ISO/IEC Draft PDTR 13211-X:2009
Proposal for global variables in Prolog

Editor: Katsuhiko Nakamura
nakamura@k.dendai.ac.jp

July 6, 2009

Introduction

This technical report specifies a set of built-in predicates for incorporating global variables into Prolog. It is an optional part of Part 1 of the International Standard for Prolog, ISO/IEC 13211-1.

Desirability of global variables in Prolog

It has been commonly recognized that Prolog has two practical problems, which represent a barrier to the use of the language and logic programming in the broader area of information processing. One problem is that Standard Prolog does not have the direct means for efficient random access to data in a large working memory such as is provided by arrays or hash memories. The other problem is that unlike Lisp and C pure Prolog has no global variables. Arrays are closely related to the global variables, as the values of both global variables and the elements in an array need to be updated.

Commonly 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 in large programs the numbers of arguments need to be substantially increased.

In general, logic programming observes the *single assignment rule* so that destructive assignment of variable usually does not take place. On the other hand, the values in the global variables and array elements do need to be updated. Logical assignment provides a compromise between these two contradictory requirements, allows for the updating of the values of global variables and array elements while preserving the single assignment rule. Logical assignment is realized by a mutable term in SICStus Prolog and an assignable term (or object) in K-Prolog.

It is useful to be able to update arrays efficiently without violating the single assignment rule not only in logic programming but also in functional programming, since otherwise updating an array element generally requires copying the entire array.

A common practice in Prolog is to realize one-dimensional arrays by terms with the appropriate arity. These terms are created by built-in predicate `functor/3`, and their elements are accessed by `arg/3`. As an approach to arrays this has the drawback that there is no consensus on how to perform an update. Several implementations of 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`. This built-in predicate is problematic in that it lacks a logical semantics and it is implemented differently in the several systems. Indeed, destructive assignment to an argument using `setarg/3` sometimes causes a fatal error that eliminates a value. This error can occur when the value of a variable in the argument is updated.

Some implementations have backtrackable global variables that does not implement logical assignment. Here again we have the issue associated to destructive assignment by `setarg/3`.

A quite different approach to global variables and arrays relies on the use of the clause creation and destruction predicates `asserta/1` and `retract/1`. These built-in predicates, however, do not implement the expected logical semantics for globals or arrays, Furthermore they 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 because it was considered than in Prolog code and data should not be distinguished.

Contributors

This proposal is based on collaboration with Nobukuni Kino and discussions in Japanese Prolog WG and WG17 Oporto meeting. Jonathan Hodgson contributed by checking the draft. Jan Wielemaker, Osker Bartenstein, Neng-Fa Zhou, Mats Carlsson, and Klaus Däßler contributed by the survey of existing implementations. The design of logical assignment is partially based on SICStus Prolog and K-Prolog.

Contents

1 Scope

The purpose of this technical report is to promote the portability of the global variables in Prolog and to promote the applicability of Prolog while preserving the logical characteristics and single assignment rule. It specifies:

a) a set of built-in predicates for defining mutable terms, assigning values to the mutable terms and updating their values;

b) a set of built-in predicates for defining global variables, assigning values to the global variables and updating their values, based on the mutable terms; and

c) some example programs to show the use of mutable terms and global variables.

2 Language concepts and semantics

2.1 Mutable terms

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

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

This technical report does not specify how mutable terms are to be implemented. However, this section does exhibit a method of realizing mutable terms using linear lists to describe their semantics (2.1.3). There are other more efficient approaches.

2.1.1 Input and output

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

a) Any mutable term in an input term can contain either an initial value, or no value in the case of the empty state; and

b) Every output of a mutable term contains either the current value of the mutable term, or no value in the case of the empty state.

2.1.2 Other requirements for mutable terms

a) The ordering of mutable terms is implementation-defined.

b) The effect of unifying two mutable terms is undefined.

c) Any copy of a mutable term created by the built-in predicates `copy_term/2`, `assert/1`, `retract/1`, or an all solution predicate is an independent copy of the original mutable term. Any assignment to either the original or the copy will not affect the other.

2.1.3 Semantics of mutable terms

Semantics of the mutable 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 `is_mutable`/1 (3.2.1) for testing mutable terms, `mutable`/1 (3.2.2) for creating mutable terms, `mutable`/2 (3.2.3), `set_mutable`/2 (3.3.1) for assigning values to mutable terms, and `mutable_value`/2 (3.3.2) for retrieving values of mutable values are in effect equivalent to those defined by the following Prolog programs.

```
is_mutable(L) :-  var(L),!.
is_mutable([I|L]) :- nonvar(I), is_mutable(L).

mutable(L) :- var(L),!.
mutable([I|L], I) :- nonvar(I),  var(L),!.

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

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

findtail([_|L], _) :- nonvar(L),
                throw(type_error(mutable_term)).

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

NOTE

This method of using lists to represent mutable terms is not efficient, particularly as the list may change and carry many values. A possible method for improving the efficiency is the use of either special pointers to the end cells of the linear lists or a cache memory for efficiently accessing the values in the ends of the lists.

2.2 Global variables

Global variables are considered as a mapping from the set of names to the set of values, which satisfy the following requirements.

a) Each name of a global variable is represented by a ground term.

b) Each value of a global variable is a term.

c) The values are assigned to the names and accessed, by built-in predicates `set_global/2` (3.5.1) and `global_value/2` (3.5.2), respectively.

d) The values can be updated and returned to the previous values in backtracking.

e) The scope of a global variable is the module (ISO/IEC 13211-2) where the variable is defined.

2.2.1 Creation of global variables

The global variables are defined and created by the built-in predicates global/1 (3.4.4) and global/2 (3.4.5), each of which creates a mutable term and associates a name to this mutable term.

These two predicates may also used as directives.

NOTE

Global variables can be used as array elements, or elements of hash memories, since the variable names include terms such as `table(100)` and `p(a,8)`.

2.3 Non-backtrackable global variables

Nonbacktrackable global variables can be used for controlling program execution and for storing and collecting all the solutions. 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 implementations.

It is possible to 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. On the other hand the semantics is similar to the pair of predicates `asserta/1` and `retract/1` and to that of `recorda/1` and `erase/1`.

This technical report does not specify non-backtrackable global variables.

3 Built-in predicates and Directives

This section specifies built-in predicates for creating mutable terms, defining global variables and retrieving values of the mutable terms and global variables. The semantics of these predicates is described by the implementation in 2.1.3.

3.1 The format of built-in predicate definition

These subclauses define the format of the definition of built-in predicates which is additional to Section 8.1 in ISO/IEC 13211-1.

3.1.1 Types of an argument

The type of each argument is defined by one of the following atoms and the atoms in 8.1.2.1 of ISO/IEC 13211-1.

`global_variable` — a ground term associated with a global variable (see 2.2).

`ground_term` — a term which contains no variables.

`mutable_term` — a mutable term (see 2.1).

3.1.2 Example

This technical report assumes in the examples that `$mutable(`X`)` is a mutable term with the value X, and that `$mutable($empty)` is a mutable term with the value empty.

NOTE

This form of mutable terms is similar to, but not equal to, that of SICStus Prolog. The form is used only for describing the examples, as the input and output of the mutable terms are implementation defined (2.1.1).

3.2 Testing and creating mutable terms

3.2.1 is_mutable/1

3.2.1.1 Description

`is_mutable(Mutable)` is true iff `Mutable` is a mutable term.

Procedurally, if `Mutable` is a mutable term, the goal `is_mutable(Mutable)` succeeds, else the goal fails.

3.2.1.2 Templates and modes

`is_mutable(@term)`

3.2.1.3 Errors

None.

3.2.1.4 Examples

`is_mutable(X).`
Fails.

`is_mutable(f(a)).`
Fails.

`is_mutable($mutable(f(a))).`
Succeeds.

3.2.2 mutable/1

3.2.2.1 Description

`mutable(Mutable)` is true iff `Mutable` unifies with a mutable term with the value empty, which is newly created by the processor.

Procedurally, the goal `mutable(Mutable)` generates a new mutable term with the value empty and unifies it with `Mutable`.
On backtracking, the mutable term and unification are withdrawn.

3.2.2.2 Templates and modes

`mutable(?term)`

3.2.2.3 Errors

None.

3.2.2.4 Examples

`mutable(X).`
Succeeds, unifying `X` with a unique mutable term with the value empty, `$mutable($empty)`.

`mutable(f(a)).`
Fails.

`mutable($mutable($empty)).`
Succeeds.

3.2.3 mutable/2

3.2.3.1 Description

`mutable(Mutable,Value)` is true, iff `Mutable` unifies with a mutable term with the value `Value`, which is newly created by the processor.

Procedurally, the goal `mutable(Mutable,Value)` generates a new mutable term with the value `Value` and unifies it with `Mutable`.
On backtracking, the mutable term and the unification are withdrawn.

3.2.3.2 Templates and modes

`mutable(?term, @term)`

3.2.3.3 Errors

None.

3.2.3.4 Examples

`mutable(M,g(X)).`
  Succeeds, unifying `M` with a mutable term `$mutable(g(X))`.

`mutable(f(a),g(b)).`
  Fails.

`mutable($mutable(f(X)), f(a)).`
  Succeeds unifying `X` with `f(a)`.

3.3 Assigning and retrieving values of mutable terms

3.3.1 set_mutable/2

3.3.1.1 Description

`set_mutable(Mutable, Value)` is true, iff `Mutable` is a mutable term.

Procedurally, `set_mutable(Mutable, Value)` is executed as follows:

a) If `Mutable` is a mutable term, the goal assigns `Value` to `Mutable`. The value of the mutable term is replaced by the term `Value`.

b) Else the goal causes a type error or an instantiation error.

On backtracking, the value `Value` returns to the previous value.

3.3.1.2 Templates and modes

`set_mutable(+mutable_term,@term).`

3.3.1.3 Errors

a) `Mutable` is not a mutable term
— `type error`.

b) `Mutable` is a variable
— `instantiation error`.

3.3.1.4 Examples

`set_mutable($mutable(g(t)),g(X)).`
Succeeds. The goal updates the value to `g(X)` of the mutable term.

`set_mutable(g(X),f(a)).`
`type_error`.

NOTE

The name `set_mutable`/2 can be renamed to `set_mutable_value`/2 or `assign_mutable`/2.

3.3.2 mutable_value/2

3.3.2.1 Description

`mutable_value(Mutable, Value)` is true, iff `Mutable` is a mutable term and the value of `Mutable` is `Value`.

Procedurally, `mutable_value(Mutable, Value)` is executed as follows:

a) If `Mutable` is a mutable term, the goal unifies the value of `Mutable` with `Value` .

b) Else the goal causes a type error or an instantiation error.

3.3.2.2 Templates and modes

`mutable_value(+mutable_term,?term)`

3.3.2.3 Errors

a) `Mutable` is a variable
— `instantiation_error`.

b) `Mutable` is not a mutable term
— `type_error`.

3.3.2.4 Examples

`mutable_value($mutable(g(Y)),X).`
Succeeds. The goal unifies the value `g(Y)` with `X`, and instantiates `X` to `g(Y)` .

`mutable_value($mutable(g(t)),f(X)).`
Fails.

`mutable_value(g(X),f(a)).`
`type_error`.

3.3.3 empty_mutable/1

3.3.3.1 Description

`empty_mutable(Mutable)` is true, iff `Mutable` is a mutable term with the value empty.

Procedurally, `empty_mutable(Mutable)` is executed as follows:

a) If `Mutable` is the mutable term with the value empty then succeeds.

b) Else if `Mutable` is the mutable term, then fails.

c) Else causes a type error or an instantiation error.

3.3.3.2 Templates and modes

`empty_mutable(+mutable_term)`

3.3.3.3 Errors

a) `Mutable` is a variable
— `instantiation_error`.

b) `Mutable` is not a mutable term
— `type_error`.

3.3.3.4 Examples

`empty_mutable($mutable($empty)).`
Succeeds.

`empty_mutable($mutable([a,b])).`
Fails.

`empty_mutable(g(X),f(a)).`
`type_error`.

3.4 Defining global variables

3.4.1 Directives

A global variable may be created with the directives `global`/1 and `global`/2. The arguments of these directives shall satisfy the same constraints as those required to the built-in predicates `global`/1 (3.4.4) and `global`/2 (3.4.5), respectively.

3.4.2 Directive global/1

A directive `global(Global)` creates a mutable term having the empty value, and links the global variable `Global` with this mutable term.

3.4.3 Directive global/2

A directive `global(Global,Value)` creates a mutable term having the value `Value`, and links the global variable `Global` to this mutable term.

After a global variable `Global` is defined, the values of `Global` are updated by `set_global(Global,Value)` (3.5.1), and are retrieved by `global_value(Global, Value)` (3.5.2).

3.4.4 global/1

3.4.4.1 Description

`global(Global)` is true, iff `Global` is a ground term.

Procedurally, `global(Global)` is executed as follows:

a) If `Global` is a ground term, the goal generates a mutable term having the empty value, and links the global variable `Global` with this mutable term.

b) Else causes `type_error`.

3.4.4.2 Templates and modes

`global(+ground_term)`

3.4.4.3 Errors

`Global` is not ground.
– `type_error`.

3.4.4.4 Examples

`global(global).`
Succeeds. The goal generates a mutable term with the empty value and links `global` to this mutable term.

`global(f(a)).`
Succeeds. The goal generates a mutable term with the empty value and links `f(a)` to this mutable term.

`global(g(X)).`
`type_error`.

3.4.5 global/2

3.4.5.1 Description

`global(Global,Value)` is true, iff `Global` is a ground term.

Procedurally, `global(Global,Value)` is executed as follows:

a) If `Global` is a ground term, the goal generates a mutable term having the value `Value`, and links the global variable `Global` to this mutable term.

b) Else causes `type_error`.

3.4.5.2 Templates and modes

`global(+ground_term,@term)`

3.4.5.3 Errors

`Global` is not ground
– `type_error`.

3.4.5.4 Examples

`global(global,g(X)).`
Succeeds. The goal generates a mutable term with the value `g(X)` and links `global` to this mutable term.

`global(f(a),[a,b]).`
Succeeds. The goal generates a mutable term with the value `[a,b]` and links `f(a)` to this mutable term.

`global(g(X),f(a)).`
`type_error`.

3.5 Assigning and retrieving values of global variables

3.5.1 set_global/2

3.5.1.1 Description

`set_global(Global, Value)` is true, iff `Global` is a global variable.

Procedurally, `set_global(Global, Value)` is executed as follows:

a) If `Global` is a global variable, the goal assigns `Value` to the mutable term to which `Global` links. The value of the global variable is replaced by the term `Value`.

b) Else the goal causes a type error or an instantiation error.

On backtracking, the value `Value` returns to the previous value.

3.5.1.2 Templates and modes

`set_global(+global_variable,@term)`.

3.5.1.3 Errors

a) `Global` is a variable
— `instatiation error`.

b) `Global` is not a global variable
— `type error`.

3.5.1.4 Examples

The following examples assume that `global` and `f(a)` have been defined as global variables.

`set_global(f(a),[c,d|X]).`
Succeeds. The goal updates the value of a mutable term linked by the global variable to `[c,d|X]`.

`set_global(Z,f(a)).`
`instantiation_error`.

`set_global(f(X),f(a)).`
`type_error`.

3.5.2 global_value/2

3.5.2.1 Description

`global_value(Global, Value)` is true, iff `Global` is a global variable and the value of `Global` is `Value`.

Procedurally, `global_value(Global, Value)` is executed as follows:

a) If `Global` is a global variable, the goal unifies the value of the mutable term, to which `Global` links, with `Value`.

b) Else the goal causes a type error or an instantiation error.

3.5.2.2 Templates and modes

`global_value(+global_variable,?term)`

3.5.2.3 Errors

a) `Global` is a variable
— `instantiation_error`.

b) `Global` is not a global variable
— `type_error`.

3.5.2.4 Examples

The following examples assume that `global` and `f(a)` have been defined as global variables with values `[a,b]` and `g(b)`, respectively.

`global_value(global,X).`
Succeeds. The goal unifies `X` with `[a,b]`.

On re-execution, fails.

`global_value(global,[c,d,e]).`
The goal fails since the value `[a,b]` of `global` with cannot be unified with `[c,d,e]`.

`global_value(g(X),f(a)).`
`type_error`.

3.5.3 current_global_variable/2

3.5.3.1 Description

`current_global_variable(Global, Value)` is true, iff `Global` is a global variable, and `Value` is the value currently associated with `Global`.

Procedurally, `current_global_variable(Global, Value)` is executed as follows:

a) Searches the current global variables defined by the user and creates a set S_{gb} of the term $gb(G, V)$ such that (1) there is a global variable G which unifies with `Global`, and (2) the value V currently associated with G unifies with `Value`.

b) If a non-empty set is found, then proceeds to d below.

c) Else the goal fails.

d) Choose a member of S_{gb} and the goal succeeds.

e) If all the members of S_{gb} have been chosen, then the goal fails.

f) Else choose a member of S_{gb} which has not already been chosen, and the goal succeeds.

`current_global_variable(Global, Value)` is re-executable. On re-execution, continue at d above.

The order in which global variables are found by `current_global_variable(Global, Value)` is implementation defined.

3.5.3.2 Templates and modes

`current_global_variable(?term,?term)`

3.5.3.3 Errors

None.

3.5.3.4 Examples

The following examples assume that `global` and `f(a)` have been defined as global variables with values `[a,b]` and `g(b)`, respectively.

`current_global_variable(global,X).`
Succeeds, unifying `X` with `[a,b]`.

On re-execution, fails.

`current_global_variable(G,X).`
Succeeds, unifying `G` with one of the global variables `global` and `X` with `[a,b]`.

On re-execution, unifies `G` with the other global variables `f(a)` and `X` with `g(b)`.

On re-execution, fails.

`current_global_variable(g(X),f(a)).`
Fails because no global variable unifies with `g(X)`.

`current_global_variable(G,g(X)).`
Succeeds, unifying `G` with `f(a)` and `X` with `b`.

On re-execution, the goal fails, because no other global variable and its value unify with `G` and `g(X)`.

3.5.4 empty_global/1

3.5.4.1 Description

`empty_global(Global)` is true, iff `Global` is a global variable with the value empty.

Procedurally, `empty_global(Global)` is executed as follows:

a) If `Global` is a global variable and its value is empty then succeeds.

b) Else if `Global` is a global variable and its value is not empty then fails.

c) Else if `Global` is not a ground term, then fails.

d) Else the goal causes a type error or an instantiation error.

3.5.4.2 Templates and modes

`empty_global(+global_variable)`

3.5.4.3 Errors

a) `Global` is a variable
— `instantiation_error`.

b) `Global` is not a global variable
— `type_error`.

3.5.4.4 Examples

The following examples assume that `f(a)` has been defined as a global variable with the value empty, and that `glbal` has been defined as a global variable with the value `[a,b]`.

`empty_global(f(a)).`
Succeeds.

`empty_global(global).`
Fails.

`empty_global(g(X)).`
`type_error`.

4 Example Programs

This section contains some example programs to show usage of mutable terms and global variables.

4.1 Reversing lists

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

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

This program is equivalent to the following program; `result` serves as an accumulator.

```
reverse(X,Result) :- rev(X,[],Result).

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

Both programs give the same results.

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

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

4.2 Generating symbols

```
initialize :- global(symbol_list),
              set_global(symbol_list, [p,q,r,s,t,u,v]).
newsymbol(Q) :- global_value(symbol_list, [Q|L]),
              set_global(symbol_list, L).

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

Q = p
R = q
```

On re-execution, the goal `newsymbol(Q)` fails and gives `Q` the previous value.

4.3 Arrays

This section describes a method of realizing one-dimensional arrays based on logical assignment. An array is represented by a complex term, in which each element, or each argument, is a mutable term.

```
array(T,N) :- functor(T,array,N), initialize(1,T,N).

initialize(K,_,N) :- K > N,!.
initialize(K,T,N) :- arg(K,T,E), mutable(E);
       K1 is K+1, initialize(K1,T,N).

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), copy_term(I,I1),
      mutable(E,I1), K1 is K+1, initialize(K1,T,N,I1).

set_array(A,K,T) :- arg(A,array,E), assign(E,T).
access_array(A,K,T) :- arg(A,K,E), mutable_value(E,T).
```

The goal `array(T,N)` generates a term representing an array of `N` global variables with empty state and unifies it with `T`. For example, the goal `array(A,3)` returns `array($mutable($empty),$mutable($empty),$mutable($empty))` to the variable `A`.

The goal `array(T,N,I)` generates a term `T` representing an array of `N` global variables with the initial values `I`, and unifies it with `T`. For example, the goal `array(A,5,0)` returns the term

```
array($mutable(0),$mutable(0),$mutable(0),$mutable(0),$mutable(0))
```

to the variable `A`. By using this predicate, we can define an array of arrays, or a two-dimensional array as shown in the next section. The goal `set_array(A,K,T)` assigns the value `T` to the `K`-th element of the array `A`, while `access_array(A,K,T)` unifies the value of `K`-th element with `T`.

The following program is an application of using the array for representing the chess board.

```
% 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), mutable_value(R,Q), arg(J,Q,A),
        assign(A,P).
```

NOTE

A problem in using terms as one-dimensional arrays is that the size of such arrays is then restricted by the maximum arity of terms, which in some implementations is not sufficiently large.