Generic Equality and Comparison for Common Lisp

Marco Antoniotti
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
Viale Sarca 336, U14, Milan (MI), Italy
mantoniotti at common-lisp.net, marco.antoniotti at unimib.it

March 1, 2011

Abstract

This document presents new generic functions for Common Lisp that provide user hooks for extensible *equality* and *comparison tests*, as well as generic *hashing*. This is in addition to the standard equality and comparison predicates. The current proposal is *minimal*, in the sense that it just provides one conceptually simple set of hooks in what is considered a cross-language consensus.

1 Introduction

Several Common Lisp functions rely on the `:test` keyword to pass a predicate to be used in their operations. This is a satisfactory solution in most cases, yet, while *writing* algorithms and libraries it would be useful to have "hooks" in the type and class system allowing for the definition of *extensible equality* and *comparison tests*.

This proposal contains a **minimal** set of (generic) functions that can be recognized in several language specifications, e.g., Java.

The specification is centered on two concepts: that of an *equality* test and that of a *comparison* generic operator. The *comparison* operator returns different values depending on whether the its execution determines the *ordering relationship* (or lack thereof) of two objects.

In addition to these two basic concepts, the specification provides notion of generic *hash function* too. Of course, there are several "caveats" to be taken into consideration in the interplay between equality and hashing.

2 Description

The the proposal describes the *equality*, *comparison* and *hashing'* operators. The *equality* operator is called `EQUALS` and some synonyms are also defined. The *comparison* operators is called `COMPARE`. The utility functions `LT`, `GT`, `LTE`, and `GTE` are also defined. Some synonyms are also defined. The *hashing* operator is called `HASH-CODE`.

The *comparison* operator returns one of four values: the symbols `<`, `>`, `=`, or `/=`. The intent of such definition is to make it usable in conjunction with `case`, `ccase`, and `ecase`; also, its intent is to make it possible to capture *partial orders* among objects in a set.

3 Equality and Comparison Dictionary

3.1 Standard Generic Function `EQUALS`

Syntax:

`EQUALS` *a b* `&rest` *keys* `&key` *recursive-p* `&allow-other-keys`[1] ⇒ *result*

Known Method Signatures:

`EQUALS` (*a* `T`) (*b* `T`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

`EQUALS` (*a* `number`) (*b* `number`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

`EQUALS` (*a* `cons`) (*b* `cons`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

`EQUALS` (*a* `character`) (*b* `character`)
`&rest` *keys* `&key` *recursive-p case-sensitive-p* `&allow-other-keys`

`EQUALS` (*a* `string`) (*b* `string`)
`&rest` *keys* `&key` *recursive-p case-sensitive-p* `&allow-other-keys`

`EQUALS` (*a* `array`) (*b* `array`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

`EQUALS` (*a* `hash-table`) (*b* `hash-table`)
`&rest` *keys* `&key` *recursive-p* (*by-key* `t`) (*by-value* `t`) (*check-properties* `t`)
`&allow-other-keys`

[1] Maybe it would make sense to supply a `:key` parameter (defaulting to `identity`) as well.

Arguments and Values:

a b − Common Lisp objects.

recursive-p − a *generalized boolean*; default is `NIL`.

result − a `boolean`.

keys − a `list` (as per the usual behavior).

by-key − a *generalized boolean*; default is `T`.

by-values − a *generalized boolean*; default is `T`.

check-properties − a *generalized boolean*; default is `NIL`.

case-sensitive-p − a *generalized boolean*; default is `T`.

Description:

The `EQUALS` generic functions defines methods to test for "equality" of two objects *a* and *b*. When two objects *a* and *b* are `EQUALS` under an appropriate and type/class dependent notion of "equality", then the function returns `T` as *result*; otherwise `EQUALS` returns `NIL` as *result*.

If the optional argument *recursive-p* is `T`, then `EQUALS` *may* recurse down the "structure" of *a* and *b*. The description of each known method contains the relevant information about its *recursive-p* dependent behavior.

`EQUALS` provides some default behavior, but it is intended mostly as a hook for users. As such, it is allowed to add keyword arguments to user-defined `EQUALS` methods, as the `&key` and `&allow-other-keys` lambda-list markers imply.

Known Method Descriptions: The following are the descriptions of `EQUALS` known methods; unless explicitely mentioned *recursive-p* and *keys* are to be considered as `ignore`d.

`EQUALS` (*a* `T`) (*b* `T`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

The default behavior for two objects *a* and *b* of type/class `T` is to fall back on the function `eq`[2].

`EQUALS` (*a* `number`) (*b* `number`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

[2] Falling back onto `eq` has a few justifications.
A Java (or C++) programmer may find the connection more immediate, as this would make the behavior of `EQUALS` similar to the default `java.lang.Object equals` method.
Another reason to fall back on `eq` would be to make the behavior between the treatment of `structure-object`s and `standard-object`s uniform.

The default behavior for two objects *a* and *b* of type/class `number` is to bypass `equalp` and to fall back directly on the function `=`[3].

`EQUALS` (*a* `cons`) (*b* `cons`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

The behavior for two objects *a* and *b* of type/class `cons` depends on the value of *recursive-p*: if the value is non-`NIL` then the `EQUALS` calls function `tree-equal` with itself as `:test`; otherwise, `EQUALS` calls `eq`.

`EQUALS` (*a* `character`) (*b* `character`)
`&rest` *keys* `&key` *recursive-p* (*case-sensitive-p* `T`) `&allow-other-keys`

The behavior for two `character` objects depends on the value of the keyword parameter *case-sensitive-p*: if non-`NIL` (the default) then the test uses `char=`, otherwise `char-equal`.

`EQUALS` (*a* `string`) (*b* `string`)
`&rest` *keys* `&key` *recursive-p* (*case-sensitive-p* `T`) `&allow-other-keys`

The behavior for two `string` objects depends on the value of the keyword parameter *case-sensitive-p*: if non-`NIL` (the default) then the test uses `string=`, otherwise `string-equal`.

`EQUALS` (*a* `array`) (*b* `array`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

The default behavior for two objects *a* and *b* of type/class `array` is to call `EQUALS` element-wise, as per `equalp`. The *recursive-p* argument is passed unmodified in each element-wise call to `EQUALS`.

Example: the following may be an implementation of `EQUALS` on `array`s (modulo "active elements", fill-pointers and other details).

```
(defmethod EQUALS ((a array) (b array)
                   &rest keys
                   &key recursive-p &allow-other-keys)
  (let ((a-ts (array-total-size a))
        (b-ts (array-total-size b))
       )
    (when (/= a-ts b-ts) (return-from equiv nil))
    (loop for i from 0 below a-ts
          always (apply #'EQUALS
                        (row-major-aref a i)
                        (row-major-aref b i)
                        keys))
     ))
```

[3]It may be worthwhile to add a `:epsilon` keyword describing the tolerance of the equality test and other keys describing the "nearing" direction (**Note:** must check the correct numerics terminology.)

`EQUALS` (*a* `hash-table`) (*a* `hash-table`)
`&rest` *keys* `&key` *recursive-p* (*by-key* `t`) (*by-value* `t`) (*check-properties* `t`) `&allow-other-keys`

The `EQUALS` default behaviour for two `hash-table` object is the following. If *a* and *b* are `eq`, the *result* is `T`. Otherwise, first it is checked that the two hash-tables have the same number of entries, then three tests are performed "in parallel".

1. if *by-key* is non-`NIL` then the *keys* of the *a* and *b* are compared with `EQUALS` (with *recursive-p* passed as-is). The semantics of this test are as if the following code were executed

   ```
   (loop for k1 in (ht-keys a)
         for k2 in (ht-keys b)
         always (equiv k1 k2 recursive-p))
   ```

   If *by-key* is `NIL`, the subtest is true.

2. if *by-value* is non-`NIL` then the *values* of the *a* and *b* are compared with `EQUALS` (with *recursive-p* passed as-is). The semantics of this test are as if the following code were executed

   ```
   (loop for v1 in (ht-values a)
         for v2 in (ht-values b)
         always (equiv v1 v2 recursive-p))
   ```

   If *by-value* is `NIL`, the subtest is true.

3. it *check-properties* is non-`NIL` then all the standard hash-table properties are checked for equality using `eql`, `=`, or `null` as needed. Implementation-dependent properties are checked accordingly.

   If *check-properties* is `NIL`, the subtest is true.

result is computed as the conjunction of the previous subtests.

Synonyms: the name `EQUALS` is Latin for "equal"; of course, this may not be the best name for a Common Lisp function; some synonims may be the symbol `==` or `EQUIV`. In general, synonyms should be defined by setting their `fdefinition` to `(symbol-function 'aequalis)`.

Examples:

```
cl-prompt> (EQUALS 42 42)
T

cl-prompt> (EQUALS 42 'a)
NIL

cl-prompt> (EQUALS "abc" "abc")
T
```

```
cl-prompt> (EQUALS (make-hash-table) (make-hash-table))
T

cl-prompt> (EQUALS "FOO" "Foo")
T

cl-prompt> (EQUALS "FOO" "Foo" :case-sensitive-p nil)
NIL

cl-prompt> (defstruct foo a s d)
FOO

cl-prompt> (EQUALS (make-foo :a 42 :d "a string")
                   (make-foo :a 42 :d "a string"))
NIL

cl-prompt> (EQUALS (make-foo :a 42 :d "a bar")
                   (make-foo :a 42 :d "a baz"))
NIL

cl-prompt> (defmethod EQUALS ((a foo) (b foo)
                              &key (recursive-p t) &allow-other-keys)
               (declare (ignore recursive-p))
               (or (eq a b)
                   (= (foo-a a) (foo-a b))))
#<STANDARD METHOD EQUALS (FOO FOO)>

cl-prompt> (EQUALS (make-foo :a 42 :d "a string")
                   (make-foo :a 42 :d "a string"))
T

cl-prompt> (EQUALS (make-foo :a 42 :d "a string")
                   (make-foo :a 42 :d "a String")
                   :case-sensitive-p t)
T
```

Side Effects:

None.

Affected By:

TBD.

Exceptional Situations:

TBD.

See Also:

`HASH-CODE`.

3.2 Standard Generic Function `COMPARE`

Syntax:

`COMPARE` *a b* `&rest` *keys* `&key` *recursive-p* `&allow-other-keys` ⇒ *result*

Known Method Signatures:

`COMPARE` (*a* `T`) (*b* `T`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

`COMPARE` (*a* `number`) (*b* `number`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

`COMPARE` (*a* `character`) (*b* `character`)
`&rest` *keys* `&key` *recursive-p* (*case-sensitive-p* NIL) `&allow-other-keys`

`COMPARE` (*a* `string`) (*b* `string`)
`&rest` *keys* `&key` *recursive-p* (*case-sensitive-p* NIL) `&allow-other-keys`

`COMPARE` (*a* `symbol`) (*b* `symbol`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

Arguments and Values:

a b − Common Lisp objects.

recursive-p − a *generalized boolean*; default is `NIL`.

result − a `symbol` of type `(member < > = /=)`.

keys − a `list` (as per the usual behavior).

case-sensitive-p − a *generalized boolean*; default is `T`.

Description:

The generic function `COMPARE` defines methods to test the *ordering* of two objects *a* and *b*, if such order exists. The *result* value returned by `COMPARE` is one of the four symbols: `<`, `>`, `=`, or `/=`. The `COMPARE` function returns `/=` as *result* by default; thus it can represent *partial orders* among objects. The equality tests should be coherent with what the generic function `EQUALS` does.

If the optional argument *recursive-p* is `T`, then `COMPARE` *may* recurse down the "structure" of *a* and *b*. The description of each known method contains the relevant information about its *recursive-p* dependent behavior.

Known Methods Descriptions:

`COMPARE` (*a* `T`) (*b* `T`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

The default behavior for `COMPARE` when applied to two objects *a* and *b* of "generic" type/class is to return the symbol `/=` as *result*. The intended meaning is to signal the fact that no ordering relation is known among them.

`COMPARE` (*a* `number`) (*b* `number`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

The default behavior for two objects *a* and *b* of type/class `number` is to compute *result* according to the standard predicates `<`, `>`, and `=`[4].

`COMPARE` (*a* `character`) (*b* `character`)
`&rest` *keys* `&key` *recursive-p* (*case-sensitive-p* NIL) `&allow-other-keys`

The behavior for two `character` objects depends on the value of the keyword parameter *case-sensitive-p*: if non-`NIL` (the default) then the test uses `char<`, `char>`, and `char=` to compute *result*; otherwise it uses `char-lessp`, `char-greaterp`, and `char-equal`.

`COMPARE` (*a* `string`) (*b* `string`)
`&rest` *keys* `&key` *recursive-p* (*case-sensitive-p* NIL) `&allow-other-keys`

The behavior for two `string` objects depends on the value of the keyword parameter *case-sensitive-p*: if non-`NIL` (the default) then the test uses `string<`, `string>`, and `string=` to compute *result*; otherwise it uses `string-lessp`, `string-greaterp`, and `string-equal`.

`COMPARE` (*a* `symbol`) (*b* `symbol`)
`&rest` *keys* `&key` *recursive-p* `&allow-other-keys`

When called with two `symbol`s, the method returns `=` if *a* and *b* are `eq`, otherwise it returns `/=`.

Examples:

```
cl-prompt> (COMPARE 42 0)
>

cl-prompt> (COMPARE 42 1024)
<

cl-prompt> (COMPARE pi pi)
=

cl-prompt> (COMPARE pi 3.0s0)
```

[4]Of course, the partition between `real` and `complex` must be taken into consideration.

```
>

cl-prompt> (COMPARE 'this-symbol 'this-symbol)
=

cl-prompt> (COMPARE 'this-symbol 'that-symbol)
/=

cl-prompt> (COMPARE '(q w e r t y) '(q w e r t y))
=

cl-prompt> (COMPARE #(q w e r t y) #(q w e r t y 42))
/=

cl-prompt> (COMPARE "asd" "asd")
=

cl-prompt> (COMPARE "asd" "ASD")
>

cl-prompt> (COMPARE "asd" "ASD" :case-sensitive-p nil)
=

cl-prompt> (defstruct foo a s d)
FOO

cl-prompt> (COMPARE (make-foo :a 42) (make-foo :a 42))
/=

cl-prompt> (defmethod COMPARE ((a foo) (b foo)
                               &rest keys
                               &key recursive-p
                               &allow-other-keys)
             (let ((d-r (apply #'COMPARE (foo-d a) (foo-d b) keys))
                   (a-r (apply #'COMPARE (foo-a a) (foo-a b) keys))
                  )
               (if (eq d-r a-r) d-r '/=)))
#<STANDARD METHOD COMPARE (FOO FOO)>

cl-prompt> (COMPARE (make-foo :a 0 :d "I am a FOO")
                    (make-foo :a 42 :d "I am a foo"))
/=

cl-prompt> (COMPARE (make-foo :a 0 :d "I am a FOO")
                    (make-foo :a 42 :d "I am a foo")
                    :case-sensitive-p nil)
```

```
<

cl-prompt> (COMPARE (make-array 3 :initial-element 0)
                    (vector 1 2 42))

Error: Uncomparable objects #(0 0 0) and #(1 2 42).
```

3.3 Functions LT, LTE, GT, and GTE

Syntax:

`LT` *a b* `&optional` *recursive-p*
`&rest` *keys* `&key &allow-other-keys` ⇒ *result*

`LTE` *a b* `&optional` *recursive-p*
`&rest` *keys* `&key &allow-other-keys` ⇒ *result*

`GT` *a b* `&optional` *recursive-p*
`&rest` *keys* `&key &allow-other-keys` ⇒ *result*

`GTE` *a b* `&optional` *recursive-p*
`&rest` *keys* `&key &allow-other-keys` ⇒ *result*

Synonyms:

the full-name synonyms `lessp`, `not-greaterp`, `greaterp`, and `not-lessp` are provided s well. Their implementation should be based on setting the relevant `fdefinition`.

Description:

The functions `LT`, `LTE`, `GT`, and `GTE` are shorthands for calls to `COMPARE`. Each one calls `COMPARE` as

```
(apply #'compare a b recursive-p keys)
```

The appropriate result is returned when `COMPARE`, on its turn, returns `<`, `>`, or `=`. If `COMPARE` returns `/=`, then no ordering relation can be established, and the functions `LT`, `LTE`, `GT`, and `GTE` signal an error[5].

Examples:

```
cl-prompt> (lt 42 0)
NIL

cl-prompt> (lt 42 1024)
T

cl-prompt> (gte pi pi)
T

cl-prompt> (greaterp pi 3.0s0)
T

cl-prompt> (lt "asd" "asd")
```

[5] Decide which error.

```
NIL

cl-prompt> (lte "asd" "ASD")
NIL

cl-prompt> (lte "asd" "ASD" t :case-sensitive-p nil)
T

cl-prompt> (defstruct foo a s d)
FOO

cl-prompt> (defmethod COMPARE ((a foo) (b foo)
                               &rest keys
                               &key recursive-p &allow-other-keys)
             (let ((d-r (apply #'COMPARE (foo-d a) (foo-d b) keys))
                   (a-r (apply #'COMPARE (foo-a a) (foo-a b) keys))
                  )
               (if (eq d-r a-r) d-r '/=)))
#<STANDARD METHOD COMPARE (FOO FOO)>

cl-prompt> (lte (make-foo :a 0 :d "I am a FOO")
                (make-foo :a 42 :d "I am a foo"))

Error: Uncomparable objects
       #S(FOO :a 0 :s NIL :d "I am a FOO") and
       #S(FOO :a 0 :s NIL :d "I am a foo")

cl-prompt> (lte (make-foo :a 0 :d "I am a FOO")
                (make-foo :a 42 :d "I am a foo")
                :case-sensitive-p nil)
<

cl-prompt> (lte (make-array 3 :initial-element 0)
                (vector 1 2 42))

Error: Uncomparable objects #(0 0 0) and #(1 2 42).
```

Side Effects:

None.

Affected By:

TBD.

Exceptional Situations:

An "error" is signalled when called on a pair of objects for which no predicate is defined (which is like what happens for undefined methods).

3.4 Standard Generic Function HASH-CODE

Syntax:

HASH-CODE *a* ⇒ *result*

Known Method Signatures:

HASH-CODE (*a* T)

Arguments and Values:

a − a Common Lisp object.

result − a positive fixnum in the range `(mod array-total-size-limit)`.

Description:

The HASH-CODE generic function is provided as a companion to EQUALS for the benefit of those Common Lisp implementations that provide a handle on the inner working of hash tables (usually in the form of an extra `:sxhash` or `:hash-function` keyword argument to `make-hash-table`), or for bottom-up hash table implementations.

HASH-CODE is modeled after the JAVA `hashCode()` method of `java.lang.Object`. The same description applies almost unchanged.
The general contract of HASH-CODE is the following.

- Whenever it is invoked on the same object more than once during an a Common Lisp session, the HASH-CODE generic function must consistently return the same fixnum, provided no information used in EQUALS comparisons on the object *a* is modified. This integer need not remain consistent from one Common Lisp session to another.
- If two objects are equal according to the EQUALS generic predicate, then calling the HASH-CODE generic function on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the EQUALS generic predicate, then calling the HASH-CODE generic function on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

Known Method Descriptions:

HASH-CODE (*a* T)

The only method defined for HASH-CODE is the default one, which simply resolves to a call to `sxhash`. An implementation of the method can be:

```
(defmethod HASH-CODE ((a T)) (sxhash a))
```

Examples:

None.

Notes:

The implementation of `HASH-CODE` should coordinate with that of `EQUALS`. In particular, Section 18.1.2 "Modifying Hash Table Keys" of [2] and the definiton of `sxhash` in the same document should be taken into consideration.

Side Effects:

None.

Affected By:

The actual implementation of the `EQUALS` methods.

Exceptional Situations:

TBD.

References

[1] *The Best of Intentions: EQUAL Rights – and Wrongs – in Lisp*, published online at `http://www.nhplace.com/kent/PS/EQUAL.html`, 1997.

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

A License