

# Global variables in HAL, a logic implementation

*Bart Demoen*

*María García de la Banda*

*Kim Marriott*

*Peter Schachte*

*Peter Stuckey*

*Report CW 271, September 1998*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Global variables in HAL, a logic implementation

*Bart Demoen*

*María García de la Banda*

*Kim Marriott*

*Peter Schachte*

*Peter Stuckey*

*Report CW271, September 1998*

Department of Computer Science, K.U.Leuven

## **Abstract**

HAL is a new logic language that makes it easy to implement constraint solvers. HAL gets most of its efficiency from compiling to Mercury code. The main mismatch between HAL and Mercury is that HAL supports variables a la Prolog, while Mercury recognises basically only the instantiations new and ground. We describe here the schema that overcomes this mismatch: it relies on a Parma representation of variables. Its main advantage is that once a datastructure is ground, it has the same internal representation as a Mercury ground term. Experiments show that our schema is very competitive with any other logic implementation that supports unbound terms. We also discuss the implementation of delay for the Herbrand solver.

# Global variables in HAL, a logic implementation \*

Bart Demoen (1), María García de la Banda (2),  
Kim Marriott (2), Peter Schachte (3), Peter Stuckey (3)  
(1) K.U.Leuven `bmd@cs.kuleuven.ac.be`  
(2) Monash University `mbanda,marriott@cs.monash.edu.au`  
(3) Melbourne University `pets,pjs@cs.mu.oz.au`  
`bmd@cs.kuleuven.ac.be`  
`{mbanda,marriot}@cs.monash.edu.au`  
`{pets,pjs}@cs.mu.oz.au`

## Abstract

Recently, there has been interest in the use of (backtrackable) global variables in Prolog: they usually are presented as syntactic sugar for threaded variables and therefore their semantics is logical. However, the implementation of global variables through program transformation has some drawbacks: it is difficult to combine with separate compilation of modules and with higher order predicates. There is also a performance problem with the transformed program if the added variables are only infrequently accessed. We here describe another implementation of such global variables, within SICStus 3.0: the changes to the system are extremely small. Backtrackable global variables in HAL - a logic language with support for writing constraint solvers - allow for a particular efficient implementation of backtrackable global variables: access to their value is constant time. We also describe another type of global variables: they are non-backtrackable and their semantics depends on the execution strategy.

## 1 Introduction

See [4] and [3] for seminal work and explanation on global variables. In [4], [3] global backtrackable variables (GBVs) are implemented by program transformation: variables are added to predicates that use directly or indirectly GBVs. We call this implementation "local" because it changes the program locally: still, as in [4], it needs global analysis to perform this transformation in an optimal way.

Hidden Accumulation Grammars (HAGs) [6] offer another implementation to a similar construct: BinProlog [7] has a global (i.e. visible from all Prolog code) table (implemented as a hash table) with an association between atoms and a value. This might look similar to the functionality that is offered by the record-predicates in DEC-10 compliant Prolog implementations, but there are two important differences: the BinProlog table is fully backtrackable (ie. any change to the table is undone on backtracking over the change) and more importantly, the values associated to the atoms, reside on the heap, so that on accessing the value, it needs not to be copied to the heap. The former difference can be mimicked in SICStus with the `undo/1` predicate, the latter is not readily possible. Still, the fact that values reside on the heap - making access independent of the size of the value - can be crucial in many cases. Paul Tarau has argued often that GBVs can be

---

\*This is a HAL working document - started July 1997, finished September 1998

implemented with HAGs: indeed, the semantics of GBVs is the same whether implemented with HAGs or with threaded variables. We call the implementation with an association table (like in BinProlog or similar ones) "global" because the association table is globally accessible. However, it has never been clear how such a table can be easily implemented in another Prolog system.

We show here two global implementations of GBVs in SICStus: both are based on two extremely simple primitives written in C<sup>1</sup>; the first is dynamic in nature (i.e. new GBVs can be created at runtime) while the second exploits the restrictions on GBVs in HAL (see later) that all GBVs are known at compile time; this results in truly constant time (and low cost) access of values associated to GBVs. Our motivation for this work is in the HAL project [1]: its aim is to design and implement a logic language in which it is easy to program (constraint) solvers and with close to Mercury performance; more details can be found in <http://www.cs.kuleuven.ac.be/bmd/HAL>. Release 0.1 of HAL compiles to SICStus and work on compiling to Mercury is under progress: eventually, we will implement GBVs also in Mercury, and the current implementation of GBVs in SICStus serves as a preparation.

HAL, as Prolog, supports dynamic loading of modules, i.e. not all modules that are used during a run of a program are known at the moment the program starts. The semantics of dynamic loading of modules is quite clear, until the moment GBVs are introduced: it is one of our main concerns to make the semantics of dynamic loading of modules concise and correctly implemented.

HAL supports the `scoped_to` construct (this construct was hinted at in [4]): we will show how our implementation caters for it straightforwardly.

We assume the reader is familiar with SICStus 3.0, in particular with mutable objects. The SICStus documentation contains enough info. We will abuse this paper to introduce some HAL programming concepts.

## 2 Backtrackable global variables in HAL

Global variables in HAL must be declared in the module they are scoped to. This is done with a declaration like:

```
:- glob_var X is list(int) init [1,2,3].
```

where `X` is the name of the global variable, `list(int)` its type and `[1,2,3]` its initial value. The initial value is optional. (The current value of) `X` can be used in 3 contexts:

- assignment: `$X := <term>`
- unification: `$X = <term>`
- scoped execution: `[$X] scoped_to Goal`

Their exact semantics can be found in [1], but it is rather intuitive and similar to [4]. E.g. the semantics of the initial value in the `glob_var` declaration is that on first usage of `X`, it will have that initial value.

HAL supports dynamic loading of modules: this is needed for the compiler, as it loads during the compilation the solver normalisation (and in the future the solver analysis); the module to be loaded, is only known at a moment that the compiler has already performed some computation, so the loading cannot be done at startup time of the compiler. It was felt that the semantics of

---

<sup>1</sup>We have only added some 25 lines of code to `initial.c`

loading a module should not depend on the moment at which this loading takes place, whether it is backtracked over or whether it happens more than once (the system could decide in a transparent way to unload a module and reload it later). As the semantics of modules loaded once and before any computation is started seems quite clear and easy to understand, we decided that the semantics of loading that takes place later, must be implemented with the exactly same semantics, i.e. as if the loading had taken place once in the beginning.

One of the guiding principles in the SICStus implementation of GBVs, was not to make any drastic changes to SICStus: this precluded e.g. putting heap terms "outside" the usual heap or using the (destructive) trailing mechanism for entities not on the heap. The former is useful e.g. for avoiding repeated copying of ground objects; the latter for implementing the backtrackable association table as in BinProlog. Using implementation constructs in such non-intended ways, can confuse tidy-trail (on execution of cut), the range tests during unification, marking and early reset during garbage collection of the heap - and possibly affects many other places which assume that pointers are within the heap and local stack.

We need to keep an association between the names of the global variables and their value - as abstracted by a pointer to the heap. The values themselves live on the heap because they can be partially instantiated and also because they are backtrackable. The association data structure thus contains pointers to the heap and these pointers have to be followed by the marking phase of a garbage collector and on backtracking over the creation of a global variable set to uninitialised: this would not be a problem if all loading happened before any execution, but above, we have shown the need to give a similar semantics to modules that are loaded later. From this it follows that the association data structure must also reside on the heap and must itself be created before any execution takes place. Since we want to focus on the principle first, let's take as association data structure just an open ended list: its elements are pairs *assoc*(< *variablename* >, < *value* >) in which the < *variablename* > is bound. The first implementation of global backtrackable variables then consists of:

```
gv_init :- '$gv_init'(_).
```

gv\_init/0 must be called before and in conjunction with any query. '\$gv\_init'/1 is implemented in C roughly as

```
gv_assoc_table = X(0); /* X(0) is the first WAM argument register */
```

where gv\_assoc\_table is a TAGGED global C variable.

To set the value of a global variable:

```
gv_set(Name,Value) :-
    '$gv_get_assoc_table'(Table),
    member_check(assoc(Name,OldValue),Table),
    (var(OldValue) ->
        create_mutable(Value,OldValue)
    ; update_mutable(Value,OldValue)
    ).
```

Here, '\$gv\_get\_assoc\_table'/1 unifies its argument with the global C variable gv\_assoc\_table. To get the value of a global variable:

```

gv_get(Name,Value) :-
    '$gv_get_assoc_table'(Table),
    member_check(assoc(Name,OldValue),Table),
    (var(OldValue) ->
        error('no value for gv'(Name))
    ; get_mutable(Value,OldValue)
    ).

```

We have to cater for initialisation of global variables and for error detection when a global variable is accessed before it is initialised (remember that the `init` part in the `glob_var` declaration is optional). This can be achieved by compiling the above `glob_var` declaration to:

```

:- multifile '$gv_initial_value'/2.
'$gv_initial_value'('X',[1,2,3]).

```

and adapting `gv_get` as follows:

```

gv_get(Name,Value) :-
    '$gv_get_assoc_table'(Table),
    member_check(assoc(Name,OldValue),Table),
    (var(OldValue) ->
        ('$gv_initial_value'(Name,InitValue) ->
            InitValue = Value,
            create_mutable(Name,Value)
        ; error('no value for gv'(Name))
        )
    ; get_mutable(Value,OldValue)
    ).

```

Incidentally, this has also solved the problem of modules loaded later: when backtracking occurs over the first occurrence of a global variable, the initial value will be reinstalled on the next "first" use.

The above association data structure is not very satisfying: its complexity is too high. Instead of an open-ended list, one could use an ordered tree or a hash table: we shortly describe the latter, completely implemented in Prolog. The hash table resides on the Prolog heap and still uses the above described `'$gv_get_assoc_table'/1` and `'gv_init'/0`.

```

gv_init :-
    functor(Table,'$gv_hashtable',251), % 251 must be a prime number
    prolog:'$gv_init'(Table).

```

```

gv_get(Name,Val) :-
    gv_get_entry(Name,Entry),
    % Entry is either free or [Name|MutableValue]
    (var(Entry) ->
        error('no value for gv'(Name))
    ; Entry = [_|MutableValue],
        get_mutable(Val,MutableValue)
    ).

```

```

gv_set(Name,Val) :-
    gv_get_entry(Name,Entry),
    % Entry is either free or [Name|MutableValue]
    (var(Entry) ->
        Entry = [Name|MutableValue],
        create_mutable(Val,MutableValue)
    ; Entry = [_|MutableValue],
        update_mutable(Val,MutableValue)
    ).

gv_get_entry(Name,Entry) :-
    prolog: '$gv_get_hash_table'(HashTable),
    functor(HashTable,_,Size),
    prolog: '$term_hash'(Name,-1,Size,HashVal),
    Loc is HashVal + 1,
    search_name(Name,HashTable,Size,Size,Loc,Entry).

search_name(Name,HashTable,Size,ToDo,Loc,Entry) :-
    arg(Loc,HashTable,E),
    (var(E) ->
        E = Entry
    ; E = [N|_],
        (N = Name ->
            E = Entry
        ; (ToDo < 1 ->
            write('gv hash table full - gv not added'(Name)), nl, fail
        ; NewToDo is ToDo - 1,
            NL is Loc + 7, % 7 is prime
            (NL > Size ->
                NewLoc is NL - Size
            ; NewLoc = NL
            ),
            search_name(Name,HashTable,Size,NewToDo,NewLoc,Entry)
        )
    ).

```

This implementation of the hash table does not cater directly for table expansion (and the arity restriction of SICStus to 255 is rather annoying), but by making the table itself a mutable and a bit more work, this can be achieved.

One might think that one could use directly the library `assoc` which implements AVL-trees: one has to be, because the global C-variable `gv_assoc_table` cannot be (destructively) trailed (it doesn't reside on the heap nor local stack). An adaption of the `assoc` library however does the job.

The implementation we prefer for HAL, uses the fact that in HAL all GBVs are known at compile time. This makes it possible to associate with each GBV an integer  $> 0$  at load time, and use this integer for direct access in an array. This is achieved by translating the declaration of a GBV, e.g. `:- glob_var X is list(int) is [1,2,3].` to the SICStus directive

```
:- r_gv_init_value('X',[1,2,3]).
```

The implementation of `remember_gv_init_value/2` is

```
remember_gv_init_value(Name,InitValue) :-
    get_next_integer(N),
    assert(associate(Name,N)).
```

```
:- dynamic current_integer/1.
current_integer(1).
```

```
get_next_integer(N) :-
    retract(current_integer(N)),
    M is N + 1,
    assert(current_integer(M)).
```

We also define `term_expansion/2`, whose main function will be to transform a goal like

```
$X := <term>
```

to the goal

```
gv_set(<NX>,'X',<term>)
```

where `<NX>` is the integer associated to `$X`, and similar for `gv_get`. The implementation of `gv_get/3` now becomes

```
gv_get(Int,Name,Val) :-
    '$gv_get_assoc_table'(Table),
    arg(Int,Table,Entry),
    (var(Entry) ->
        error('no value for gv'(Name))
    ; get_mutable(Val,Entry)
    ).
```

and `gv_set` looks similar.

It is clear that this will give constant time and low cost access to GBVs in HAL: the limitation of the arity of compound terms in SICStus is a nuisance at the moment, but this will not be a problem later in Mercury.

### 3 The `scoped_to` construct for BGVs

HAL features a construct which scopes BGVs to a particular goal:

```
[$X] scoped_to Goal
```

means that the value of `$X` before and just after the execution of `Goal` is the same and that when the execution backtracks into `Goal`, the old value of `$X` is restored. Given one of the above implementations of BGVs, the `scoped_to` construct is simply translated to SICStus as:

```

gv_get('X',ValBefore),
call(Goal),
gv_set('X',ValBefore)

```

It has been suggested that the variables that are not initialized in the `scoped_to` construct, are initted to `uninit` so that subsequent use is detected as error. At this point, there has not been a decision on this issue: in `[$X] scoped_to Goal`, the `$X` has a value that one wants to compute with in `Goal` to start with, although one wants it reset after `Goal` is finished; maybe one could write for that `[$X := $X] scoped_to Goal` but that looks ugly.

## 4 More implementation issues related to GBVs

Since GBVs reside on the heap, one has to consider their (heap) garbage collection: this is not a moot point, as the reachability of GBVs in the global implementation, does not translate into reachability of ordinary variables. A simple example illustrates this:

```

a :- $X := f(9), b.
b :- write($X).

```

The term `f(9)` is reachable from the body of `b/0`, but there is no X- or Y-variable (in terms of WAM) that embodies this reachability. The garbage collector in SICStus is rather precise, but it is also conservative in two more respects: it approximates conservatively the reachability of intermediate values of mutables (see section 4.1) and it does not rejuvenate garbage as in [2]. If it were more precise, the term `f(9)` might be collected by the garbage collector before `b/0` is entered (resulting in the output of garbage :-). Fortunately, the remedy is easy in SICStus: all we have to do is make sure that the association table is marked during garbage collection; this is achieved by putting the (entry to the) table in the foreign language interface stack (`fli_stack`).

### 4.1 Garbage collection of mutables

The fact that the SICStus garbage collector approximates conservatively the reachability of intermediate values of mutables, is exactly why we have not to do more about early reset: the following example shows a situation in which the usual reachability of data structures together with a more aggressive identification of garbage would have caused problems:

```

a :-
    $X := f(1),
    ( $X := f(2), gc
    ; true
    ),
    write($X).

?- a.

```

At the moment of garbage collection (call to `gc/0`), there is no "classical" sense in which the structure `f(1)` is alive and one could rightfully expect a precise garbage collector to collect it; however, it is fortunate for us that SICStus does not collect this term. On the other hand, a very local change in `heapgc.c` would solve this problem otherwise.

## 5 Comparison with threaded variables

Backtrackable global variables can be implemented by threading variables [4, 3]. This has advantages and disadvantages: threading variables is difficult to combine with separate compilation and with higher-order predicates. The former problem occurs when module `foo` uses module `bar` and the implementation of `bar` changes in that it starts using global variables, then module `foo` must be recompiled, even if the interface of `bar` doesn't change. We think this is a serious drawback of threading, especially since in HAL, constraint solvers will typically keep their solver state in global variables, and it is rather inconceivable that an application must be recompiled because the solver implementation has changed. Higher-order predicates are problematic for threading variables, because the analysis - as described in [4] - is in general inaccurate and therefore the safe approximation consists in assuming that all global variables have to be threaded through, resulting in unnecessary overhead. A global implementation as described in section ]refsect2 solves both problems. However, several caveats remain: goal reordering, efficiency ...

### 5.1 Goal reordering in the presence of global variables.

The context being HAL, means that conjunctions (and disjunctions) can be reordered as long as the modes are respected: this is exactly like in Mercury. With the semantics of global variables given by threading variables (i.e. by a program transformation), the goal reordering phase in the compiler clearly comes **after** the transformation phase: the program transformation must decide about the modes of the added arguments, and in this way implicitly restricts the allowed reordering. Without threading variables, no further restrictions are put on reordering, however different reorderings will result in different results computed by the program; a representative example is:

```
a :- b, c, ground_write($X).
b :- $X := 1.
c :- $X := 2.

?- a.
```

Depending on the goal reordering in predicate `a/0`, the resulting output is 1 or 2 (assuming that the mode of `ground_write/1` is "in"). It shows that the semantics of our implementation of global variables is not unique. This is true: the HAL semantics of the above program is indeed that either 1 or 2 is output. The situation is similar to the semantics in concurrent languages, where any valid execution path leads to a possible and correct answer: it is up to the programmer to ensure that every possible answer is intended. In HAL, the programmer is helped by the fact that she has explicit control over reordering.

We want to stress that we believe that the issue of reordering in the presence of global variables, is not "in contradiction" with our proposed implementation method: threading restricts the semantics in an evenly unlogical way, because (simplifying the issue) it imposes implicitly the selection strategy to left-to-right.

### 5.2 Efficiency.

It is very easy to give examples in which threading is more efficient than our implementation, and vice versa: it might appear that within a strongly connected component of the call graph which does not surpass the module boundary (and which consequently has no higher-order predicates

which cannot be analysed), threading of the global variables might be more efficient than the global implementation method, but whether this is really true depends on generally undecidable properties like failure, multiple solutions, frequency of activation, granularity ... However, whichever way GBVs are implemented, the following code

```
a :-
    ($X = 0 ->
      true
    ; Y is $X - 1,
      $X := Y,
      dosomething, % does not change $X
    a
  ).
```

should lead to the following generated code

```
/* pickup $X from threaded variable or from association table */
register int i = $X;
while (i-- != 0)
    dosomething;
$X = i;
```

So, there is no point in trying to decide for one implementation method or the other. When intermodule or higher-order calls are involved, the global implementation method seems most appropriate. But for pieces of code inside one module and without higher-order calls, threading can be the more efficient implementation, depending on the qualities of the compiler and/or properties of the program which might have to be approximated by analysis.

## 6 Discussion

We have shown an alternative implementation of backtrackable global variables in SICStus: this implementation can be carried over to any implementation which has a destructive update trail. Eventually, there will be an implementation of Mercury with such a trail, to support the concepts we desire. The design of the Mercury implementation of GBVs is already done and guarantees the same access time to a GBV as C has to a static variable: it is only a minor variation on the third implementation method in section 3. The global implementation of GBVs caters for separate compilation and higher-order calls, but is locally not necessarily most efficient: a new analysis needs to be developed to allow for a more flexible implementation that switches between the two.

## Acknowledgements

The first author was a guest at the University of Melbourne and the University of Monash at the time of this research. All authors acknowledge support from the HAL project ARC Large Grant.

## References

- [1] Bart Demoen, María García de la Banda, Kim Marriott, Peter Schachte, Peter Stuckey *HAL: a highly attractive language for writing (constraint) solvers* Documentation Manual release 0.1 - June 1997
- [2] B. Demoen, G. Engels, and P. Tarau. *Segment Preserving Copying Garbage Collection for WAM based Prolog*. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386, Philadelphia, Feb. 1996. ACM Press.
- [3] Andreas Kägedal *Logical State Thread Package* Available from the author
- [4] Peter Schachte *Global variables in Logic Programming* In *Proceedings of the 14th International Conference on Logic Programming*, pages 3–17, Leuven, Belgium, July 1997. The MIT Press.
- [5] Zoltan Somogyi, Fergus Henderson and Thomas Conway. *The execution algorithm of Mercury: an efficient purely declarative logic programming language*. *Journal of Logic Programming*, volume 29, number 1-3, October-December 1996, pages 17-64.
- [6] Paul Tarau, Vernoica Dahl and Andrew Fall. *Backtrackable state with linear affine implication and Assumption Grammars* In LLNCS, Proceedings of the Asian Computing Science Conference, Singapore, December 1996, Springer
- [7] Paul Tarau *BinProlog 5.25 User Guide* Departement d'Informatique, Universite de Moncton, Moncton, Canada. Part of the BinProlog 5.25 distribution, available from <ftp://clement.info.umoncton.ca/BinProlog>