

# ISO/IEC Draft PDTR 13211-X:2011

## Proposal for global variables in Prolog

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

June 21, 2010

### 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 do 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 that 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 and Pasadena meetings. 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

<b>Introduction</b>	<b>1</b>
<b>1 Scope</b>	<b>4</b>
<b>2 Language concepts and semantics</b>	<b>4</b>
2.1 Mutable terms . . . . .	4
2.1.1 Input and output . . . . .	4
2.1.2 Other requirements for mutable terms . . . . .	4
2.1.3 Semantics of mutable terms . . . . .	5
2.2 Global variables . . . . .	5
2.2.1 Creation of global variables . . . . .	6
2.3 Non-backtrackable global variables . . . . .	6
<b>3 Built-in predicates and Directives</b>	<b>7</b>
3.1 The format of built-in predicate definition . . . . .	7
3.1.1 Types of an argument . . . . .	7
3.1.2 Errors . . . . .	8
3.1.3 Examples . . . . .	8
3.2 Testing and creating mutable terms . . . . .	8
3.2.1 mutable/1 . . . . .	8
3.2.2 new_mutable/2 . . . . .	9
3.3 Assigning and retrieving values of mutable terms . . . . .	9
3.3.1 set_mutable/2 . . . . .	9
3.3.2 mutable_value/2 . . . . .	10
3.4 Creating global variables . . . . .	11
3.4.1 Directive create_global/2 . . . . .	11
3.4.2 create_global/2 . . . . .	11
3.5 Assigning and retrieving values of global variables . . . . .	12
3.5.1 set_global/2 . . . . .	12
3.5.2 global_value/2 . . . . .	13
3.5.3 current_global/2 . . . . .	14
<b>4 Example programs</b>	<b>15</b>
4.1 Reversing lists . . . . .	15
4.2 Generating symbols . . . . .	16
4.3 Fibonacci numbers . . . . .	16
4.4 Arrays . . . . .	16
<b>Appendix A: History and changes from the previous draft . . . . .</b>	<b>17</b>
<b>Appendix B: A reference implementation . . . . .</b>	<b>17</b>

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

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 contains an initial value; and
- b) Every output of a mutable term contains the current value of the mutable term

#### 2.1.2 Other requirements for mutable terms

- a) The ordering of mutable terms is implementation-dependent.
- b) The effect of unifying two mutable terms is implementation-dependent, except in the case when the two terms are the same terms, this succeeds with the same mutable term.

- c) A mutable term must not be ground.
- d) 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 `mutable/1` (3.2.1) for testing mutable terms, `new_mutable/1` (3.2.2) for creating mutable terms, `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.

```
mutable([I|L]) :- mutable1(L).

mutable1(L) :- var(L), !.
mutable1(_|L) :- mutable(L).

new_mutable([I|L], 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).
```

#### 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 mutable terms, 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) A global variable with an initial value is created by built-in predicate `create_global/2` (3.4.2) similarly as the mutable terms.
- d) 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.
- e) A current global variable with its value is accessed by the built-in predicate `current_global/2` (3.5.3).
- f) The values can be updated and returned to the previous values in backtracking.
- g) 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 predicate `create_global/2` (3.4.2), which creates a mutable term and associates a name to this mutable term.

This predicate may also be used as directives.

#### NOTE

Global variables can be used to represent 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

A non-backtrackable global variable is a ground term with an associated value similarly as the global variable. The non-backtrackable global variable differs from the global variable in that updates are not undone on backtracking. In other words, the old value associated with a non-backtrackable global variable will not be restored after execution backtracks over the point where the update took place.

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

A non-backtrackable global variable shall meet the following requirements:

- a) Each name of a non-backtrackable global variable is represented by a ground term.
- b) Each value of a non-backtrackable global variable is a term.

- c) A non-backtrackable global variable with an initial value is created by the built-in predicate `create_nb_global/2`.
- d) The current value associated with a non-backtrackable global variable is accessed by the built-in predicate `nb_global_value/2`.
- e) The value associated with a non-backtrackable global variable is updated by the built-in predicate `set_nb_global/2`.
- f) A current non-backtrackable global variable with its value is accessed by the built-in predicate `current_nb_global/2`.
- g) The scope of a global variable is the module (ISO/IEC 13211-2) where the variable is defined.

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.

### 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 describe 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 Errors

Error classification in 7.12.2 of ISO/IEC 13211-1 is extended to include the following types.

`ValidType` of Type Error includes the types `global_variable` and `ground_term` in addition to those in 7.12.2 b of ISO/IEC 13211-1.

`ObjectType` of Existence Error includes the type `global_variable` in addition to those in 7.12.2 d of ISO/IEC 13211-1.

`PermissionType` of Permission Error includes the type `global_variable` in addition to those in 7.12.2 e of ISO/IEC 13211-1.

### 3.1.3 Examples

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

#### NOTES

- a) 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).
- b) Because any mutable term is not ground (2.1.2), the example `$mutable(a)` of the mutable term with a ground value `a` is not ground.

## 3.2 Testing and creating mutable terms

### 3.2.1 mutable/1

#### 3.2.1.1 Description

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

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

#### 3.2.1.2 Templates and modes

`mutable(@term)`

#### 3.2.1.3 Errors

None.

**3.2.1.4 Examples**

```
mutable(X).
  Fails.

mutable(f(a)).
  Fails.

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

**3.2.2 new\_mutable/2****3.2.2.1 Description**

`new_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 `new_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.

**3.2.2.2 Templates and modes**

```
new_mutable(?term, @term)
```

**3.2.2.3 Errors**

None.

**3.2.2.4 Examples**

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

new_mutable(f(a), g(b)).
  Fails.

new_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 `instantiation_error` or `type_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 a variable  
— `instantiation_error`.
- b) `Mutable` is not a mutable term  
— `type_error(mutable_term,Mutable)`.

### 3.3.1.4 Examples

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

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

## 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 `type_error` or `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(mutable_term,Mutable)`.

### 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(mutable_term,g(X))`.

## 3.4 Creating global variables

### 3.4.1 Directive `create_global/2`

A global variable may be created with the directive `create_global/2`. The arguments of this directive shall satisfy the same constraints as those required to the built-in predicate `create_global/2` (3.4.2).

A directive `create_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).

### 3.4.2 `create_global/2`

#### 3.4.2.1 Description

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

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

- a) If `Global` is not a ground term, then causes `type_error`.
- b) If `Global` has been a global variable, then causes `permission_error`.
- c) Else the goal generates a mutable term having the value `Value`, and links the global variable `Global` to this mutable term.

**3.4.2.2 Templates and modes**

```
create_global(+ground_term,@term)
```

**3.4.2.3 Errors**

- a) Global is not ground
  - `type_error(ground_term,Global)`.
- b) Global has been a global variable
  - `permission_error(create, global_variable, Global)`.

**3.4.2.4 Examples**

```
create_global(global,g(X)).
```

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

```
create_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.

```
create_global(g(X),f(a)).
```

```
type_error(ground_term,g(X)).
```

**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 `instantiation_error` or `existence_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  
— `instantiation_error`.
- b) `Global` is not a global variable  
— `existence(global_variable,Global)`.

**3.5.1.4 Examples**

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

```
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].
  On re-execution, the value returns to [a,b].

set_global(Z,g(b)).
  instantiation_error.

set_global(f(X),g(b)).
  existence_error(global_variable,f(X)).
```

**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 `instantiation_error` or `existence_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  
— `existence_error(global_variable,Global)`.

### 3.5.2.4 Examples

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

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

Succeeds. The goal unifies `X` with `[a,b]`.

```
global_value(f(a),[a,b,c]).
```

The goal fails since the value `[a,b]` of the global variable does not unify with `[a,b,c]`.

```
global_value(g(a),[a]).
```

```
existence_error(global_variable,g(a)).
```

### 3.5.3 current\_global/2

#### 3.5.3.1 Description

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

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

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

#### 3.5.3.2 Templates and modes

```
current_global(?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(global,X).
```

Succeeds, unifying `X` with `[a,b]`.

```
current_global(G,X).
```

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

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

On re-execution, fails.

```
current_global(g(X),f(a)).
```

Fails because no global variable unifies with `g(X)`.

```
current_global(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)`.

## 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) :- create_global(result,empty), 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 :- create_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 Fibonacci numbers

The following program is to compute  $N$ -th Fibonacci numbers.

```
fibonacci(1,1) :- !.
fibonacci(2,1) :- !.
fibonacci(N,X) :- N >= 3, N1 is N-1, N2 is N-2,
                fibonacci(N1,Y), fibonacci(N2,Z), X is Y+Z.
```

The computation of this program is inefficient, because it repeats same sub-computation. The following improved program uses global variables of the form `fib(N)` to hold intermediate results and avoid the repeated subcomputation.

```
fibonacci(1,1) :- !.
fibonacci(2,1) :- !.
fibonacci(N,X) :- current_global(fib(N),X),!.
fibonacci(N,X) :- N >= 3, N1 is N-1, N2 is N-2,
                fibonacci(N1,Y), fibonacci(N2,Z), X is Y+Z,
                create_global(fib(N),X).
```

## 4.4 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,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),
    new_mutable(E,I1), K1 is K+1, initialize(K1,T,N,I1).

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

```

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),
    set_mutable(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.



```

    [ new_mutable/2,
      mutable/1,
      set_mutable/2,
      mutable_value/2,
      create_global/2,% +Var, +Value
      set_global/2,% +Var, +Value
      global_value/2,% +Var, -Value
      current_global/2 % ?Var, ?Value
    ]).

:- use_module(library(error)).

%% mutable(M) is semidet.
%
% True if M is a mutable term.  Type-check.

mutable(M) :-
    compound(M), functor(M, '$mutable', 1).

%% new_mutable(-M, +Value) is det.
%
% Create a new mutable term.
%

new_mutable(M, Value) :- var(M),
    M = '$mutable'(box(Value)).

%% set_mutable(+M, +Value) is det.
%
% Set the value of the mutable term M to Value.
%

set_mutable(M, Value) :-
    ( mutable(M)
      -> setarg(1, M, box(Value))
      ; type_error(mutable, M)
    ).

%% mutable_value(+M, ?Value) is semidet.
%
% True if Value is the current value of the mutable M.

mutable_value(M, Value) :- nonvar(M),
    ( M = '$mutable'(box(Value0))
      -> Value = Value0
      ; type_error(mutable, M)
    ).

```

```

    ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   Global Variables
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% create_global(+Var:atom, +Value:term) is det.
%
% Create a new global variable with given Value. Intended to be
% used in directives to define the available global variables.
% @error permission_error(global, Var) if the global exists.

create_global(Var, Value) :- ground(Var), term_to_atom(Var,Atom),
    (nb_current(Atom, V), V \== [] -> permission_error(create_global,Var, V); true),
    b_setval(Atom, Value).

%% set_global(+Var, +Value) is det.
%
% Set the value of a global variable.
%
% @error existence_error(global, Var) if the global does not
% exist.

set_global(Var, Value) :- ground(Var), term_to_atom(Var,Atom),
    b_getval(Atom, _),% Demand existence
    b_setval(Atom, Value).

%% global_value(+Var, ?Value) is semidet.
%
% True if Value is the current value of the global variable Var.
%
% @error existence_error(global, Var) if the global does not
% exist.

global_value(Var, Value) :- ground(Var), term_to_atom(Var,Atom),
    b_getval(Atom, Value).

%% current_global(?Var, ?Value) is nondet.
%
% True if Value is the current value of the global variable Var.

current_global(Var, Value) :- ground(Var),!, term_to_atom(Var,Atom),
    nb_current(Atom, Value), Value \== [].

```

```
current_global(Var, Value) :-  
    nb_current(Atom, Value), Value \== [], term_to_atom(Var,Atom).
```