OBJ3's Built-ins

Timothy Winkler and José Meseguer
SRI International, Menlo Park CA 94025[*]

August 22, 2001

1 Introduction

OBJ3 provides two ways to take advantage of the (Common-)Lisp underlying its implementation: *built-in sorts* and *built-in righthand sides* for rules; we call rules with such built-in righthand sides *built-in rules.*

Built-in sorts are sorts whose elements are constants represented by Lisp values. General mechanisms are provided for reading, printing, creating Lisp representations for, and testing sort membership for constants of these sorts. In general, built-in sorts can be used in any context a non-built-in sort can, although a constant of a built-in sort cannot be the lefthand side of a rule.

The built-in rules come in two varieties, a simplified version that makes writing rules for functions defined on built-in sorts easy, and a general kind that allows arbitrary actions on the redex to be specified. However, to take full advantage of this latter type of rule, one must be familiar with the internal term representation of OBJ and the implementation functions for manipulating this representation. Built-in rules can be used wherever an ordinary rule can be.

2 Built-in Sorts

Built-in sorts consist of an indefinitely large number of constants. For example. a version of NATS with a built-in sort Nat is equivalent to an idealized non-built-in version of the form

```
obj NATS is
  sort Nat .
  ops 0 1 2 3 4 5 6 7 8 9 10 11 12 13 ... : -> Nat .
  op _+_ : Nat Nat -> Nat .
  ...
endo
```

in which an infinite number of constants have been declared. (The name NATS was chosen to avoid a clash with the predefined object NAT.) Other examples of useful built-in sorts are floating point numbers, identifiers, strings, and arrays.

Constants in a built-in sort have an associated Lisp representation. Such a built-in sort is introduced by a declaration of the form

bsort $\langle$*SortId*$\rangle$ ($\langle$*Token-Predicate*$\rangle$ $\langle$*Creator*$\rangle$ $\langle$*Printer*$\rangle$ $\langle$*Sort-Predicate*$\rangle$) .

Which gives the name of the sort, two Lisp functions used for reading, a function for printing constants of the sort, and a predicate that can be used to test whether a Lisp value represents a constant of the given sort. A sort declaration of this kind can occur wherever an ordinary declaration of a sort can occur.

When an OBJ expression is read, it is first lexically analyzed into a sequence of tokens which are either single character symbols, such as "(" and "]", or are sequences of characters delimited by these single character symbols or spaces. Internally such tokens are represented by Lisp strings. The representation of the token "37" is the Lisp string `"37"` of length two.

[*]This documented was edited for clarity by Joseph Kiniry to be included with the 2.06 release of OBJ3 (June, 2000). Any errors herein are his alone. Please provide feedback to `obj3-feedback@kindsoftware.com`.

- ⟨*Token-Predicate*⟩ is a Lisp predicate that can be applied to an input token (a Lisp string) to determine if the token is a representation of a value in the built-in sort. (It is applied by funcall.) E.g., for NATS, `"37"` should result in true and `"A+B"` in false. With this mechanism the syntactic representation of a built-in constant can only be a single token.
- ⟨*Creator*⟩ is a Lisp function that will map a token (a Lisp string) to a Lisp representation for that token as a built-in constant. The Lisp function `read-from-string` is very useful as a creator function for built-in sorts that correspond directly to Lisp types. E.g., `"37"` should be mapped to the Lisp value `37`.
- ⟨*Printer*⟩ is a Lisp function that will print out the desired external representation of the internal Lisp value representing one of the built-in sort constants. The Lisp function `prin1` is very useful as a ⟨*Printer*⟩ function for printing out values that correspond directly to Lisp types. E.g., 37 should be printed by printing the digit `3` followed by the digit `7`. Since the user can define the printer function to meet particular needs, there is no assumption that this function is an inverse to the ⟨*Creator*⟩ function. Indeed, the syntactic representation of a built-in constant may involve many tokens, but then this representation cannot be read in as a built-in constant.
- ⟨*Sort-Predicate*⟩ should be a Lisp predicate that is true only for Lisp values that are representations of constants in the built-in sort. E.g., `3` should be considered to be in sort Nat and and `-3` should not. The purpose and use of this predicate will be discussed further below.

For NATS we might have the specific declaration

```
obj NATS is
  bsort Nat (obj_NATS$is_Nat_token obj_NATS$create_Nat
               obj_NATS$print_Nat obj_NATS$is_Nat) .
endo
```

where the functions referred to have these definitions

```
(defun obj_NATS$is_Nat_token (token)
  (every #'digit-char-p token))
(defun obj_NATS$create_Nat (token) (read-from-string token))
(defun obj_NATS$print_Nat (x) (prin1 x))
(defun obj_NATS$is_Nat (x) (and (integerp x) (<= 0 x)))
```

With the above definition of the object NATS one can use the natural number constants. E.g.,

```
OBJ> red 100 .
reduce in NATS : 100
rewrites: 0
result Nat: 100
```

As it stands, since only a built-in sort has been introduced, and no associated functions have been defined, this object is not very useful.

Note: a current implementation restriction does not allow a built-in constant to be the lefthand side of a rule. Built-in constants are always treated as being in reduced form (application of rules would never be attempted).

2.1 Subsorts of Built-in Sorts

It is possible for a built-in sort to be a subsort of another built-in sort, but a non-built-in sort cannot be a subsort of a built-in sort. For the sort of newly created built-in constants to be properly assigned, a sort predicate must be provided for each built-in sort. An example of this will later be seen in a version of the rational numbers using Common Lisp rationals.

When there are built-in subsorts of the sort of a newly created built-in constant, then the sort that is assigned to the constant is determined by scanning the list of subsorts, applying the sort predicates to the Lisp value to determine if it lies in the corresponding subsort, and choosing the lowest acceptable sort as the sort of the constant. It is assumed that there is always a unique lowest sort. It is critical only that the sort predicate for a built-in sort should be false for values that are in supersorts of the built-in sort. It is not necessary for it to be false for constants in subsorts of the given sort.

If there is no enclosing built-in supersort, since the ⟨*Sort-Predicate*⟩ function will only be called for built-in constants of that sort (if it is called at all), then it can be constantly true, and have a definition like

```
(defun obj_NATS$is_Nat (x) t)
```

This will not affect the operational behavior of OBJ in this case. However, it is better for the predicate to be exact in order to allow the easy incorporation of the sort a supersort.

3 Built-in Rules

Built-in rules provide a way of using Lisp expressions to perform computations. This is essential for the usefulness of built-in sorts, but can also be used for non-built-in data. These rules are either of a special *simple* form or are *general.*

Simple built-in rules can be unconditional or conditional with syntax

```
bq ⟨Term⟩ = ⟨Lisp Expression⟩ . |
cbq ⟨Term⟩ = ⟨Lisp Expression⟩ if ⟨BoolTerm⟩ .
```

The key requirement for simple built-in rules is that *all the variables appearing in the lefthand side must have sorts that are built-in sorts.*

The lefthand side of the rule is matched against terms in exactly the usual fashion; also, in the conditional case, the condition is just an OBJ term and is treated in exactly the same way as a condition in a non-built-in rule. If a match for the variables of the lefthand side is found, where each variable is matched to a built-in constant and which satisfies the condition if the built-in rule is conditional, then the righthand side of the equation is evaluated in a Lisp environment where Lisp variables with names corresponding to the OBJ variables (as usual in Common Lisp, the case, upper or lower, of variables in the Lisp expression is ignored) are bound to the Lisp value corresponding to the built-in constants to which they were matched. Since the variables must match constants of their associated built-in sorts, this forces a bottom-up evaluation strategy regardless of the strategy specified for the operator. The sort of the lefthand side will usually be a built-in sort and the LISP value of the righthand side of the rule will be automatically converted to a built-in constant of that sort. If the sort of the lefthand side is not a built-in sort, then, with one exception that will be mentioned next, the value of the righthand side should be a Lisp representation of a term of that sort (or a subsort of that sort). A special case is that, if the sort of the lefthand side is Bool, then the value of the righthand side Lisp expression can be any Lisp value which will be converted to a Boolean value by mapping `nil` to **false** and all other Lisp values to **true**. For this case, a special conversion is performed which makes it very easy to define predicates.

As an example, consider

```
obj NATS is
  bsort Nat (obj_NATS$is_Nat_token obj_NATS$create_Nat
             obj_NATS$print_Nat obj_NATS$is_Nat) .
  op _+_ : Nat Nat -> Nat .
  vars M N : Nat .
  bq M + N = (+ M N) .
endo
```

We can then do the following reduction.

```
OBJ> red 123 + 321 .
reduce in NATS : 123 + 321
rewrites: 1
result Nat: 444
```

Since the matching of the lefthand side is done in the usual fashion, the operators appearing in the lefthand side may even be associative and commutative.

The *general* form of a built-in rule has the following syntax

> `beq` ⟨*Term*⟩ `=` ⟨*Lisp Expression*⟩ `.` |
> `cbeq` ⟨*Term*⟩ `=` ⟨*Lisp Expression*⟩ `if` ⟨*BoolTerm*⟩ `.`

where now the variables in the lefthand side can have arbitrary sorts. The lefthand side and condition are treated as usual.

The process of applying the rule is a bit different in this case. The lefthand side is matched as usual creating the correspondence between variables in the lefthand side and subterms of the term being rewritten. The righthand side is evaluated in an environment where Lisp variables with names corresponding to the OBJ variables (case is ignored) are bound to the internal OBJ3 representation of the terms matched by the variables. The Lisp value of the righthand side is expected to be an internal OBJ3 representation of a term which then destructively replaces the top-level structure of the term matched. An exception is that, if the Lisp code evaluates the expression `(obj$rewrite_fail)`, then the rewrite is aborted and the term is left unchanged. (This has the effect of making the rule conditional in an implicit way; the condition is checked in the Lisp code for the righthand side.) An additional feature is that the righthand side is evaluated in an environment where `module` is bound to the module that the rule comes from. This last feature is necessary to correctly treat general built-in rules in instances of parameterized modules.

A simple example is

```
obj NATS is
  bsort Nat (obj_NATS$is_Nat_token obj_NATS$create_Nat
             obj_NATS$print_Nat obj_NATS$is_Nat) .
  op _+_ : Nat Nat -> Nat .
  vars M N : Nat .
  bq M + N = (+ M N) .
  op print _ : Nat -> Nat .
  beq print M = (progn (princ " = ") (term$print M) (terpri) M) .
endo
```

This definition provides a function, `print`, that is an identity function that has the side-effect of printing the value of its argument preceded by the "=" sign. A simple example of the the use of `print` is:

```
OBJ> red (print (3 + 2)) + 4 .
reduce in NATS : print (3 + 2) + 4
 = 5
rewrites: 3
result Nat: 9
```

The line containing "`= 5`" is the output produced by the use of `print`. Such printing functions that are an identity function from the point of view of the rewriting are actually useful. (Typically one may want to add an extra argument that provides a label for the output.) General built-in rules can be written to perform arbitrary transformations on a term using any of the functions defined in the OBJ3 implementation. Thus it is useful to be familiar with the functions provided by the implementation when writing such general built-in rules. Some of those basic functions will be discussed below.

Often is is useful to initialize some Lisp variables after certain OBJ objects are created. This can be done using `eval` or `ev`. There are examples of this in the OBJ3 standard prelude.

In general, the module that the rules are associated with may be an instance of a parameterized module. In this case, it is necessary to write the rules so that the extra parameter `module` is used to create structures within that module.

When locating the correct instance of an operator one must first determine its module, then the sorts of its arguments and result, and then its name. In the case where there are no ambiguities, some simpler functions can be used, e.g., find an operator based only on its name. Functions that are useful for the general built-in rules include the following (note these are all Lisp functions from the OBJ3 implementation).

The Lisp functions will be described, in part, by giving declarations similar to OBJ operator declarations. Of course these need to be interpreted as informal descriptions of Lisp functions that may have side-effects and which manipulate particular Lisp representations of the values given as arguments.

The sorts that will be referred to are:

- Bool, NzNat, Lisp-Value
  Bool, NzNat, and Lisp-Value are names for the related standard LISP types.

- Sort-Name
  A Sort-Name is a Lisp string naming a sort.

- Op-Name
  An Op-Name is a Lisp list of the tokens, represented as Lisp strings, that constitute the name of the operator. For example, the name of `_+_ :  Nat Nat -> Nat` is `("_" "+" "_")`.

- Sort-Order
  A Sort-Order is a representation of a partial order on the sorts.

- Sort, Operator, Term, Module, Module-Expression
  Sort, Operator, Term, Module, and Module-Expression correspond to the Lisp representations of these sorts. Values of the sorts Sort, Operator, Term, and Module are composite objects with many components, some of which are likely not to be of interest here. For these sorts, functions selecting the interesting features of the values are indicated below.

- SortSet
  SortSet is a set of sorts represented by a list.

- LIST[Term], LIST[Sort]
  LIST[-] indicates that the values so described will be Lisp lists of the specified sort.

Following is a list of functions that are useful in writing term manipulations functions.

- `modexp_eval$eval` : Module-Expression →Module
  The argument can be the name of a specific named module, e.g. `"INT"`. This can be used to find specific named modules.

- `sort$is_built_in` : Sort →Bool
  This predicate decides whether the sort given is a built-in sort.

- `module$sort_order` : Module →Sort-Order
  This selector provides access to the sort order for the given module, i.e. the representation of the sort structure.

- `sort_order$is_included_in` : Sort-Order Sort Sort →Bool
  This predicate decides if the first sort is a subsort of the second in the given sort order.

- `sort_order$is_strictly_included_in` : Sort-Order Sort Sort →Bool
  Same as above but excludes the case when two sorts are equal.

- `sort_order$lower_sorts` : Sort-Order Sort →SortSet
  This function produces a list of the sorts lower than a given sort in the given sort order.

- `mod_eval$$find_sort_in` : Module Sort-Name →Sort
  This function can be used to find the named sort in the given module. A typical sort name is "Int".

- `sort$name` : Sort →Sort-Name
  This selector provides the name of a given sort.
- `operator$name` : Operator →Op-Name
  This selector provides the name of the given operator.
- `operator$is_same_operator` : Operator Operator →Bool
  This predicate decides if the two operators are the same operator.
- `operator$arity` : Operator →LIST[Sort]
  This selector provides the arity of the given operator as a list of sorts, which may be `nil`.
- `operator$coarity` : Operator →Sort
  This selector provides the co-arity of the given operator.
- `mod_eval$$find_operator_in` : Module Op-Name LIST[Sort] Sort →Operator
  This function locates the operator with the given name, arity (list of sorts) and coarity, or returns `nil` if there is none such.
- `mod_eval$$find_operator_named_in` : Module Op-Name →Operator
  This function attempts to locate an operator purely based on its name.
- `term$is_var` : Term →Bool
  This predicate decides if a term is a variable. It may be that the terms that you will be manipulating will primarily be ground terms, but, in general, it is preferable to consider the case of variables in definitions of functions.
- `term$is_constant` : Term →Bool
  This predicate decides if a term is a constant.
- `term$head` : Term →Operator
  This function produces the operator that is the head operator of a non-variable term. It is an error to apply this function to a term that is a variable.
- `term$subterms` : Term →LIST[Term]
  This function produces the list of top-level subterms of the given term.
- `term$make_term` : Operator LIST[Term] →Term
  This function creates a new term with the given head operator and list of arguments.
- `term$make_term_with_sort_check` : Operator LIST[Term] →Term
  This function is similar to the last, but may replace the operator with a lower operator in the case of overloading. If there is a lower overloaded operator whose arity fits the sorts of the given arguments this operator will be used instead of the given operator.
- `term$arg_n` : Term NzNat →Term
  This function gives easy access to the n-th (counting from 1) top-level argument of the given term.
- `term$sort` : Term →Sort
  This function computes the sort of a term whether it is a variable or not.
- `term$is_reduced` : Term →Bool
  This function checks whether or not the term has been marked as fully reduced. This flag is updated by side-effect.
- `term$!replace` : Term Term →Term
  The Lisp representation for the first argument term is destructively altered in such a way that it will appear to have the same term structure as the second term argument. The altered representation of the first term is returned.

- `term$!update_lowest_parse_on_top` : Term →Term
  This will update the sort of the term, e.g. in the case where a subterm has been altered and now has a lower sort.
- `term$retract_if_needed` : Sort-Order Term Sort →Term
  This function either returns the term, or a retract applied to the term depending on whether the sort of the term is included in the given sort or not.
- `term$is_built_in_constant` : Term →Bool
  This predicate decides if the term is a built-in constant or not.
- `term$similar` : Term Term →Bool
  Tests if the two terms have the same term structure without taking attributes into account.
- `term$equational_equal` : Term →Bool
  Tests if the terms have the equivalent structure taking attributes into account.
- `term$make_built_in_constant` : Sort Lisp-Value →Term
  This function creates a *term* which is a built-in constant for the given built-in sort and Lisp value. The sort predicate for the built-in sort is not applied.
- `term$make_built_in_constant_with_sort_check` : Sort Lisp-Value →Term
  Similar to above, but may replace the given sort by a lower sort.
- `term$built_in_value` : Term →Lisp-Value
  This function produces the Lisp value from a built-in constant.
- `obj_BOOL$is_true` : Term →Bool
  This function tests whether the term given as its argument is the constant `true`. The value is a Lisp boolean, i.e. `T` for `true` and `NIL` for `false`.
- `rew$!normalize` : Term →Term
  The is the OBJ evaluation function. The term given as an argument is reduced and is updated by side-effect as well as being returned as the value of the function.

The following functions are specific to A and AC terms.

- `term$list_assoc_subterms` : Term Operator →LIST[Term]
  This function computes the list of subterms of the given term that are on the fringe of the tree at the top of the term the nodes of which are all terms headed with the given associative operator or operators overloaded by this operator. This can be the whole term.
- `term$list_AC_subterms` : Term Operator →LIST[Term]
  Similar to the above, but for associative-commutative (AC) operators.
- `term$make_right_assoc_normal_form` : Operator LIST[Term] →Term
  This function builds a term from the given associative operator and the list of terms by building a right-associated binary tree.
- `term$make_right_assoc_normal_form_with_sort_check` : Operator LIST[Term] →Term
  Similar to the above, but may replace the operator by lower operators.

Final note: it is actually possible for subterms of righthand side to be a built-in term, with an interface provided by the prelude object BUILT-IN. The subterm will be a built-in constant of sort `Built-in`. The external syntax is "`built-in:` ⟨*Lisp*⟩". The value of the constant is a function that will map a substitution (binding of variables to terms) to a term representation (the function will be `funcall`-ed).

4 Larger Example: Rationals

As a somewhat larger example, the rational numbers could be defined using the Common Lisp representation of rationals and based on the existing predefined modules INT, NAT, and NZNAT as follows.

```
ev (progn
(defun obj_RAT$is_NzRat_token (token) nil)
(defun obj_RAT$create_NzRat (x) (read-from-string x))
(defun obj_RAT$is_Rat_token (x) nil)
(defun obj_RAT$is_NzRat (x) (and (rationalp x) (not (= 0 x))))
(defun obj_RAT$print (x)
  (if (typep x 'ratio)
    (progn
        (prin1 (numerator x))
        (princ " / ")
        (prin1 (denominator x)))
    (prin1 x)))
) .

obj RATS is
  protecting INT .
  bsort NzRat (obj_RAT$is_NzRat_token obj_RAT$create_NzRat
               obj_RAT$print obj_RAT$is_NzRat) .
  bsort Rat (obj_RAT$is_Rat_token car obj_RAT$print rationalp) .
  subsorts Int < Rat .
  subsorts NzInt < NzRat < Rat .
  op _/_ : Rat NzRat -> Rat .
  op _/_ : NzRat NzRat -> NzRat .
  op -_  : Rat -> Rat [prec 2] .
  op -_  : NzRat -> NzRat [prec 2] .
  op _+_ : Rat Rat -> Rat [assoc comm] .
  op _*_ : Rat Rat -> Rat [assoc comm] .
  op _*_ : NzRat NzRat -> NzRat [assoc comm] .
  op _-_ : Rat Rat -> Rat .
  vars R S : Rat .
  vars NS : NzRat .
  bq - R = (- R) .
  bq R / NS = (/ R NS) .
  bq R + S = (+ R S) .
  bq R * S = (* R S) .
  bq R - S = (- R S) .
jbo
```

5 Larger Example: Cells

The basic idea of this example is very simple, namely to provide a parameterized object that creates *cells* containing values of a given sort. Such cells are an abstract version of procedural variables that can be modified by side-effects or destructive assignments. Of course, this module is not functional.

```
*** obj code for cells

ev (defun set-cell-rule (i x) (setf (cadr i) x) i)

obj CELL[X :: TRIV] is
```

```
  sort Cell .
  op cell _ : Elt -> Cell .
  op new-cell _ : Elt -> Cell .
  op val _ : Cell -> Elt .
  op set _ _ : Cell Elt -> Cell .
  var I : Cell .
  var X : Elt .
  eq new-cell X = cell X .
  eq val (cell  X) = X .
  beq set I X = (set-cell-rule I X) .
endo

*** sample program using this
obj TEST is
  pr CELL[INT] .

  sort A .
  subsort Int Cell < A .
  op _|_ : A A -> A .

  op dbl _ : A -> A .
  op incr _ : A -> A .

  var U V : A .
  var C : Cell .

  eq dbl U = U | U .

  eq incr (U | V) = (incr U) | (incr V) .
  eq incr C = val (set C (1 + (val C))) .
endo

red incr (dbl (dbl (dbl (new-cell 0)))) .
*** result A: ((1 | 2) | (3 | 4)) | ((5 | 6) | (7 | 8))
```

6 Larger Example: Arrays of Integers

This provides arrays of integers that can be modified by side-effect. It might be useful for a functional program for table-lookup (side-effects only would be used for building the table).

```
ev
(defun arrayint$print (x)
  (princ "[")
  (dotimes (i (length x))
    (when (< 0 i) (princ ",")) (print$check)
    (prin1 (aref x i)))
  (princ "]"))

obj ARRAYINT is
  pr INT .
  bsort ArrayInt ((lambda (x) nil) (lambda (x) (break))
                 arrayint$print (lambda (x) t)) .

  op make-array : Nat Int -> ArrayInt .
  op length _ : ArrayInt -> Nat .
  op _[_] : ArrayInt Nat -> Int .
  op _[_] := _ : ArrayInt Nat Int -> ArrayInt .

  var A : ArrayInt .
  var I : Int .
  var N : Nat .

  bq make-array(N,I) = (make-array (list N) :initial-element I) .
  bq length(A) = (length A) .
  bq A[N] = (aref A N) .
  bq A[N] := I = (progn (setf (aref A N) I) A) .
endo

OBJ> red make-array(10,1) .
reduce in ARRAYINT : make-array(10,1)
rewrites: 1
result ArrayInt: [1,1,1,1,1,1,1,1,1,1]
OBJ> red (make-array(10,1))[5] .
reduce in ARRAYINT : make-array(10,1)[5]
rewrites: 2
result NzNat: 1
OBJ> red (make-array(10,1))[5] := 33 .
reduce in ARRAYINT : make-array(10,1)[5]:= 33
rewrites: 2
result ArrayInt: [1,1,1,1,1,33,1,1,1,1]
```

7 Larger Example: Sorting

This provides a parameterized sorting module. The parameter provides the partial order used and the sorting is done using the Lisp function `sort`. One minor interesting point is that an operator named _ `<<` _ is introduced as an alias for the parameter operator _ `<` _ simply to provide an easy way to locate the parameter operator after instantiation. This is needed because the name of a parameter operator cannot be known for an instance of the parameterized module, where such a parameter may have been mapped to an arbitrary term by the view defining the instantiation. For similar reasons, the operator _ `<<` _ as well as the other operators _ `,` _ and `empty` appearing in the parameterized SORT module below should not be renamed by a module renaming.

The parameter of our sorting module is as usual the theory of partially ordered sets, given by the theory module:

```
th POSET is
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 < E1 = false .
  cq E1 < E3 = true if E1 < E2 and E2 < E3 .
endth
```

The function `sort-list` used in the `SORT` module below has two arguments, a module (namely the given instantiation of the parameterized module `SORT`) and a list to be sorted. Its definition is as follows:

```
ev
; NOTE: sort-list will not work if any of the operators found by name,
; i.e. _<<_, empty, and _,_, below are renamed in a module renaming.
```

```
(defun sort-list (mod l)
  (let ((test (mod_eval$$find_operator_named_in
               mod '("_" "<<" "_")))
        (empty (mod_eval$$find_operator_named_in
                mod '("empty")))
        (conc (mod_eval$$find_operator_named_in
               mod '("_" "," "_"))))
  (if (eq empty (term$head l))
      l
    (let ((sorted (sort (term$list_assoc_subterms l conc)
                        #'(lambda (x y)
                            (obj_BOOL$is_true
                             (rew$!normalize
                              (term$make_term test
                                  (list x y)))))
                        )))
      (term$make_right_assoc_normal_form_with_sort_check
       conc sorted)
      ))
  ))
```

We are now ready to define the parameterized SORT module which has a built-in equation involving the sort-list function.

```
obj SORT[ORDER :: POSET] is
  sort List .
  subsort Elt < List .
  op empty : -> List .
  op _,_ : List List -> List [assoc idr: empty] .
  op sort _ : List -> List .
  op _<<_ : Elt Elt -> Bool .
  vars E1 E2 : Elt .
  eq E1 << E2 = E1 < E2 .
  var L : List .
  beq sort L = (sort-list module L) .
endo
```

Here is a sample reduction for for sorting lists of integers.

```
obj TEST is pr SORT[INT] . endo

red sort (9, 8, 7, 6, 5, 4, 3, 2, 1, 0) .
***> result List: 0,1,2,3,4,5,6,7,8,9
```