

Evaluation and Operators

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 `new C(e1, ..., em)` evaluated?

Answer: The expression arguments `e1, ..., em` are evaluated like the arguments of a normal function. That's it.

The resulting expression, say, `new C(v1, ..., vm)`, 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?

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

Classes and Substitutions (2)

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

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

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 $\text{new } C(v_1, \dots, v_m)$.

Object Rewriting Examples

```
new Rational(1, 2).numer
```

Object Rewriting Examples

`new Rational(1, 2).numer`

`→ [1/x, 2/y] [] [new Rational(1,2)/this] x`

Object Rewriting Examples

`new Rational(1, 2).numer`

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

$= 1$

Object Rewriting Examples

```
new Rational(1, 2).numer
```

```
→ [1/x, 2/y] [] [new Rational(1,2)/this] x
```

```
= 1
```

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


Object Rewriting Examples

```
new Rational(1, 2).numer
```

```
→ [1/x, 2/y] [] [new Rational(1,2)/this] x
```

```
= 1
```

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

```
→ [1/x, 2/y] [newRational(2,3)/that] [new Rational(1,2)/this]
```

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

Object Rewriting Examples

```
new Rational(1, 2).numer
```

```
→ [1/x, 2/y] [] [new Rational(1,2)/this] x
```

```
= 1
```

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

```
→ [1/x, 2/y] [newRational(2,3)/that] [new Rational(1,2)/this]
```

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

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

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

Object Rewriting Examples

`new Rational(1, 2).numer`

→ `[1/x, 2/y] [] [new Rational(1,2)/this] x`

= 1

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

→ `[1/x, 2/y] [newRational(2,3)/that] [new Rational(1,2)/this]`

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

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

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

→ `1 * 3 < 2 * 2`

→ `true`

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: Infix Notation

Any method with a parameter can be used like an infix operator.

It is therefore possible to write

`r add s`

`r less s`

`r max s`

`/* in place of */`

`r.add(s)`

`r.less(s)`

`r.max(s)`

Step 2: Relaxed Identifiers

Operators can be used as identifiers.

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_ =

Operators for Rationals

A more natural definition of class Rational:

```
class Rational(x: Int, y: Int) {  
  private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)  
  private val g = gcd(x, y)  
  def numer = x / g  
  def denom = y / g  
  def + (r: Rational) =  
    new Rational(  
      numer * r.denom + r.numer * denom,  
      denom * r.denom)  
  def - (r: Rational) = ...  
  def * (r: Rational) = ...  
  ...  
}
```

Operators for Rationals

... and rational numbers can be used like Int or Double:

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


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

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

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