Japanese Proposal for Global Variables and
Associative Functions in Standard Prolog

Katsuhiko Nakamura and Nobukuni Kino
Japanese Prolog WG

December 1, 2005

1 Introduction

This document describes a proposal for incorporating global variables and associative functions into standard Prolog. This proposal is a result of discussions in Japanese Prolog WG.

Data types of Prolog are essentially restricted to terms and lists at the cost of compactness and the unity of a logic programming language. On the other hand, it has been commonly recognized that Prolog has two practical problems, which considerably prevent the language and logic programming from being used for broader area of information processing. One problem is that Standard Prolog does not have the means for efficient random access of data in a large working memory such as arrays or hash memories. The other problem is that pure Prolog has no global variables such as those in Lisp and C. The array functions are closely related to the global variables, for the following reasons. First, the values of both global variables and the array elements need to be updated. Secondly, the scope of the array is generally wider than that of the clause, which is the same as the scope of Prolog variables.

It is well known that the global variables in logic programs are converted to additional arguments. This conversion causes, however, inefficiency in program execution as well as difficulties in reading and writing programs, since the numbers of arguments need to be increased in large programs. Schachte [6] showed formal semantics for programs with the global variables and a method of transforming programs with the global variables into efficient logic programs.

The clause creation and destruction predicates asserta/1 and retract/1 are used in some part of global variables and array functions in Prolog. These built-in predicates, however, hardly have usual logical meaning, and are not efficient since these predicates are originally interpreter-based functions for altering existing programs. Some earlier versions of Prolog had other built-in predicates for “record database” such as recorda/2 and erase/2 for storing terms. These predicates were excluded from the standard at an early stage of Prolog standardization for the reason that these are similar to asserta/1 and retract/1 and that code and data should not be distinguished in Prolog [5].
It has been an important problem how to realize arrays efficiently without violating the single assignment rule not only in logic programming but in functional programming, since updating an array element generally requires coping the entire array. An method of realizing arrays in single assignment languages is described in [2].

Most existing Prolog implementations provide some array functions as implementation-defined extensions. There are two approaches for representing arrays. A common method is to use terms as one-dimensional arrays. Some implementations, including IF-Prolog and SICStus Prolog, employ the other approach such as extendible arrays with logarithmic access time.

Several implementations, including SWI-Prolog and GNU-Prolog, have built-in predicate setarg/3 for updating terms. A goal setarg(I, Term, Value) destructively assigns the Value to the I-th argument of Term. Tarau [8] describes the problem of this built-in predicate as follows.

Unfortunately, setarg/3 lacks a logical semantics and is implemented differently in various systems. This may be the reason why the standardization of setarg/3 can hardly reach a consensus in the predictable future.

Furthermore, destructive assignment to an argument by setarg/3 sometimes causes a fatal error that eliminates some value. This error occurs, when the value of a variable in the argument is updated. Some implementations, including BinProlog and SICStus Prolog, provide improved functions for updating arguments of terms, or improved versions of setarg/3.

BinProlog provides global variables called global logical variables and functions for accessing hash tables, as well as arrays and built-in predicates for storing terms. Each global logical variable can be used to access a hashing table by two keys and instantiated by a special built-in predicate.

2 Assignable Objects

In general, logic programming observes the single assignment rule so that no variable is destructively assigned in usual Prolog execution. On the other hand, we need to update the values in the global variables and array elements. To compromise these two contradictory conditions, we introduce a special data type called assignable objects, by which we can update the values of global variables and array elements preserving the single assignment rule.

A value is assigned to, and accessed from, the assignable object by built-in predicates. The values are backtrackable: In backtracking, an assigned value is withdrawn and the assignable object has the previous value. The assignable objects generated by a built-in predicate have a special state empty, which represents that the object has no value.

Every assignable object is represented by a special term. The form of the terms representing assignable objects is implementation-defined.
2.1 Predicates for Logical Assignment

We have the following built-in predicates for generating assignable objects and for updating the values of assignable objects.

- \texttt{assignable(-Variable)}
  The goal \texttt{assignable(X)} generates an assignable object with the value empty and returns the term representing the assignable object to \texttt{X}.

- \texttt{assignable(-Variable, +Term)}
  The goal \texttt{assignable(X, I)} generates an assignable object with the initial value \texttt{I} and returns the term representing the assignable object to \texttt{X}.

- \texttt{assign(+Assignable<Object>, +Term)}
  The goal \texttt{assign(X, T)} assigns a term \texttt{T} to an assignable object \texttt{X}: The value of the assignable object is replaced by the term \texttt{T}. On backtracking, the value of \texttt{X} returns to the previous value.

- \texttt{access(+Assignable<Object>, +Term)}
  The goal \texttt{access(X, T)} unifies the value of an assignable object \texttt{X} with a term \texttt{T}. If \texttt{X} is empty, it raises an instantiation error.

- \texttt{assign_unique(+Assignable<Object>, +Term)}
  The goal \texttt{assign_unique(X, T)} sets an assignable object \texttt{X} to have only the value of a term \texttt{T}. On backtracking, the assignable object returns to empty.

The predicate \texttt{assign_unique/2} sets a single value to the assignable object. The role of this predicate is similar to that of the cut (!) in the sense that the value of a variable can be restricted to a single value. We introduce this predicate for the economy of memory usage, since the repetitive use of assignment by the predicate \texttt{assign/2} may increase useless memory consumption.

2.2 Example

\begin{verbatim}
counter(C) :- assignable(C, 0).
count_up(C) :- access(C, I), I is I+1, assign_unique(C, I).
counter_value(C, I) :- access(C, I).
\end{verbatim}

The goal \texttt{counter(C, 0)} generates an assignable object with the initial value 0, and returns it to the variable \texttt{C}. The goal \texttt{count_up(C)} increments the value of the assignable object by one, which is the only value of the object.

2.3 Notes

1. An access to empty object by \texttt{access/2} causes an error. An option is that the access to empty object fails.

2. In many cases, the predicate \texttt{assign_unique/2} can be replaced by \texttt{assign/2} without changing the result of the program.
3. The names of the built-in predicates are subject to change. For example, `access/2` can be renamed to `draw/2` or `retrieve/2`.

4. The predicates `assign/2`, `assign_unique/2`, and `access/2` are applied not only to assignable objects but to also global variables (Section 3).

5. Input and output of assignable objects is implementation-defined. It is desirable that the output by `write/1` contains the current value of the assignable object.

2.4 A Model of Assignable Objects

An effective model of assignable objects is given by linear lists terminated with variables. Any initial value $V_0$ is represented by the list of the form $[V_0|L]$, and the empty value by a variable. The predicates `assignable/1`, `assignable/2`, `assign/2` and `access/2` are in effect equivalent to those defined by the following Prolog predicates.

```prolog
assignable(_).
assignable([L|_], 1).

assign(L, V) :- addtail(L, V).
addtail(L, V) :- var(L), !, L=[V|L1].
addtail([_|L], V) :- addtail(L, V).

access(L, V) :- findtail(L, V).
findtail([V|L], V) :- var(L), !,
                 type_error(assignable_object).
findtail([_|L], V) :- findtail(L, V).
```

Note that this method using lists as the assignable objects is not efficient, when the list changes to have many values. Note also that it is impossible to define `assign_unique/2` in standard Prolog, since it needs to destructively assign a list $[V|L]$ with the single value $V$ to the assignable object. On the other hand, we can efficiently implement not only these predicates but also `access/2` and `assign/2` by “safe” use of `setarg/3` (see Note in Section 3).

It is not specified in the standard how the assignable objects can be efficiently implemented. A possible method is the use of either special pointers to the end cells of the linear list or a cache memory for efficiently accessing the values in the ends of the lists. There would be other different approaches that do not use the linear lists.

3 Global Variables

The global variables are considered as a mapping from the set of names to the set of assignable objects. The name of a global variable is represented by an atom. The scope of a global variable is the module.
A global variable is defined by built-in predicates `global/1` and `global/2`, which are also directives.

- `global(+Atom)`
  The goal `global(Name)` generates an empty assignable object and relates the global variable `Name` with the assignable object.

- `global(+Atom, +Term)`
  The goal `global(Name, T)` generates an assignable object having an initial value `T`, and relates the global variable `Name` with the assignable object.

After a global variable `X` is defined, the values of `X` are retrieved by `access(X, V)` and are updated by `assign(X, V)` and `assign_unique(X, V)`. These built-in predicates can distinguish the global variables from assignable objects, since the global variables are atoms.

### 3.1 Examples

**Example 1.**

```prolog
reverse(X,Y) :- global(result), rev(X,[]),
            access(result, Y).

rev([],Y) :- assign(result, Y).
rev([X|X],Y) :- rev(X,Y).

?- reverse([a,b,c],Y).
Y = [c,b,a]

?- reverse(X,[a,b,c]).
X = [c,b,a]
```

**Example 2.**

```prolog
initialize :- global(symbol_list),
            assign(symbol_list, [p,q,r,s,t,u,v]).
newsymbol(Q) :- access(symbol_list, [Q|L]),
               assign(symbol_list, L).
newsymbol1(Q) :- access(symbol_list, [Q|L]),
                assign_unique(symbol_list, L).

?- initialize, repeat, newsymbol(Q), newsymbol(R).
Q = p
R = q ;
Q = p
R = q
```
?- initialize, repeat, newsymbol(Q), newsymbol(R).
   Q = p
   R = q ;
   Instantiation error

On backtracking, the goal newsymbol(Q) fails and gives Q the previous value.
The goal newsymbol(Q) also fails on backtracking, but the assignable object
returns to the empty state.

3.2 Note

The predicates for the assignable objects and the global variables are imple-
mented in K-Prolog version 5 [1], where an initial value for an assignable object
is represented as 'Term'. The following web site contains implementation
programs for predicates of logical assignment and example programs.

One of the implementation programs is portable and logical: Predicates equi-
alent to assign/2 and access/2 are implemented in Standard Prolog. A faster
version of these predicates and the implementation of assign_unique/2 use
“safe” destructive updating by setarg/3 as mentioned in Section 2.

4 Arrays and Associative Memories

An array and an associative memory, or a hash memory, are defined as a set
of assignable objects with keys, or as a mapping from a set of keys to the set
assignable objects. Each element of an array and an associative memory are
specified by a special term representing the array and the associative memory,
respectively, and an index of the array or a key of the associative memory.

4.1 Arrays

The following built-in predicates define a one-dimensional array with a size and
an initial value.

- array(-Variable, +Integer)
  The goal array(V, N) generates N assignable objects with the state empty
  and returns a special term array(N, #AS) to V, where #AS is an imple-
  mentation-dependent term representing the set of assignable objects.

- array(-Variable, +Integer, +Term)
  The goal array(V, N, I) generates N assignable objects with the initial
  value I and returns a special term array(N, #AS) to V, where #AS is
  a term representing the set of assignable objects. If the initial value is a
  variable, or a term containing variable(s), each variable is replaced by a
  unique variable in the array elements.
The array elements are updated and accessed by the following built-in predicates, which are similar to \texttt{assign/2}, \texttt{assign\_unique/2} and \texttt{access/2}. The element is specified by the second argument of the predicates.

- \texttt{assign(+Array, +Integer,+Term)}
  The goal \texttt{assign(X,K,T)} assigns a term \(T\) to the assignable object in the array \(X\) with the index \(K\), if \(1 \leq K \leq N\), where \(N\) is the size of the array. The goal raises an error, if the condition \(1 \leq K \leq N\) does not satisfied. On backtracking, the value of the assignable object returns to the previous value.

- \texttt{access(+Array, +Integer,+Term)}
  The goal \texttt{access(X,K,T)} unifies \(T\) with the value of the assignable object in the array \(X\) with the index \(K\), if \(1 \leq K \leq N\), where \(N\) is the size of the array. The goal raises an error, otherwise. If the value of the assignable object is empty, it raises an instantiation error.

- \texttt{assign\_unique(+Array, +Integer,+Term)}
  The goal \texttt{assign\_unique(X,K,T)} sets the assignable object in the array \(X\) with the index \(K\) to have only the value of a term \(T\), if \(1 \leq K \leq N\), where \(N\) is the size of the array. The goal raises an error, otherwise. On backtracking, the value of the assignable object returns to empty.

4.1.1 Implementing Arrays as Terms

We can write a Prolog program for implementing arrays. In this program, an array is represented by a term in which each argument has an assignable object.

\begin{verbatim}
array(T,N) :- functor(T, array, N), initialize(1,T,N, empty).
array(T,N,I) :- functor(T, array,N), initialize(1,T,N,I).

initialize(K,_,N,_) :- K > N, !.
initialize(K,T,N,I) :- arg(K,T,E),
  (I = empty -> assignable(E); assignable(E,I)),
  Ki is K+1, initialize(Ki,T,N,I).

assign(A,K,T) :- arg(K,T,E), assign(E,T).
assign\_unique(A,K,T) :- arg(K,T,E), assign\_unique(E,T).
access(A,K,T) :- arg(K,T,E), access(E, T).
\end{verbatim}

4.1.2 Notes

1. Many programs in conventional languages have "nested declaration" of arrays. Arrays are recursively declared every time subprocedures are called. For the nested declaration in Prolog, we need to generate arrays so that the generated array is represented by a variable.
2. A problem of using terms as one-dimensional arrays is that the size of the arrays is restricted by the maximum arity of terms, which is not sufficiently large in some implementations.

4.2 Associative Functions

The following built-in predicates define an associative memory.

- **assoc(-Variable)**
  The goal `assoc(V)` generates an associative memory of assignable objects with the empty state and returns a special term `assoc(#AS)` to V, where #AS is an implementation-dependent term representing the associative memory.

- **assoc(-Variable, +Term)**
  The goal `assoc(V, I)` generates an associative memory for assignable objects with the initial value I, and returns a special term `assoc(I, #AS)` to V, where #AS is a term representing the associative memory.

The elements in the associative memory are updated and accessed by the following built-in predicates. The element is specified by the key of the second argument of the predicates, which is a list of atoms and integers.

- **assign(+Associative_memory, +Key, +Term)**
  The goal `assign(X, K, T)` assigns a term T to the assignable object, if the associative memory X with the key K has the assignable object. Otherwise, the goal generates an assignable object in the associative memory X with the key K, and assigns T to the assignable object. On backtracking, the value of the assignable object returns to the previous value.

- **access(+Associative_memory, +Key, +Term)**
  The goal `access(X, K, T)` unifies the value of the assignable object with a term T, if the associative memory X with the key K has the assignable object. Otherwise, the goal generates an assignable object in the associative memory X with the key K, and unifies T with the initial value of the assignable object. If the value of the assignable object is empty, it raises an instantiation error.

- **assign_unique(+Associative_memory, +Key, +Term)**
  The goal `assign_unique(X, K, T)` sets the assignable object to have only the value of a term T, if the associative memory X with the key K has the assignable object. Otherwise, the goal generates an assignable object in the associative memory X with the key K, and sets the assignable object to have only the value T. On backtracking, the value of the assignable object returns to empty.
4.3 Global Arrays and Global Associative Functions

The predicates `garray/3` and `gassoc/2` defined as follows generate an array
and an associative memory and assign a given global variable to this array and
associative memory, respectively.

\[
\begin{align*}
garray(G,N) & :- \text{array}(A,N), \text{assign}(G,A). \\
garray(G,N,I) & :- \text{array}(A,N,I), \text{assign}(G,A).
\end{align*}
\]

\[
\begin{align*}
gassoc(G) & :- \text{assoc}(A), \text{assign}(G,A). \\
gassoc(G,I) & :- \text{assoc}(A,I), \text{assign}(G,A).
\end{align*}
\]

After a global variable for an array or for an associative memory is defined,
their elements are manipulated by `assign/3`, `assign_unique/3` and `access/3`
which have the atom in the first arguments. The second argument (Key) is an
integer index in the case of an array, and a list in the case of an associative
memory.

4.3.1 Note

Functionally, the associative memories include the arrays: We can convert any
program using the arrays to the programs using associative memories instead
of the arrays. On the other hand, the arrays are generally more efficient than
the associative memories.

We have questions on arrays and associative functions as follows.

1. Should the standard have only the associative functions?
2. Should the standard have only the arrays?
3. Do we need two- or more dimensional arrays in addition to one-dimensional
arrays?

4.3.2 Example: A Dynamic Associative Database

The following program defines predicates `record/3`, `retrieve/3` and `erase/3`
for an associative database. The associative memory with the initial value of a
variable is defined by the goal `assoc(A,_)`.

\[
\begin{align*}
\text{record}(\_,[\_],[\_]) & :- !. \\
\text{record}(A,[K|L],V) & :- \text{access}(A,K,VL), \text{dmember}(V,VL), \\
& \quad \text{record}(A,L,V). \\
\text{retrieve}(A,K,V) & :- \text{access}(A,K,VL), \text{memberx}(V,VL). \\
\text{erase}(\_,\_,\_) & :- !. \\
\text{erase}(A,[K|L],V) & :- \text{access}(A,K,VL), \text{delete}(VL,V,VL1), \\
& \quad \text{assign}(A,K,VL1), \text{erase}(A,L,V).
\end{align*}
\]
\text{member}(A, [A|\_]) \leftarrow !.
\text{member}(A, [\_|Y]) \leftarrow \text{member}(A, Y).

\text{member}(\_, Y) \leftarrow \text{var}(Y), !, \text{fail}.
\text{member}(A, [A|\_]) \leftarrow !.
\text{member}(A, [\_|Y]) \leftarrow \text{member}(A, Y).

\text{deletex}(\_, X) \leftarrow \text{var}(X), !.
\text{deletex}([A|X], A, X) \leftarrow !.
\text{deletex}([B|X], A, [B|Y]) \leftarrow \text{deletex}(X, A, Y).

The goal \text{record}(A, KL, T) stores term \( T \) with the keys in the list \( KL \) in the associative memory \( A \), which can be a global variable \( A \) specifying the associative memory. The goal \text{retrieve}(A, K, T) unifies the term \( T \) with the stored item with the key \( K \) in the associative memory \( A \). After defining the global associative memory by \text{gassoc(database, \_)} the goals,
\begin{align*}
\text{record} & (\text{database}, [[g,a],[g,b]], g(a,b)), \\
\text{record} & (\text{database}, [[g,a],[g,c]], g(a,c)),
\end{align*}
store the terms \( g(a,b) \) and \( g(a,c) \), each of which has two keys, in the global associative memory. The goal,
\text{retrieve}(\text{database}([g,a], V)
\text{succeeds with} \( V = g(a,b) \). On backtracking, this nondeterministic goal succeeds with \( V = g(a,c) \).

5 Non-Backtrackable Global Variables

Non-backtrackable global variables, or static global variables, are variables such that their values do not change on backtracking. These variables can be used for controlling the program execution and for storing and collecting all the solutions. The non-backtrackable global variables have been realized in Standard Prolog by \text{asserta/1} and \text{retract/1}, and the predicates \text{recorda/2} and \text{erase/2} for the recorded database in old Edinburgh Prolog. An important problem of these built-in predicates for global variables is that they change not only the values of global variables but also program clauses.

We can extend the notation and the syntax of backtrackable global variables and arrays to those of non-backtrackable ones. The semantics of these global variables, however, is intrinsically different from that of backtrackable variables, and rather similar to the pair of \text{asserta/1} and \text{retract/1} and that of \text{recorda/1} and \text{erase/1}.

The followings are questions on the non-backtrackable global variables.

1. Do we need to include the non-backtrackable global variables into the standard?
2. Which form of the non-backtrackable global variables is better, the recorded database or the extension of backtrackable global variables?

3. Do we need to include non-backtrackable global arrays or associative memories?

References


