned (fail)



Forgivable (medium grade*)



Enlightened (top grade)

variables <
assignments <
return statement <
Any/Object type <

< values
< value bindings
< expression value
< parametric polymorphism

loops < tail recursion* < simple recursion < folds*
if conditions < pattern matching*

< compose dedicated HOFs
< use dedicated API
< Option or Either monad

exceptions <

* unless asked for explicitly, or really important for memory use.

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)
- **Folding**