Partial Documentation

adv-genarith

©Copyright 1991, 1992, 1993 by ILOG,
2, avenue Galliéni
B.P. 85
94250 Gentilly Cedex
FRANCE
All Rights Reserved

This document and the software described in this document are furnished under a license or non-disclosure agreement, and may be used or copied only in accordance with the terms of such license or non-disclosure agreement.

Version of December 2, 1993

0.1 Extending the Generic Arithmetic

The generic arithmetic is implemented by means of a few generic functions which can be extended by the user to take in account new types of numbers. In this section, we illustrate this extensibility with a simplistic and partial definition of the class `<complex>` of complex numbers.

0.1.1 Defining new types of numbers

The class `<number>` can be subclassed to introduce new types of numbers. For example, you can define a new type of complex numbers with:

```
(defstructure <complex> <number>
   (:real
    :imaginary))
```

The predicate `numberp` will naturally take this new type into account:

```
? (numberp #s(<complex> real: 12 imaginary: 4.5))
= #s(<complex> real: 12 imaginary: 4.5)
```

0.1.2 Unary generic functions

You can safely add methods to the unary generic functions `integer`, `float` and `abs` for new types of numbers:

```
(defmethod integer ((c <complex>))
  (assert (= 0 (complex-imaginary c))
    (error 'integer #m:imaginary-part-not-zero c))
  (integer (complex-real c)))

(defmethod float ((c <complex>))
  (assert (= 0 (complex-imaginary c))
    (error 'float #m:imaginary-part-not-zero c))
  (float (complex-real c)))

(defmethod abs ((c <complex>))
  (let ((r (complex-real c))
        (im (complex-imaginary c)))
    (sqrt (+ (* r r) (* im im)))))
```

0.1.3 Mixed-arithmetic functions and comparisons

`(binary+` *x y*`)` [*generic function*]

`(binary*` *x y*`)` [*generic function*]

`(binary-` *x y*`)` [*generic function*]

`(binary/` *x y*`)` [*generic function*]

`(binary=` *x y*`)` [*generic function*]

`(binary<` *x y*`)` [*generic function*]

Summary Generic pairwise operations.

Arguments x `<number>`
y `<number>`

Result `<number>`

These functions are the binary and generic versions of the arithmetic functions `+`, `*`, `-`, `/`, `=` and `<`, which are all implemented in terms of the binary generic functions. For example, the expression `(+ x y z)` is computed as `(binary+ (binary+ x y) z)`, and `(< x y z)` is computed as `(and (binary< y z) (binary< x y))`. Arithmetic functions which do not have a direct binary equivalent are also computed as appropriate in terms of binary functions; for example, `(> x y)` is computed as`(if (binary< y x) x y)`. So to have the arithmeric functions work with a new type of number, you just have to add methods to the binary functions to take the new type into account.

For comparisons, you do not have to take care of returning the first argument when the result is true; just return any true boolean value.

At first sight, extending the arithmetic to a new type, say `<complex>`, requires extending each binary generic function with methods for complexes in combination with any possible number type, so that you can apply mixed operations involving both complexes and other kinds of numbers. Fortunately, this is not necessary. All you have to do is to define a method for two complexes for each binary generic function. The system can be extended with new coercion rules to be applied when the generic function is called with a complex and any other kind of number. This extension is done through the generic function `promote-number`.

`(promote-number` *x y*`)` [*generic function*]

Summary Promotion to the higher type.

Arguments x `<number>`
y `<number>`

Result `<number>`

This generic function introduces a conversion hierarchy between number types which is orthogonal to the class hierarchy. A number class `<X>` is said to be *higher* than another number class Y when, in binary operations applied to an instance of `<X>` and an instance of `<Y>`, the instance of X is coerced to a `<Y>`. In the core TALK, `<float>` is higher than `<ratio>` and `<integer>`, and `<ratio>` is higher than `<integer>`. Thus in operations involving floating-point numbers, all the other arguments are converted to floating-point numbers, and in operations involving ratios and integers, the integers are converted to ratios.

This hierarchy is implemented by the generic function `promote-number`. If the class of y is higher than the class of x, then `promote-number` returns the coercion of x into the class of y; otherwise, it returns x.

The function `promote-number` is called in the default method for numbers of every binary operation `binary-op`:

```
(defmethod binary-op ((x <number>) (y <number>))
   (binary-op (promote-number x y) (promote-number y x)))
```

Since `(promote-number x y)` and `(promote-number y x)` have the same class, methods for `binary-op` have to be defined only for pairs of arguments of this class.

In conclusion, to extend the basic arithmetic operations with a new type, you have to:

1. Define `promote-number` methods for coercion from lower types to the new type, and from the new type to higher types;
2. Define binary generic operations methods for pairs of instances of the new type.

The default method for `promote-number` is:

```
(defmethod promote-number ((x <number>) (y <number>))
  x)
```

So you only have to define those methods which actually involve a coercion of x.

For example, given that `<complex>` is higher than any rational or floating-point class, we define:

```
(defmethod promote-number ((x <rational>) (c <complex>))
  (make-complex real: x imaginary: 0))

(defmethod promote-number ((x <float>) (c <complex>))
  (make-complex real: x imaginary: 0))
```

To have `+` work with complex numbers, define a `generic+` method for two complexes:

```
(defmethod binary+ ((x <complex>) (y <complex>))
  (make-complex real: (+ (complex-real x) (complex-real y))
                imaginary: (+ (complex-imaginary x) (complex-imaginary y))))
```

Now the function `+` can be applied to mixed arguments including both complex numbers and other types of numbers:

```
? (+ #s(<complex> real: 12 imaginary: 3/4)
     #s(<complex> real: 1.2 imaginary: 0))
= #s(<complex> real: 13.2 imaginary: 3/4)
? (+ #s(<complex> real: 12 imaginary: 3/4) 2/3)
= #s(<complex> real: 38/3 imaginary: 3/4)
? (1+ #s(<complex> real: 12 imaginary: 3/4))
= #s(<complex> real: 13 imaginary: 3/4)
```

0.1.4 Other Functions

All the other arithmetic non-generic functions are defined in terms of the previous ones. So they will work automatically.