Extra Numerical Types for Common Lisp

Marco Antoniotti
mantoniotti at common-lisp.net

February 25, 2008

Abstract

This document presents a new set of portable type specifiers that can be used to improve the "precision" of type declarations in numerical code.

1 Introduction

When working on numerical algorithms it is sometimes useful to further constrain the types of certain values to sub-intervals of the usual types present in Common Lisp. A typical example is that of indices running over the dimensions of an array: such integers values should not be negative. While several Common Lisp implementations already have certain special "sub-interval" type specifiers that can be used in implementation dependent code, it seems natural and relatively uncontroversial to propose a set of specialized types that codify usual mathematical numerical sets and intervals. This document puts forward such a proposal.

The rest of this document is organized in two parts: a description of the new types proposed, and a brief discussion pertaining the rationale and the foreseen costs of adoption by the various implementers.

2 Description

The extra types are presented in terms of the original Common Lisp type they partition. As it will appear in the following, most types are simply partitions around the appropriate *zero*.

2.1 Numerical Sub-interval Types

There are several numerical types which represent traditional mathematical sets in their representation-dependent implementations. The numerical types are the following:

- `negative-`T

- `non-positive-`T
- `non-negative-`T
- `positive-`T
- `array-index`

where T is one of `fixnum`, `integer`, `rational`, `ratio`, `real`, `float`, `short-float`, `single-float`, `double-float`, `long-float`. Each of these types is defined in a very straightforward way. The pseudo-code in the subsections hereafter shows how each type can be defined.

2.1.1 `FIXNUM` Sub-interval Types

The subtypes of type `fixnum`[1] may be defined as follows. Note that `fixnum` does not allow for compound type specifiers.

```
(deftype negative-fixnum ()
   `(integer ,most-negative-fixnum -1))

(deftype non-positive-fixnum ()
   `(integer ,most-negative-fixnum 0))

(deftype non-negative-fixnum ()
   `(integer 0 , most-positive-fixnum))

(deftype positive-fixnum ()
   `(integer 1 ,most-positive-fixnum))
```

The predicates following predicates are also defined in the most straightforward way.

- `negative-fixnum-p`
- `non-positive-fixnum-p`
- `non-negative-fixnum-p`
- `positive-fixnum-p`

2.1.2 `INTEGER` Sub-interval Types

The subtypes of type `integer` may be defined as follows.

[1] There is no *class* for `fixnum` in the ANSI specification , only a *type*. This is a consequence of several factors and choices made in standardization process.

```
(deftype negative-integer ()
   '(integer * -1))

(deftype non-positive-integer ()
   '(integer * 0))

(deftype non-negative-integer ()
   '(integer 0 *))

(deftype positive-integer ()
   '(integer 1 *))
```

The following predicates are also defined in the most straightforward way.

- `negative-integer-p`
- `non-positive-integer-p`
- `non-negative-integer-p`
- `positive-integer-p`

2.1.3 `RATIONAL` Sub-interval Types

The subtypes of type `rational` may be defined as follows.

```
(deftype negative-rational ()
   '(rational * (0)))

(deftype non-positive-rational ()
   '(rational * 0))

(deftype non-negative-rational ()
   '(rational 0 *))

(deftype positive-rational ()
   '(rational (0) *))
```

The following predicates are also defined in the most straightforward way.

- `negative-rational-p`
- `non-positive-rational-p`
- `non-negative-rational-p`
- `positive-rational-p`

2.1.4 `RATIO` Sub-interval Types

The subtypes of type `ratio` may be defined as follows. Note that `ratio` does not allow for compound type specifiers. Also, there are other technical difficulties in this case if we wanted to be *very* coherent with the background of the ANSI specification. `ratio`s are defined exactly as the *ratio of two non-zero integers, whose greatest common divisor is one and of which the denominator is greater than one*[2]. This makes it very difficult to use the type specifier machinery effectively, and we must resort to the `satisfies` type specifier. A possible definition of the `ratio` sub-interval based on `satisfies` needs therefore the definition of the `ratiop` predicate (which is absent from the ANSI specification) alongside the `ratio-plusp` and `ratio-minusp` predicates.

```
(defun ratiop (x)
   (and (typep x 'rational)
        (> (denominator x) 1)))

(defun ratio-plusp (x)
   (and (ratiop x) (plusp x)))

(defun ratio-minusp (x)
   (and (ratiop x) (minusp x)))
```

These predicates could be implemented more efficiently by a given implementation.
Now it is possible to define the `ratio` types.

```
(deftype negative-ratio ()
   '(satisfies ratio-minusp))

(deftype non-positive-ratio ()
   'negative-ratio)

(deftype non-negative-ratio ()
   'positive-ratio)

(deftype positive-ratio ()
   '(satisfies ratio-plusp))
```

The following predicates are also defined in the most straightforward way.

- `negative-ratio-p`
- `non-positive-ratio-p`
- `non-negative-ratio-p`
- `positive-ratio-p`

[2] A consequence of the definition is that, in general `(typep 42 'ratio)` ⇒ `NIL`, and, in particular, `(typep 0 'ratio)` ⇒ `NIL`.

2.1.5 `REAL` Sub-interval Types

The subtypes of type `real` may be defined as follows.

```
(deftype negative-real ()
   '(real * (0)))

(deftype non-positive-real ()
   '(real * 0))

(deftype non-negative-real ()
   '(real 0 *))

(deftype positive-real ()
   '(real (0) *))
```

The following predicates are also defined in the most straightforward way.

- `negative-real-p`
- `non-positive-real-p`
- `non-negative-real-p`
- `positive-real-p`

2.1.6 `FLOAT` Sub-interval Types

The subtypes of the various *float* types may be defined as follows.

```
(deftype negative-T () '(T * (zero)))
(deftype non-positive-T () '(T * zero))
(deftype non-negative-T () '(T zero *))
(deftype positive-T () '(T (zero) *))
```

where T is one of `float`, `short-float`, `single-float`, `double-float`, and `long-float`. Also, *zero* is written in the appropriate syntax; i.e.,

- `0.0E0` for `float` types
- `0.0S0` for `short-float` types
- `0.0F0` for `single-float` types
- `0.0D0` for `double-float` types
- `0.0L0` for `long-float` types

as per the ANSI specification.

The appropriate type predicates (whose names are not listed here) can be implemented in the usual straightforward way.

2.1.7 `ARRAY-INDEX` Sub-interval Type

The `array-index` type is obviously useful and used in many implementations. It is just not standardized. The definition of `array-index` could be the following one.

```
(deftype array-index ()
   `(integer 0 (,array-total-size-limit)))
```

The `array-index-p` predicate can thus be defined immediately.

3 Discussion

The proposed types are obviously just a convenience, yet they may be used to improve the "self-documenting" nature of programs. Some of the definitions are directly useful. Looping through an array can be done in several ways, but often one just forgoes to write down the proper index declaration or resorts to the practically omni-present implementation-dependent equivalent of `array-index`.

The cost of adoption is very small for this facility. Legacy code is not affected and new code may - as stated - become more self-documenting, and possibly efficient.

Some of the names are long. This is in the tradition of Common Lisp. However, it would be possible to provide abbreviated versions by shortening them using the following scheme

- `negative` becomes `neg`
- `non-negative` becomes `nonneg`
- `positive` becomes `pos`
- `non-positive` becomes `nonpos`

References

[1] *The Common Lisp Hyperspec,* published online at `http://www.lisp.org/HyperSpec/FrontMatter/index.html`, 1994.