Proposal for Global Variables in Standard Prolog

Katsuhiko Nakamura and Nobukuni Kino
Japanese Prolog WG

September 20, 2007

1 Introduction

This document describes a proposal for incorporating global variables based on logical assignment into standard Prolog. This proposal is a result of discussions in Japanese Prolog WG, which took account of the Minutes of WG17 Seattle meeting in 2006. It follows the previous document [3].

2 Necessity of Logical Assignment and Global Variable

Data types of Prolog are essentially restricted to terms 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 direct 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, as the values of both global variables and the array elements need to be updated.

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 [5] showed formal semantics for programs with the global variables and a method of transforming programs with the global variables into efficient logic programs.

In general, logic programming observes the *single assignment rule* so that no variable is destructively assigned in usual execution. On the other hand, we need to update the values in the global variables and array elements. The logical assignment is a mechanism to compromise these two contradictory conditions, by which we can update the values of global variables and array elements preserving the single assignment rule. The logical assignment is realized by a mutable term in SICStus Prolog and an assignable term (or object) in K-Prolog.

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 copying the entire array. An method of realizing arrays in single assignment languages is described in [1].

A common practical method in Prolog is that one-dimensional arrays are realized by terms usually with large arities. These terms are created by built-in predicate `functor/3`, and their elements are accessed by `arg/3`. There is, however, a problem that it needs a means to updating elements of the array. 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 [7] describes the problem of this built-in predicate as "`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 have backtrackable global variables without the logical assignment. The problem of destructive assignment by `setarg/3` is common to that of these global variables. It is written in the manual of SWI-Prolog that"The goal `b_getval(+Name,?Value)` gets the value associated the global variable `Name` and unifies it with `Value`. Note that this unification may further instantiate the value of the global variable. If this is not undesirable the normal precautions (double negation or copy_term/2) must be taken." The global variables based on logical assignment need no such precaution.

The clause creation and destruction predicates `asserta/1` and `retract/1` are used in some parts 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 early implementations of Prolog had nonbacktrackable built-in predicates for "recorded 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 [4].

3 A Survey on Global Variables in Existing Implementations

This section describes global variables and associative functions in existing implementations based on the responses to the authors' request for the survey.

3.1 Logical Assignment

The logical assignment is implemented by assignable terms (or mutable terms) in K-Prolog and by mutable terms in SICStus Prolog and YAP prolog. The mutable term is almost equivalent to the assignable terms. The global variables and the arrays in IF/Prolog are essentially based on logical assignment. IF/Prolog, however, has no predicate for directly creating and manipulating special objects like the assignable term or the mutable terms.

SICStus Prolog and YAP prolog have the following built-in predicate for the mutable terms:

- `create_mutable(+Datum,-Mutable)` creates a mutable term with an initial value `Datum` and returns to `Mutable`;
- `get_mutable(?Datum,+Mutable)` unifies `Datum` with the current value of `Mutable`; and
- `update_mutable(+Datum,+Mutable)` updates a value of `Mutable` to `Datum`.

A mutable term is represented as a compound term of the form

$$\texttt{\$mutable}(Value, Timestamp),$$

where $Value$ is the current value and $Timestamp$ is a term reserved for bookkeeping purpose.

In K-Prolog, an assignable term with an initial value t is represented by a special term "`}`t`{`." Hence, it needs not to have a predicate for creating the assignable terms. The assignable term can be "empty" state, which has no value and represented by "`}{`." It has built-in predicates defined as operators `+Mutable <= +Datum` and `+Mutable =>?Datum` for updating and accessing the assignable term, respectively.

3.2 Arrays

K-Prolog provides built-in predicates for manipulating several forms of arrays and associative functions based on the assignable terms.

SICStus Prolog removed `setarg/3` because of the problem of destructive assignment. Hence the users should use the mutable terms for updating values in arguments of terms instead of `setarg/3`. It has, however, no built-in predicate for binding the mutable terms to arrays, or terms used for one -dimensional arrays. SICStus Prolog also adopts the other approach such as extendible arrays with logarithmic access time based on logical assignment.

BinProlog provides `change_arg/3` for safe update of an argument of a term. It works like `setarg/3` except that the side-effects are permanent.

IF/Prolog has one-dimensional arrays based on logical assignment. It provides the following built-in predicates:

- create_array(?Array,@Dimension) creates an array with the size Dimension;
- set_array(@Array,@Index,@Term) assigns Term to the element with Index of Array, where Index can be an executable arithematic expression; and
- get_array(@Array,@Index,?Term) unifies Term with a current of the element.

If the element is assigned more times successively, then the old values are pushed down. The values are popped up by successive get_array/3.

3.3 Backtrackable Global Variables

K-Prolog has backtrackable global variables based on the logical assignment. In K-Prolog, the goal global(+Variable) makes Variable a global variable having an assignable term with the empty state. The scope of the global variable is inside the module.

IF/Prolog provides the following built-in predicates for global variables in addition to set_global(+Name,@Value) and get_global(+Name,?Value).

- push_global(+Name,@Value) creates and/or the set global variable Name to Value. If Name already existed, then its current value is pushed down and Value becomes the new value.
- pop_global(+Name,+Value) unifies Value with the topmost element of value stack of Name and pop it up one step. If it was the last element, then the Name is deleted.

Some implementations have backtrackable global variables not based on the logical assignment. Recent version of SWI-Prolog has built-in predicates b_setval(+Name,+Value) and b_getval(+Name,?Value) for backtrackable global variables, where Name is an atom.

B-Prolog has built-in predicates, global_heap_set(+Name,+Value) and global_heap_get(+Name,?Value). If Name is an atom, these are similar to b_setval/2 and b_getval/2 of SWI-Prolog. The Name can be any ground term other than any atom, these predicates might be classified to associative functions.

3.4 Non-Backtrackable Global Variables

SWI-Prolog provides the built-in predicates for the recorded database such as recorda/2 and erase/2. It also provides nonbacktrackable built-in predicates nb_setval(+Name,+Value) and nb_getval(+Name,?Value).

The global variable in BIN-Prolog is called global *logical* variable. These are logical in the sense that "Although a global variable cannot be changed, it can be further instanciated as it happens to ordinary Prolog terms."

IF/Prolog and SICStus-Prolog have non-backtrackable built-in predicates for hash memories called "blackboard." For example, `bb_get(+Boad,?Key,?Term)` in IF/Prolog retrieves `(Key,Term)` tuple in the blackboard Board, where Key can be any ground term.

BinProlog provides global variables called *global logical variables*, each of which can be used to access a hash table by two keys and instantiated by a special built-in predicate. B-Prolog has built-in predicates `global_set(+Name,+Value)` and `global_get(+Name,?Value)` for non-backtrackabel global variables. MINERVA IF/Prolog has similar built-in predicates `set_global(+Name,+Value)` and `get_global(+Name,?Value)` in addition to the predicates for the blackboard.

4 Logical Assignment

An *assignable term* is a special term for realizing logical assignment. A value is assigned to, and accessed from, the assignable term by built-in predicates. In backtracking, an assigned value is withdrawn and the assignable term has the previous value.

The assignable terms are created either by a built-in predicate or by unifying a variable with an associative term with an initial value. The created assignable term has either an initial value or a special *empty* state, which represents that the assignable term has no value.

Input and output of the assignable terms are implementation-defined, but should satisfy the following requirements:

1. Any assignable term in an input term can contain either an initial value, or no value in the case of the empty state; and

2. Every output of the assignable terms contains either the current value of the assignable term, or no value in the case of the empty state.

Notes:

1. The ordering of the assignable terms is implementation-defined.

2. The effect of unifying two assignable objects is undefined.

3. Any copy of an assignable object created by `copy_term/2`, assert, retract, or an all solution predicate is an independent copy of the original assignable term. Any assignment to either the original or the copy will not affect the other (This is based on SICStus Prolog Manual, Section 4.8.9).

4.1 Predicates for Assignable Terms

The followings are built-in predicates for creating, updating and accessing the values of the assignable terms.

- `assignable(-Variable)`
  The goal `assignable`(X) generates an assignable term with the value empty and returns the term representing the assignable term to X.

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

- `assign(+Assignable_Term,+Term)`
  The goal `assign`(X, T) assigns a term T to an assignable term X: The value of the assignable term is replaced by the term T. On backtracking, the value of X returns to the previous value.

- `access(+Assignable_Term,+Term)`
  The goal `access` (X, T) unifies the value of an assignable term X with a term T. If X is empty, it raises an instantiation error.

- `assign_unique(+Assignable_Term,+Term)`
  The goal `assign_unique` (X, T) sets an assignable term X to have only the value of a term T. On backtracking, the assignable term returns to empty.

- `empty(+Assignable_Term)`
  The goal `empty(+Assignable_Term)` succeeds, if `Assignable_Term` is empty, fails otherwise.

The predicate `assign_unique`/2 sets a single value to the assignable term. 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`assign`/2 may increase useless memory consumption.

4.2 Notes

1. An access to an assignable term with the value empty by `access`/2 causes an error. An option is that the access to the empty assignable term fails.

2. In many cases, the predicate `assign_unique`/2 can be replaced by `assign`/2 without changing the result of the program.

3. The names of the built-in predicates are subject to change. For example, `assignable/1` access-/2 can be renamed to `mutable/1` as in SICSTUS Prolog. Similarly, `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 the assignable terms but to also the global variables (Section 5).

4.3 Example 1: A Counter

```
counter(C) :- assignable(C,0).
count_up(C) :- access(C,I), I1 is I+1, assign_uniqe(C,I1).
counter_value(C,I) :- access(C,I).
```

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

Another example of game boards using `assign_unique/3` is shown in Section 6.2.

4.4 Semantics of Assignable Terms

Semantics, or a model, of the assignable terms 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.

```
assignable(_).
assignable([I|_],I).

assign(L,V) :- addtail(L,V).

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

access(L,V) :- var(L),!,
      instatiation_error(assignable_term).
access(L,V) :- findtail(L, V).

findtail([V|L],V) :- var(L),!.
findtail([_|L],V) :- findtail(L,V).

empty(X) :- var(X).
```

Note that this method using lists as the assignable terms 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 term. 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).

The standard does not specify how the assignable terms can be 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.

5 Arrays as Complex Terms

This section describes a method of realizing one-dimensional arrays based on the logical assignment. The array is a complex term, in which each element, or each argument, has an assignable term.

5.1 A Reference Implementation for Arrays

The following program implements predicates for the array, which is represented by a term in which each argument has an assignable term.

```
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),
      (var(I) -> assignable(E);
         functor(I,array,_) -> E=I; assignable(E,I)),
      K1 is K+1, copy_term(I,I1), initialize(K1,T,N,I1).

assign_array(A,K,T) :- arg(A, array,E), assign(E,T).
assign_unique_array(A,K,T) :- arg(A, array,E), assign_unique(E,T).
access_array(A,K,T) :- arg(A, array,E), access(E, T).
```

For defining an array of arrays, or a two-dimensional array, `initialize/4` simply copies each argument of a term with functor name `array`.

5.2 Note:

1. It is desirable that the standard includes the above predicates as built-ins.
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 implementation.

5.3 Example: Chess Boards

```
% board(N,B): returns a term representing an N*N array to B.
  board(N,B) :- array(Row,N,_), array(B,N,Row).

% place(B,I,J,P): place P to (I,J) of board B.
  place(B,I,J,P) :- arg(I,B,R), arg(J,R,A), assign(A,P).

% move(B,I,J,P): a player's move P to (I,J) of board B.
  move(B,I,J,P) :- arg(I,B,R), arg(J,R,A), assign_unique(A,P).
```

In game programming, the board is updated by the moves not only of the players but also of the search program. Although the moves in the search by `place/4` need to be backtracked, the moves by the players generally not to be backtracked. By using `assign_unique/3` for the moves by the players, we can save the memory for storing the boards.

6 Global Variables Including Associative Functionality

The (backtrackable) global variables are considered as a mapping from the set of keys to the set of assignable terms. The key of a global variable is represented by either an atom or a ground complex term. The global variable with keys of the complex term are considered as associative functionality, especially those with keys of the form, for example, `a(15)` or `a(10,20)` as array elements. The scope of a global variable is the module.

The value of global variable is accessed by the built-in predicates.

6.1 Built-in Predicates for Global Variables

A global variable is defined by built-in predicates `global`/1 and `global`/2, which can be also directives.

- `global(+Key)`
  The goal `global`(Key) generates an empty assignable term and relates the global variable Key with the assignable term. Key is a ground term except a number.

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

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 terms, since the global variables are atoms.

Note

The values of global variables with keys of complex terms are generally stored in hash memories.

6.2 Examples

Example 1.

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

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

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

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

Example 2.

```
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, newsymbol1(Q), newsymbol1(R).
Q = p
R = q ;
Instantiation error
```

On backtracking, the goal `newsymbol(Q)` fails and gives `Q` the previous value. The goal `newsymbol1(Q)` also fails on backtracking, but the assignable term returns to the empty state.

6.3 Questions

The followings are questions on the backtrackable global variables.

1. Does the standard need to include backtrackable global arrays?
2. Does backtrackable global variable need to be extended to access hash memories? (Does the atom in `global(+Atom, +Term)` need to be extended to ground term?)

7 Non-Backtrackable Global Variables

Nonbacktrackablle global 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 `asserta/1` and `retract/1`, and the predicates `recorda`/2 and `erase`/2 for the recorded database used in old Edinburgh Prolog and some recent implementation.

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 global variables, and rather similar to the pair of `asserta/1` and `retract/1` and that of `recorda/1` and `erase/1`.

7.1 Questions

The followings are questions on the nonbacktrackable global variables.

1. Does the standard need to include the nonbacktrackable global variables into the standard?
2. Which form of the nonbacktrackable global variables is better, the recorded database or the extension of backtrackable global variables?
3. Does nonbacktrackable global variable need to include arrays?
4. Does nonbacktrackable global variable need to be extended to access hash memories?

Acknowledgement

The authors thank Jan Wielemaker, Osker Bartenstein, Neng-Fa Zhou, Mats Carlsson, and Klaus Däßler for sending us information on global variables and arrays in existing implementations, which is reflected in Section 3.

References

[1] A. Kågedal and S. Debray, A practical approach to structure reuse of arrays in single assignment languages, *Proc. ICLP,* 1997 pp. 18-32.

[2] Toshinori Munakata, Notes on implementing sets in Prolog, *Comm. ACM* 35, 1992, pp.112-120.

[3] K. Namaura and N. Kino, Japanese Proposal for Global Variables and Associative Functions in Standard Prolog (Revised), September 2006.

[4] N.D. North, and Roger S. Scowen (eds.), Paris 1989 Meeting–Minutes, *SO/IEC JTC1 SC22 WG17, N36*, 1989.

[5] Peter Schachte, Global variables in logic programming, *Proc. of ICLP*, 1997, pp.3-17.

[6] Roger S. Scowen (ed.), *Prolog Part I: General Core*, ISO/IEC DIS 13211-1, 1994.

[7] Paul Tarau, *Binprolog 5.25 Users Guide.* Departement d'Informatique Universit, de Moncton Moncton, Canada, available from `http://139.103.16.8/BinProlog/public_html/public_html/bp575pc/doc/html/art.html`, 1997.