(Week 4) Types and Pattern matching

Functions as Objects

Function values are treated as objects in Scala

The function type A => B is just an abbreviation for
scala.Function1[A, B], which is defined as

```
package scala
trait Function1[A, B] {
    def apply(x: A): B
}
```

So functions are objects with apply method

Anonymous functions are expanded to class def:

```
(x: Int) => x * x
        ⇓
{ class AnonFunc extends Function1[Int, Int] {
    def apply(x: Int) = x * x
  }
  new AnonFunc
}
```

or

anonymous class syntax

```
new Function1[Int, Int] {
  def apply(x: Int) = x * x
}
```

A function class $f(a, b)$ is expanded to
$f.apply(a, b)$

```
val f = (x: Int) => x * x
f(7)
```

=>

```
val f = new Function[Int, Int] {
  def apply(x: Int) = x * x
}
f.apply(7)
```

But a method, such as

```
def f(x: Int): Boolean = ...
```

is not a function value.
But if it's used where a function value is expected, it's converted automatically to the function value

$$(x: Int) => f(x)$$

Objects Everywhere

~~Syb~~ Subtyping and Generics

Principal forms of polymorphism:

- subtyping
- generics

Two main areas:

- bounds
- variance

Type Bounds

Consider the method assertAllPos which
- takes an IntSet
- returns the IntSet itself if all elements are positive
- throws an exception otherwise

What's the best type for this method?

def assertAllPos(s: IntSet): IntSet. ?

but:

. assertAllPos (Empty) = Empty
assertAllPos (NonEmpty) =
- NonEmpty
- Throw Exc

If we want to express that if it takes Empty set, it returns Empty,

if it gets NonEmpty, it returns NonEmpty ?

$$\text{def assertAllPos}[\underbrace{S <: \text{IntSet}}_{\text{upper bound of the type parameter } S}](r: S): S = \dots$$

It means that S can be instantiated only to types that conform to IntSet

Generally, the notation is

$S <: T$ - S is a subtype of T

$S >: T$ - S is a supertype of T,
or T is a subtype of S

Mixed Bounds

It's possible to mix lower and upper bounds

For instance,

[S >: NonEmpty <: IntSet]

would restrict S between NonEmpty and IntSet

Covariance

if NonEmpty <: IntSet

is

List[NonEmpty] <: List[IntSet] ?

It makes sense

we call this relationship covariance -
types for the sets hold ~~cov~~
covariant
because their subtyping relationship varies with the type parameter

Does covariance make sense for all types, not just for lists?

Java: Arrays in Java are covariant

But covariant array typing causes problems

Consider the following

```
NonEmpty[] a = new NonEmpty[] {
        new NonEmpty(...) }

IntSet[] b = a

b[0] = Empty
NonEmpty s = a[0]    <- ArrayStoreException!
```

The Liskov Substitution Principle

Barbara Liskov:

if $A <: B$, then everything one can do with a value of type B one should also be able to do with a value of type A.

Variance

Some types should be covariant, but some shouldn't.

A type that accepts mutations of its elements, should not be covariant.

But immutable types can be covariant if some conditions on methods are met.

List	Array
V	X.

Say C[T] is a parametrized type
A, B - types, $A <: B$

There are 3 possible relationships between
C[A] and C[B]

$C[A] <: C[B]$	C is covariant
$C[A] >: C[B]$	C is contravariant
neither C[A] nor C[B] is a subtype of the other	C is nonvariant

in Scala

```
class C[+A] {...}   covariant
class C[-A] {..}    contravariant
class C[A]  {..}    non-variant
```

<u>Eg.</u> We have two function

```
type A = IntSet => NonEmpty
type B = NonEmpty => IntSet
```

According to LSP, $A <: B$

<u>Typing Rules for functions</u>

$$\begin{array}{ccc} A2 & \Rightarrow & B2 \\ \wedge & & \vee \\ A1 & \Rightarrow & B1 \end{array}$$

If $A2 <: A1$ and $B1 <: B2$ then

$$A1 \Rightarrow B1 \quad <: \quad A2 \Rightarrow B2$$

So functions are contravariant in their argument type(s) and covariant in their result types

This leads to the following definition of the Function1 trait

```
package scala

trait Function1 [-T, +U] {
    def apply (x: T): U
}
```

In array example problematic operation was the update on an array

If we turn the array into class

```
class Array [+T] {
    def update(x: T) ...
}
```

The problematic combination is

- the covariant type parameter T
- which appears in parameter position of the ~~metod~~ method update

The Scala compiler will check that there are no problematic combinations when compiling a class with variance annotations

So:

- covariant types can appear only in method results
- contravariant types can appear only in method parameters.
- invariant types can appear everywhere

<u>We can make ~~th~~ List covariant</u> (Making Classes Covariant)

```
trait List[+T] {..}

object Empty extends List[Nothing] {...}
```

We want to have a (singleton) object for an empty list - there is only one empty list, no matter what's inside

Consider adding a method prepend which adds a new element and yields a new list

```
trait List [+T] {

    def prepend (elem: T): List [T] = new
        Cons (elem, this)

}
```

doesn't work!

it fails variance checking

And it violates LSP.

We have a list xs of type List[IntSet]

xs.prepend(Empty)

But if we have ys of type List[NonEmpty],

ys.prepend(Empty)

^
type mismatch

required: NonEmpty
found: Empty

So List[NonEmpty] cannot be a subtype of List[IntSet]

How can we make it variance-correct?

We can use a lower bound

```
def prepend [U >: T] (elem: U):
    List[U] = new Cons(elem, this)
```

This passes variance check, because

- covariant type parameters can appear in lower bounds of method type parameters

- contravariant type parameters can appear in upper bounds of a method

What type will be result of this function:

```
def f(xs: List[NonEmpty], x: Empty) =
    = xs prepend x
```

List[IntSet]

```
def prepend [U >: T] (elem: U): List[U] = ...
                 ||          ||
             NonEmpty      Empty
```

```
def f(xs: List[NonEmpty], x: Empty)
          List[IntSet]
```

$$
\begin{tikzcd}
 & \text{IntSet} & \\
\text{NonEmpty} \arrow[ur] & & \text{Empty} \arrow[ul]
\end{tikzcd}
$$

Decomposition

Suppose you want to write a small interpreter for arithmetic operations

Let's restrict ourselves to numbers and +

Expressions can be represented as a class hierarchy, with a base trait Expr, and two subclasses, Number and Sum

Solution 1 Object-Oriented Decomposition

```
trait Expr {
    def eval : Int
}

class Number(n: Int) extends Expr {

    def eval : Int = n
}

class Sum (e1 : Expr, e2 : Expr) extends Expr {
    def eval: Int = e1.eval + e2.eval
}
```

Limitation of O.O. decomposition

what if you want to simplify the expressions, say, using the rule

$$a * b + a * c = a * (b + c)$$

Problem: This is not local simplification - It cannot be encapsulated in the method of a single object

Pattern Matching

let's have the following methods:

eval
show
simplify

Expr

Number Sum Prod

Solution 2 Functional decomposition with Pattern-Matching

Attempts seen previously

- Classification and access methods: quadratic explosion (procedural way, instance of etc) isNum, isSum. etc
- Type tests and casts unsafe, low-level
- OO Decomposition - doesn't always work, need to touch all the classes to add a new method

The sole purpose of test and accessor functions

isSum isSum numValue left right

is to reserve the construction process:

- Which subclass was used?
- what were the arguments of the constructor

The situation is so common in many functional languages

Case Classes

A case class definition

```
trait Expr
case class Number(n: Int) extends Expr
case class Sum(...) extends Expr
```

it also implicitly defines companion objects with apply methods

```
object Number {
    def apply(n: Int) = new Number(n)
}

object Sum {
    def apply(e1: Expr, e2: Expr) = new Sum(e1, e2)
}
```

Now you can write Number(1) instead of new Number(1)

Pattern matching – a generalization of switch from Java to class hierarchies

the keyword match is used

Eg

```
def eval(e: Expr): Int = e match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

Syntax

Rules:

- match is followed by a sequence of cases,

  pat => expr

- each case associates an expression expr with a pattern pat
- A MatchError exception is thrown if no pattern matches the value of the selector

Forms of Patterns

Patterns are constructed from

- constructors, eg. Number, Sum
- variables, e.g. n, e1, e2
- wildcard patterns _
- constants, e.g. 1, true.

Variables always begin with a lowercase letter

The same variable can appear only once in a pattern.
So, Sum(x, x) is not a legal pattern

What do patterns match?

- A constructor pattern $C(p_1, \ldots, p_n)$ matches all the values of type C (or a subtype) that have been constructed with arguments matched the pattern $p_1 \ldots p_n$

- a variable pattern x matches any value, and binds the name x to this value

- a constant pattern c matches values that are equal to c (in the sense of ==)

Lists

a list with elements $x_1 \ldots x_n$ is written as
List($x_1, \ldots, x_n$)

- lists are immutable
- lists are recursive
- they are homogeneous - the elements of a list must all have the same type

All lists are constructed from

- the empty list Nil, and
- the construction operator :: (cons)

x :: xs gives a new list with the first element x followed by xs

```
fruit = "apples" :: ("pears" :: Nil)
nums = 1 :: (2 :: (3 :: Nil))

empty = Nil
```

:: has the right associativity

A :: B :: C is A :: (B :: C)

So we can omit the parenthesis

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
                   ||
Nil.::(4).::(3).::(2).::(1)
```

Operators on Lists

- head   the first element of the list
- tail   all elements except first
- isEmpty   true if empty, false otherwise

It's also possible to use lists in pattern matching

- Nil → the Nil constant
- p :: ps → a list with head p and tail ps
- List(p1, .., pn) → same as p1 :: ... :: pn

e.g.

List(2 :: xs) will match a list starting from 2

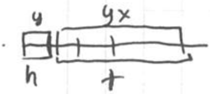
Sorting list

Suppose we want to sort a lost of numbers in ascessing order:

Insertion Sort

```
def isort (xs: List[Int]): List[Int] = xs match {
      case List() => List()
      case y :: ys => insert(y, isort(ys))
}
```

```
def insert (x: Int, xs: List[Int]): List[Int] =
        xs match {
    case List() => List(x)
    case y :: ys => if(x <= y)
                        x :: xs
                   else
                        y :: insert(x, ys)
}
```



$O(n^2)$