A Proposal for Global Variables and Arrays in Standard Prolog

Katsuhiko Nakamura
Japanese Prolog WG

January 8, 2005

1 Introduction

This document describes the significance of incorporating global variables and array functions into standard Prolog, and an informal proposal of global variables and array functions in Prolog. The global variables in the proposal are based on the notion of logical assignment, which is implemented in K-Prolog [1].

Prolog is a unique language, which allows various methods of representing sets of data [2]. On the other hand, it has been commonly recognized that Prolog has two practical problems, which considerably prevent logic programming from being used for broader area of information processing. One problem is that Standard Prolog does not specify means for efficient random access of data in large working memory such as arrays or hash tables. The other problem is that pure Prolog has no global variables such as those in Lisp and C. This causes the inefficiency of programs as well as the difficulty in reading and writing programs, since the numbers of arguments need to be increased in large programs. The array functions are closely related to the global variables, since array elements are considered as global variables, whose values need to be updated.

The clause creation and destruction predicates assert/1 and retract/1 are used in the 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-base functions for altering existing programs. Some earlier versions of Prolog had other built-in predicates for “recorded database” such as recorda/2 and erase/2 for storing terms. These predicates were excluded from the standard in an early stage of Prolog standardization for the reason that these are similar to assert/1 and retract/1 and that code and data should not be distinguished in Prolog [4].
1.1 Research Works and Existing Implementations for Global Variables and Arrays

It is well known that global variables can be transformed to additional arguments. Schachte [5] showed a formal semantics for programs with the global variables and a method to transform programs with the global variables into efficient logic programs.

It has been an important problem how to realize arrays efficiently without violating single assignment rule not only in logic programming but in functional programming, since updating an array element generally needs to copy the whole array. An method of realizing arrays in single assignment languages is described in [2].

Most existing Prolog implementations provide some array functions as implementation-dependent 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 setarg/3 for updating terms. A goal setarg(I,Term,Value) assigns destructively the Value to the I-th argument of Term. Tarau [7] 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 instantiated to 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 Logical Assignment

Our proposal is based on the notion called logical assignment, by which we can updating the values of global variables without using destructive assignments.

2.1 Assignable Objects

The logical assignment is applied to a assignable objects. A value is assigned to an assignable object, and the value is retrieved from the assignable object. The
assignable object has a special state `empty`, in which the object has no value. We can set a Prolog variable \( X \) to a logical assignable object by unifying the variable with a special term \( \#T \), where \( T \) is a term. The generated object has the initial value \( T \).

2.2 Predicates for Logical Assignment

We have the following four built-in predicates for setting values to assignable objects and for retrieving to assignable objects.

- \(<=\>/2\) The goal \( X \leftarrow T \) assigns a term \( T \) to an assignment object \( X \). On backtracking, the value of \( X \) returns to the old value.
- \(=>\>/2\) The goal \( X \Rightarrow T \) unifies the value of an assignment object \( X \) with a term \( T \). If \( X \) is empty, it raises an instantiation error. On backtracking, \( T \) returns to the previous value.
- \(<=\>/2\) The goal \( X \leftarrow T \) sets an assignable object \( X \) to have only the value of a term \( T \). On backtracking, the assignable object returns to empty.
- \(<=\>/1\) The goal \( X \leftarrow V \) sets an assignable object \( X \) to have only the current value. On backtracking, the assignable object returns to empty.

These predicate are deterministic, and are defined as operators by the following directives.

\[
\text{:- op(700, xf, <=).}
\text{:- op(700, xf, =>).}
\text{:- op(700, xfx, <=).}
\text{:- op(700, xf, <=).}
\]

Predicates \(<=\>/2\) and \(<=\>/1\) are introduced for the economy of memory, since the repetitive use of assignment by predicate \(<=\>/2\) increases memory consumption.

Errors:

- `instantiation_error`
- `type_error(assignable_object)`

Notes

1. Predicate \(<=\>/2\) can be generally replaced by \(<=\>/2\) without changing the result of the program.
2. The goal \( X \leftarrow \) is equivalent to \( (X \Rightarrow T, X \leftarrow T) \).
3. These two predicates are related to the model of assignable objects, which is given by the linear list as described in the next subsection.
2.3 A Model of Assignable Objects

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

```prolog
assignment_object(L) :- assign(L,V).
assignment_object(L) => V :- findtail(L,V).

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

Note that the terms with functor assignment_object/1 in this program represent the assignable objects. It is not possible by pure logic program to implement $<=$/2 and $=>$/1. On the other hand, we can efficiently implement not only these predicates but also $=>$/2 and $<=$/2 by “safe” use of setarg/3 (see Note in Section 3).

It is not specified in the standard how the logical assignable objects can be efficiently implemented. Some possible methods include the use of either special pointers to end cells or cache memories for efficiently accessing the values in the ends of the linear lists, and other different approaches that do not use the linear lists.

3 Global Variables

A global variable is represented by an atom. The scope of a global variable is the module.

A global variable is defined by directive global/1. The goal of the form global(+Name) associates the name of the global variable Name with an empty assignable object. The values of the global variables $X$ are retrieved by $X => V$ and updated by $X <= V$ and $X <> V$.

Example 1.

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

rev([],Y) :- result <= Y.
rev([A|X],Y) :- rev([Y|A],X).

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

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

4
Example 2.

```
initialize :- global(symbol_list),
    symbol_list <= [p,q,r,s,t,u,v].
newsymbol(Q) :- symbol_list => [Q|L], symbol_list <= L.
newsymbol1(Q) :- symbol_list => [Q|L], symbol_list <= L.

?- initialize, newsymbol(Q), newsymbol(R).
  Q = p
  R = q

?- initialize, newsymbol1(Q), newsymbol1(R).
  Q = p
  R = q
```

On backtracking, the goal newsymbol(Q) fails and gives Q the previous value. The goal newsymbol(Q) also fails on backtracking, but Q returns to an un-instantiated variable.

Note

The predicates for the assignable objects and the global variables are implemented in K-Prolog version 5 [1], where an initial value for an assignable object is represented as ’Term’ instead of #Term.

The following web site contains implementation programs for predicates of logical assignment and example programs written by N. Kino.

One of the implementation programs is almost portable and almost logical: predicates <=/2 and =>/2 are implemented in Standard Prolog. A faster version of these predicates and the implementation of <=/2 and <=/1 uses “safe” destructive updating by setarg/3 as mentioned in Section 2.

4 Arrays

We represent one dimensional array by a term with an arity of the size of the array. The goal of the template array(-array,+name(N1,N2,...,Nn)), generates an assignable array object and unifies it with array, where m is the size of this array and name is an atom representing the name of the array. The array itself and the elements of the array are assignable terms. After name is declared as global by global/1 directive, predicate array/1 is defined by

```
array(Form) :- functor(Form,Name,_),
    array(Name,Form).
```

Elements of an array array are accessed by predicate array_arg/3 of the template array_arg(+array, +form, -element). When name is declared as global by using global/1 directive, predicate array_arg(+form,-element) is defined by

```
array(Form) :- functor(Form,Name,_),
    array(Name,Form).
```
array_arg(Form, Element) :-
    functor(Form, Name, _),
    array_arg(Name, Form, Element).

Errors:

instantiation_error

function_error(compound)

function_error(integer)

domain_error(index_out_of_bound)

Example

The following program defines an array representing a two dimensional game board:

:-global(board).
board_initialize :- array(board(4,4)).
board_put(N, M, Value) :-
    array_element(board(N, M), Place),
    Place <- Value.
board_get(N, M, Value) :-
    array_element(board(N, M), Place),
    Place => Value.

For using this subprogram in game programming, the predicate <=/2 in the clause for board_put/3 needs to be replaced by <=/2, so that the backtracking allows the search for better moves.

5 Static Global Variables

By static global variables, we mean the global variables such that their values do not change to the old values on backtracking. These variables can be used for controlling the program execution and for storing and collecting all the solutions. Static global variables have been realized in Standard Prolog by asserta/1, assertz/1 and retract/1. An important defect of these built-in predicates is that they change not only global variables but also program clauses.

We can extend the notation and the syntax of dynamic global variables and arrays to those of static global variables by restricting the values not to be restored in backtracking. The semantics of static global variables, however, is intrinsically different from that of dynamic variables. Hence, we leave further discussion on this problem open in this document.

6 Concluding Remarks

A feature of our approach is that global variables and array functions are based on a logic model called logical assignment, of which the essential part can be
specified by simple Prolog programs. Another feature is that the proposed methods can be efficiently implemented in both computation time and memory usage.

There are several different implementations for array functions and global variables, which are essential for writing efficient and practical programs for broader areas in Prolog. Therefore, we are convinced that adding these functions makes ISO Prolog a more powerful and useful standard.

References


