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 $\texttt{C}(e_1, ..., e_m)$ evaluted?

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

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

Classes and Substitutions

Now suppose that we have a class definition,

$$\texttt{class C}(\texttt{x}_1, ..., \texttt{x}_m)\{ \; ... \; \texttt{def f}(\texttt{y}_1, ..., \texttt{y}_n) = \texttt{b} \; ... \; \}$$

where

- The formal parameters of the class are $\texttt{x}_1, ..., \texttt{x}_m$.
- The class defines a method $\texttt{f}$ with formal parameters $\texttt{y}_1, ..., \texttt{y}_n$.

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

Question: How is the following expression evaluated?

$$\texttt{C}(\texttt{v}_1, ..., \texttt{v}_m).\texttt{f}(\texttt{w}_1, ..., \texttt{w}_n)$$

Classes and Substitutions (2)

Answer: The expression $\texttt{C}(\texttt{v}_1, \ldots, \texttt{v}_m).\texttt{f}(\texttt{w}_1, \ldots, \texttt{w}_n)$ is rewritten to:

$$[\texttt{w}_1/\texttt{y}_1, \ldots, \texttt{w}_n/\texttt{y}_n][\texttt{v}_1/\texttt{x}_1, \ldots, \texttt{v}_m/\texttt{x}_m][\texttt{C}(\texttt{v}_1, \ldots, \texttt{v}_m)/\texttt{this}]\,\texttt{b}$$

There are three substitutions at work here:

- the substitution of the formal parameters $\texttt{y}_1, \ldots, \texttt{y}_n$ of the function $\texttt{f}$ by the arguments $\texttt{w}_1, \ldots, \texttt{w}_n$,
- the substitution of the formal parameters $\texttt{x}_1, \ldots, \texttt{x}_m$ of the class $\texttt{C}$ by the class arguments $\texttt{v}_1, \ldots, \texttt{v}_m$,
- the substitution of the self reference *this* by the value of the object $\texttt{C}(\texttt{v}_1, \ldots, \texttt{v}_n)$.

Object Rewriting Examples

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

Object Rewriting Examples

`Rational(1, 2).numer`

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

Object Rewriting Examples

`Rational(1, 2).numer`

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

$= 1$

Object Rewriting Examples

`Rational(1, 2).numer`

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

$=$ `1`

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

Object Rewriting Examples

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

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

$=$ `1`

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

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

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

Object Rewriting Examples

`Rational(1, 2).numer`

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

$=$ `1`

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

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

```
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
```

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

$=$ `1`

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

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

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

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

$\twoheadrightarrow$ `1 * 3 < 2 * 2`

$\twoheadrightarrow$ `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: 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.

```
class Rational {
  def + (x: Rational): Rational = ...
  def * (x: Rational): Rational = ...
}
```

Step 2: Infix Notation

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

A normal method with a single parameter can also be used as an infix operator if it is declared @infix. E.g.

```
class Rational {
  @infix def max(that Rational): Rational = ...
}
```

It is therefore possible to write

```
  r + s                                            r.+(s)
  r < s            /* in place of */               r.<(s)
  r max s                                          r.max(s)
```

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 then a else gcd(b, a % b)
  private val g = gcd(x, y)
  def numer = x / g
  def denom = y / g
  def + (r: Rational) = Rational(
    numer * r.denom + r.numer * denom,
    denom * r.denom)
  def - (r: Rational): Rational = ...
  def * (r: Rational): Rational = ...
  ...
end Rational
```

Operators for Rationals

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
```

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 ^? c ?^ d less a ==> b | c
```

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