



# Evaluation and Operators

Principles of Functional Programming



## Classes and Substitutions

We previously defined the meaning of a function application using a computation model based on substitution. Now we extend this model to classes and objects.

*Question:* How is an instantiation of the class  $C(e_1, \dots, e_m)$  evaluated?

*Answer:* The expression arguments  $e_1, \dots, e_m$  are evaluated like the arguments of a normal function. That's it.

The resulting expression, say,  $C(v_1, \dots, v_m)$ , is already a value.



## Classes and Substitutions

Now suppose that we have a class definition,

```
class C( $x_1, \dots, x_m$ ) { ... def f( $y_1, \dots, y_n$ ) = b ... }
```

where

- ▶ The formal parameters of the class are  $x_1, \dots, x_m$ .
- ▶ The class defines a method  $f$  with formal parameters  $y_1, \dots, y_n$ .

(The list of function parameters can be absent. For simplicity, we have omitted the parameter types.)

*Question:* How is the following expression evaluated?

```
C( $v_1, \dots, v_m$ ).f( $w_1, \dots, w_n$ )
```



## Classes and Substitutions (2)

*Answer:* The expression  $C(v_1, \dots, v_m).f(w_1, \dots, w_n)$  is rewritten to:

$$[w_1/y_1, \dots, w_n/y_n][v_1/x_1, \dots, v_m/x_m][C(v_1, \dots, v_m)/this] b$$

There are three substitutions at work here:

- ▶ the substitution of the formal parameters  $y_1, \dots, y_n$  of the function  $f$  by the arguments  $w_1, \dots, w_n$ ,
- ▶ the substitution of the formal parameters  $x_1, \dots, x_m$  of the class  $C$  by the class arguments  $v_1, \dots, v_m$ ,
- ▶ the substitution of the self reference *this* by the value of the object  $C(v_1, \dots, v_m)$ .



# Object Rewriting Examples

`Rational(1, 2).numer`



## Object Rewriting Examples

`Rational(1, 2).numer`

$\rightarrow [1/x, 2/y] [] [\text{Rational}(1, 2)/\text{this}] \text{ x}$



## Object Rewriting Examples

`Rational(1, 2).numer`

$\rightarrow [1/x, 2/y] [] [\text{Rational}(1, 2)/\text{this}] \ x$

$= 1$



## Object Rewriting Examples

`Rational(1, 2).numer`

$\rightarrow [1/x, 2/y] [] [Rational(1, 2)/this] x$

$= 1$

`Rational(1, 2).less(Rational(2, 3))`



## Object Rewriting Examples

`Rational(1, 2).numer`

$\rightarrow [1/x, 2/y] [] [Rational(1, 2)/this] x$

$= 1$

`Rational(1, 2).less(Rational(2, 3))`

$\rightarrow [1/x, 2/y] [Rational(2, 3)/that] [Rational(1, 2)/this]$

`this.numer * that.denom < that.numer * this.denom`



## Object Rewriting Examples

`Rational(1, 2).numer`

$\rightarrow [1/x, 2/y] [] [Rational(1, 2)/this] x$

$= 1$

`Rational(1, 2).less(Rational(2, 3))`

$\rightarrow [1/x, 2/y] [Rational(2, 3)/that] [Rational(1, 2)/this]$   
`this.numer * that.denom < that.numer * this.denom`

$= Rational(1, 2).numer * Rational(2, 3).denom <$   
`Rational(2, 3).numer * Rational(1, 2).denom`



## Object Rewriting Examples

`Rational(1, 2).numer`

$\rightarrow [1/x, 2/y] [] [Rational(1, 2)/this] \times$

$= 1$

`Rational(1, 2).less(Rational(2, 3))`

$\rightarrow [1/x, 2/y] [Rational(2, 3)/that] [Rational(1, 2)/this]$

`this.numer * that.denom < that.numer * this.denom`

$= Rational(1, 2).numer * Rational(2, 3).denom <$

`Rational(2, 3).numer * Rational(1, 2).denom`

$\Rightarrow 1 * 3 < 2 * 2$

$\Rightarrow true$



## Extension Methods

Having to define all methods that belong to a class inside the class itself can lead to very large classes, and is not very modular.

Methods that do not need to access the internals of a class can alternatively be defined as extension methods.

For instance, we can add min and abs methods to class Rational like this:

```
extension (r: Rational):  
  def min(s: Rational): Boolean = if s.less(r) then s else r  
  def abs: Rational = Rational(r.numer.abs, r.denom)
```



## Using Extension Methods

Extensions of a class are visible if they are listed in the companion object of a class (as in the code above) or if they defined or imported in the current scope.

Members of a visible extensions of class C can be called as if they were members of C. E.g.

```
Rational(1/2).min(Rational(2/3))
```

### Caveats:

- ▶ Extensions can only add new members, not override existing ones.
- ▶ Extensions cannot refer to other class members via this



## Extension Methods and Substitutions

Extension method substitution works like normal substitution, but

- ▶ instead of this it's the extension parameter that gets substituted,
- ▶ class parameters are not visible, so do not need to be substituted at all.

```
Rational(1, 2).min(Rational(2, 3))
```



## Extension Methods and Substitutions

Extension method substitution works like normal substitution, but

- ▶ instead of this it's the extension parameter that gets substituted,
- ▶ class parameters are not visible, so do not need to be substituted at all.

```
Rational(1, 2).min(Rational(2, 3))
```

→  $[Rational(1, 2)/r] [Rational(2, 3)/s]$  if  $x.less(r)$  then  $s$  else  $r$



## Extension Methods and Substitutions

Extension method substitution works like normal substitution, but

- ▶ instead of this it's the extension parameter that gets substituted,
- ▶ class parameters are not visible, so do not need to be substituted at all.

```
Rational(1, 2).min(Rational(2, 3))
```

→  $[Rational(1, 2)/r] [Rational(2, 3)/s]$  if  $x.less(r)$  then  $s$  else  $r$

=

```
if Rational(2, 3).less(Rational(1, 2)
then Rational(2, 3)
else Rational(1, 2)
```



# Operators

In principle, the rational numbers defined by `Rational` are as natural as integers.

But for the user of these abstractions, there is a noticeable difference:

- ▶ We write  $x + y$ , if  $x$  and  $y$  are integers, but
- ▶ We write `r.add(s)` if  $r$  and  $s$  are rational numbers.

In Scala, we can eliminate this difference. We proceed in two steps.



## Step 1: Relaxed Identifiers

Operators such as + or < count as identifiers in Scala.

Thus, an identifier can be:

- ▶ *Alphanumeric*: starting with a letter, followed by a sequence of letters or numbers
- ▶ *Symbolic*: starting with an operator symbol, followed by other operator symbols.
- ▶ The underscore character '\_' counts as a letter.
- ▶ Alphanumeric identifiers can also end in an underscore, followed by some operator symbols.

Examples of identifiers:

x1      \*      +?%&      vector\_++      counter\_ =



## Step 1: Relaxed Identifiers

Since operators are identifiers, it is possible to use them as method names.  
E.g.

```
extension (x: Rational):  
  def + (y: Rational): Rational = x.add(y)  
  def * (y: Rational): Rational = x.mul(y)  
  ...
```

This allows rational numbers to be used like Int or Double:

```
val x = Rational(1, 2)  
val y = Rational(1, 3)  
x * x + y * y
```



## Step 2: Infix Notation

An operator method with a single parameter can be used as an infix operator.

An alphanumeric method with a single parameter can also be used as an infix operator if it is declared with an `@infix` annotation. E.g.

```
extension (x: Rational):  
  @infix def min(that Rational): Rational = ...
```

It is therefore possible to write

<code>r + s</code>		<code>r.+(s)</code>
<code>r &lt; s</code>	<code>/* in place of */</code>	<code>r.&lt;(s)</code>
<code>r min s</code>		<code>r.min(s)</code>



## Precedence Rules

The *precedence* of an operator is determined by its first character.

The following table lists the characters in increasing order of priority precedence:

(all letters)

|

^

&

< >

= !

:

+ -

\* / %

(all other special characters)



## Exercise

Provide a fully parenthesized version of

$a + b \wedge c \vee d \text{ less } a \implies b \mid c$

Every binary operation needs to be put into parentheses, but the structure of the expression should not change.