Generic Equality and Comparison for Common Lisp

Marco Antoniotti
mantoniotti at common-lisp.net

February 15, 2011

Abstract

This document presents new generic functions for Common Lisp that provide user hooks for extensible *equality* and *comparison tests*. 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.

2 Description

The the proposal describes the *equality* and *comparison* operators. The *equality* operator is called `AEQUALIS` 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 *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 `AEQUALIS`

Syntax:

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

Known Method Signatures:

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

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

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

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

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

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

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

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`.

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

Description:

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

If the optional argument *recursive-p* is `T`, then `AEQUALIS` *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.

`AEQUALIS` 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 `AEQUALIS` methods, as the `&key` and `&allow-other-keys` lambda-list markers imply.

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

`AEQUALIS` (*a* `T`) (*b* `T`) `&optional` *recursive-p*
`&rest` *keys* `&key &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].

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

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].

`AEQUALIS` (*a* `cons`) (*b* `cons`) `&optional` *recursive-p*
`&rest` *keys* `&key &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 `AEQUALIS` calls function `tree-equal` with itself as `:test`; otherwise, `AEQUALIS` calls `eq`.

`AEQUALIS` (*a* `character`) (*b* `character`) `&optional` *recursive-p*
`&rest` *keys* `&key` (*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`.

[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 `AEQUALIS` 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-objects` and `standard-objects` uniform.

[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.)

`AEQUALIS` (*a* `string`) (*b* `string`) `&optional` *recursive-p*
`&rest` *keys* `&key` (*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`.

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

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

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

```
(defmethod AEQUALIS ((a array) (b array)
                     &optional recursive-p
                     &rest keys
                     &key &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 #'AEQUALIS
                        (row-major-aref a i)
                        (row-major-aref b i)
                        recursive-p
                        keys))
      ))
```

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

The `AEQUALIS` 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 `AEQUALIS` (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 `AEQUALIS` (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 `AEQUALIS` 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> (AEQUALIS 42 42)
T

cl-prompt> (AEQUALIS 42 'a)
NIL

cl-prompt> (AEQUALIS "abc" "abc")
T

cl-prompt> (AEQUALIS (make-hash-table) (make-hash-table))
T

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

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

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

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

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

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

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

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

Side Effects:

None.

Affected By:

TBD.

Exceptional Situations:

TBD.

3.2 Standard Generic Function `COMPARE`

Syntax:

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

Known Method Signatures:

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

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

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

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

COMPARE (*a* symbol) (*b* symbol) &optional *recursive-p*
&rest *keys* &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 AEQUALIS 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) &optional *recursive-p*
&rest *keys* &key &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`) `&optional` *recursive-p* `&rest` *keys* `&key` `&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`) `&optional` *recursive-p* `&rest` *keys* `&key` (*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`) `&optional` *recursive-p* `&rest` *keys* `&key` (*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`) `&optional` *recursive-p* `&rest` *keys* `&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)
>

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

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

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

```
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" t :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)
                           &optional recursive-p
                           &rest keys
                           &key &allow-other-keys)
              (let ((d-r (apply #’COMPARE (foo-d a) (foo-d b)
                                recursive-p
                                keys))
                    (a-r (apply #’COMPARE (foo-a a) (foo-a b)
                                recursive-p
                                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")
                t
                :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
```

[5] Decide which error.

```
cl-prompt> (greaterp pi 3.0s0)
T

cl-prompt> (lt "asd" "asd")
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)
                           &optional recursive-p
                           &rest keys
                           &key &allow-other-keys)
              (let ((d-r (apply #’COMPARE (foo-d a) (foo-d b)
                                recursive-p
                                keys))
                    (a-r (apply #’COMPARE (foo-a a) (foo-a b)
                                recursive-p
                                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> (COMPARE (make-foo :a 0 :d "I am a FOO")
                (make-foo :a 42 :d "I am a foo")
                t
                :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).

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