UPMAIL Technical Report No. 40

Hash Tables in Logic Programming

Jonas Barklund & Håkan Millroth

Uppsala Programming Methodology and Artificial Intelligence Laboratory
Computing Science Dept., Uppsala University
P. O. Box 520, S-751 20 UPPSALA, Sweden
+46–18–18 25 00

Abstract

The paper discusses different aspects of hash tables as a data structure in a Horn clause language such as Prolog, e.g., semantics, implementation and applications.

Hash tables are a new concept in Prolog since they can not be efficiently implemented in the language itself. We give informal semantics for hash tables as partial functions on logical terms. We are careful to ensure that we can represent the hash tables as logical terms in order to make the ordinary syntactic unification apply.

The implementation is a generalisation of the method suggested by Eriksson and Rayner. Their "mutable arrays" are a special case of our multiple version hash tables. The implementation has been used in Tricia, an implementation of Prolog developed at Uppsala University.

Hash tables open application areas previously beyond the efficiency limits of Prolog. Application programs taking great benefit from the hash tables have been written in Tricia.

An earlier version of this paper was presented at the 4th International Conference on Logic Programming.

The research reported herein was supported by the National Swedish Board for Technical Development (STU).

1. Introduction

Many data structures can be implemented as logical terms. However, some operations on data structures may turn out to be very expensive if the data structures are represented too naively in the machine.

A striking example are predicates on two terms X_1 and X_2 saying that X_2 is like X_1 except for some small difference. The naive approach to generate X_2 given X_1 is to make a copy of X_1 and then change the copy according to the stated difference. The reason for doing the changes in a copy of the term is of course to ensure the property that two occurrences of the same variable should stand for identical objects. If the relation above holds for two terms X_1 and X_2 we say that X_2 is a *new version* of X_1. Defining such *update* predicates is of course possible for all logical terms, but to emphasize the data structures where such predicates are implemented efficiently we call them *multiple version* data structures (MVDS).

There have been several proposals, e.g., [Warren 83b; Cohen 84; Eriksson & Rayner 84], how to incorporate multiple version arrays into Prolog. However, less attention has been directed to other MVDS [Cohen 84; Pereira 85].

Most proposals for implementation of multiple version arrays can claim superiority over the others at some point. We think that the scheme given in [Eriksson & Rayner 84] for LM-Prolog [Kahn 83; Carlsson 84] is the one with the best performance for average programs, besides being reasonably easy to implement. The idea is to implement arrays in terms of *physical arrays* and *virtual arrays*. In Tricia, a new implementation of Prolog on the DEC-2060 computer developed at Uppsala University [Barklund *et al.* 86; Barklund & Millroth A], we have implemented multiple version arrays following their scheme (with minor modifications) and then applied the idea to multiple version hash tables (MVHT). These arrays and hash tables are heavily used in the internals of the runtime system (symbol tables, the internal data bases etc.), but are also accessible for users by means of built-in predicates. The reason for implementing MVHT was that hash tables are very powerful data structures (included in Common Lisp [Steele 84]), which are difficult, if not impossible, to implement efficiently in Prolog itself. Also MVHT are complex enough to serve as an example on how to implement non-trivial MVDS.

This paper has five goals:

1. to introduce a new data structure into Prolog: multiple version hash tables,
2. to give informal semantics for MVHT,
3. to supply formal definitions for predefined predicates on MVHT in Horn clauses,
4. to describe an efficient implementation of MVHT,
5. to suggest some applications for MVHT.

2. Semantics

2.1 Informal semantics

When solving problems by computer programming it is very common to use tables indexed by keys in some domain. Given such a table we can look up the value for a certain key. Naturally, manipulations of such tables, such as addition or deletion of key-value pairs are also frequent.

More abstractly, such tables can be viewed as functions from the domain of the keys to the domain of the values. Since the domain of the keys (e.g., character strings or integers) is usually infinite but the tables are finite, it is evident that for a certain table there must be keys for which it contains no value. This implies that the functions represented by tables must be partial functions: the values for some keys are undefined. Below we will sometimes ignore the distinction between tables and the functions which they tabulate. Thus, if T is a table containing the key k we let $T(k)$ stand for the value stored for k in T.

There are many ways of storing such tables in a computer: if the key domain is a finite interval of integers then arrays are commonly used. Other storage techniques include sequential tables, linked lists, trees and hash tables.

When storing tables as logical terms, the most common techniques are to employ lists or trees, or to use the internal database (storing tables as predicates). Both lists and trees work fine for small tables. However, updating the tables usually becomes expensive operations since logical terms are not modifiable (for certain problems it is possible to employ coding tricks to improve the behavior, e.g., using lists with uninstantiated

tails or trees with uninstantiated fringes, but this is neither generally applicable or elegant). Using the internal database is fine until modification of tables is considered, as this involves asserting and retracting clauses. The semantics of such modifications are not solved satisfactorily.

Because of the restrictions imposed on the functions represented by tables it is possible to represent them by ordinary first-order logical terms such that the functions are equal if and only if the terms representing the functions are unifiable. One such representation is sorted A-lists. It is not difficult to write Horn clause predicates for table lookup or modification where tables are represented by A-lists. They are a nice representation of tables in every respect except that they are inefficient for anything but small tables.

The efficiency problem is an intensional property of A-lists, not of tabulated functions as such. The solution is to find another representation for them which is as abstract as sorted A-lists but more efficient. We propose to use hash tables for storing tables in logic programs. They can not be implemented efficiently as logical terms so it is necessary to implement them at a lower level. However, this does not necessarily change the langauge as we will show below.

2.2 Unification

In Prolog two logical terms unify if they are syntactically equal or if they can be made syntactically equal by instantiating variables in the terms. This definition is nice since it is decidable. Different kinds of *semantic unification* have been proposed, usually for amalgamations of logic and functional programming languages. This means introducing a new equality theory where two terms may be considered unifiable although they look different. Such languages may become important in the future but now their properties are not very well known, so we do not want to abandon the normal syntactic unification for hash tables.

Following this, if we introduce a new data structure we should always be able to give a representation of this data structure as logical terms, such that two instances of the data structure are supposed to unify iff the logical terms representing them unify. We should then be able to define any primitive operations on the representation of the data structure in Prolog, thereby giving the primitives declarative semantics (if the definitions are given in sufficiently pure Prolog) or at least semantics in terms of Prolog. We call this the *syntactic unification condition* for data structures.

We think two hash tables T_1 and T_2 should be considered unifiable if they contain exactly the same set K of keys and for every key $k \in K$ we have that $T_1(k) = T_2(k)$. This is in accordance with the behavior of the representation of tables as sorted A-lists above. Nothing is added to the language, we only give an alternative implementation of some logical terms.

In [Pereira 85] unification of a similar data structure (*DAG*s) is discussed and another view on unification of tables is expressed: unification of two DAGs T_1 and T_2 corresponds to taking the set union of key-value pairs in T_1 and T_2, failing only if for some key k both in T_1 and T_2 we have that $T_1(k) \neq T_2(k)$. For example, if

$$T_1 = \{a \to 1, b \to X, c \to f(42)\}$$
$$T_2 = \{b \to 2, c \to f(Y), d \to 4\}$$

then after unification T_1 and T_2 both become

$$\{a \to 1, b \to 2, c \to f(42), d \to 4\}.$$

If such tables were to be represented as logical terms, the terms would have to contain values for every possible key (uninstantiated for all keys without explicitly given values), so a table with no value for some key K could unify with one having a particular value for K. There is no finite representation of such tables as logical terms, so the syntactic unification condition is violated and the language is changed.

If one is deliberately changing the language to include e.g., a non-standard equality theory, and semantic justifications can be given, then DAG unification is an interesting extension. We have been told that J. Goguen has came up with results in this field but as we said above, we do not want to extend the language at this point and we think we can achieve many interesting properties without DAG unification.

Our solution is to avoid augmenting unification with DAG unification and as an alternative we propose the introduction of a ternary predicate over terms T_1, T_2 and T_3 which holds if T_3 is the union of T_1 and T_2 as in DAG unification, but without affecting T_1 and T_2. (This predicate is called `mergehash/3` below).

3. Formal Definitions for Predefined Predicates

We now give a representation of tables as logical terms and give the semantics for the predefined predicates manipulating hash tables in terms of these. (However, if the predicates are run in Prolog, solutions from the actual implementation may be delivered in another order than solutions from the predicates defined below. Also not all modes for calling the predicates are supported by the implementation).

We use a representation of tables equivalent to sorted A-lists: empty hash tables are represented by the reserved atom `empty_ht` and non-empty hash tables by ternary structures with the reserved functor `ht/3`. A table containing

$$\{K_1 \rightarrow V_1, K_2 \rightarrow V_2, \ldots, K_n \rightarrow V_n\}$$

where $\forall i\{K_i < K_{i+1}\}$ would be represented by a term

$$\texttt{ht(}K_1\texttt{,}V_1\texttt{,ht(}K_2\texttt{,}V_2\texttt{,...))}.$$

Since we use reserved functors the tables will not unify with any other terms. With this representation the syntactic unification condition holds.

[Barklund & Millroth B] investigates further methods of fully integrating MVDS and other internally complicated data structures as ordinary Prolog terms with syntax, unification etc.

3.1 Primitive Predicates

empty_hash_table(Table)

`Table` is unified with an empty hash table. This is used to create a hash table or to test if it is empty.

```
empty_hash_table(empty_ht).
```

gethash(Table,Key,Value)

This is true if `Key` has value `Value` in `Table`. This is used to look up items in the table. `Table` must be instantiated but `Key` may be uninstantiated. If `Key` is not ground, all `Key-Value` pairs will be tried. An alternative would be to freeze the call [Naish 85] until `Key` is ground but this can be programmed using other primitives [Carlsson 86].

```
gethash(ht(K,V,_),K,V).
gethash(ht(_,_,T),K,V) :- gethash(T,K,V).
```

addhash(Table_1,Key,Value,Table_2)

This is true if `Table_1` and `Table_2` are identical except that `Table_1` contains no value for `Key` but `Table_2` maps `Key` to `Value`. This is used to add or remove items from the table. Either `Table_1` and `Key` or `Table_2` should be instantiated.

```
addhash(empty_ht,K,V,ht(K,V,empty_ht)).
addhash(ht(K1,V1,T1),K2,V2,ht(K2,V2,ht(K1,V1,T1))) :- K1 @> K2.
addhash(ht(K1,V1,T1),K2,V2,ht(K1,V1,T2)) :- K1 @< K2, addhash(T1,K2,V2,T2).
```

mergehash(Table_1,Table_2,Table_3)

This is true if every key with a value in `Table_1` or `Table_2` has the same value in `Table_3`. If a key has values in both `Table_1` and `Table_2` the values are unified. This combines information from two hash tables and simulates Pereira's DAG unification. `Table_1` and `Table_2` must be instantiated.

```
mergehash(empty_ht,T,T).
mergehash(T,empty_ht,T).
mergehash(ht(K,V,T1),ht(K,V,T2),ht(K,V,T3):- mergehash(T1,T2,T3).
mergehash(ht(K1,V1,T1),ht(K2,V2,T2),ht(K1,V1,T3):-
   K1 @< K2, mergehash(T1,ht(K2,V2,T2),T3).
mergehash(ht(K1,V1,T1),ht(K2,V2,T2),ht(K2,V2,T3):-
   K1 @> K2, mergehash(ht(K1,V1,T1),T2,T3).
```

is_hash_table(Object)

This is true if `Object` is currently instantiated to a hash table. To be consistent with the behaviour of `atom/1` and other type-testing predicates it should probably fail if `Object` is not instantiated.

```
is_hash_table(empty_ht).
is_hash_table(ht(K,V,empty_ht)).
is_hash_table(ht(K1,V1,ht(K2,V2,T))) :- K1 @< K2, is_hash_table(ht(K2,V2,T)).
```

recent_hash_table(Table)

For all purposes except efficiency this predicate is equivalent to the predicate `is_hash_table/1` above. The internal structure of `Table` is changed so the version represented by `Table` becomes the most efficient to use. `Table` should be instantiated.

```
recent_hash_table(T) :- is_hash_table(T)
```

3.2 Convenient Predicates

The following predicates do not define any primitive predicates but are useful. They can easily be defined in terms of the primitive predicates but can also be implemented more efficiently.

modhash(Key,Table_1,Value_1,Table_2,Value_2)

This is used to change the value for a key in a table. Either `Table_1` or `Table_2` should be instantiated.

puthash(Key,Table_1,Value,Table_2)

This is used to ensure that `Key` has a certain `Value` in `Table_2` which is otherwise like `Table_1`. `Key` and `Table_1` should be instantiated.

remhash(Key,Table_1,Table_2)

This is used to ensure that `Key` has no value in `Table_2` which is otherwise like `Table_1`. `Table_1` should be instantiated.

4. Implementation

4.1 Multiple Version Hash Tables

Our first approach was to implement MVHT in terms of multiple version arrays but we found that very much efficiency was lost by not taking advantage of the particular properties of hash tables.

The implementation of a MVHT consists of two parts: a linked structure (*chain*) containing values for particular keys and a physical table containing values for keys not in the chain. The last block in a chain (the *termination* block) points to the physical table but contains no key or value.

The purpose of the chain is to hide changes made to the physical table. A block R in the chain has three parts: a key $R.key$, a value $R.value$ and another MVHT $R.mvht$. Suppose that the value for a key k is searched in R. If R is not a termination block and $k = R.key$ then the contents of $R.value$ is returned, otherwise the value of k is searched in $R.mvht$. If R is a termination block then k is looked up in the physical table. The termination block is identified by having a special constant, distinguishable from all Prolog terms, in its key slot. We also have a way to record explicitly in the chain that some key has no value in a table (by storing a special constant distinguishable from all Prolog terms in the value slot of a block). This has no correspondence in [Eriksson & Rayner 84].

The algorithms for searching and updating a key are analogous to the algorithms given for multiple version arrays in [Eriksson & Rayner 84]. Our work can be viewed as a generalisation of theirs since their MVDS is a less powerful special case of ours. Since the method of using chains to mask values in the physical table is essentially the same as theirs, we do not discuss that idea further here.

The algorithms for updating hash tables ensure that the most recent version of a table never gets a longer chain than the previous version. This favors searches in (and updates to) the most recent version and we are convinced that this is optimal in almost any real application. This is similar to the idea of shallow variable binding in functional and logic programming languages [Baker 78; Miyazaki 85] and we have indeed used MVHT to implement interpreters for functional languages very comfortably.

The idea of a predicate which makes some version of a hash table the most efficient to use is due to Baker who uses a *shallow* primitive to give fast access to an old variable binding environment [Baker 78]. Our algorithm for doing this was inspired by his. In general, many of his ideas are similar to ours although he only considers having one physical table and has no direct counterpart to our primitives for updating tables.

The physical table is a hash table implemented according to one of the usual schemes [Knuth 73; Standish 80]. It is desirable that the scheme allows deletions from the physical table. The algorithms for the physical table are beyond the scope of this paper, see the references above for several useful techniques. We only assume the existence of primitives for looking up, adding and removing keys and for asking how many keys the table contains.

The algorithms in the Appendix assume the existence of routines to trail the old values of locations (not only unbound variables) in order to restore their old contents upon backtracking. For a description of how Warren's Abstract Machine [Warren 83a] is extended to do this, see [Barklund & Millroth A] (or [Turk 86] for an alternative solution).

4.2 Computing Hash Functions

One interesting problem to observe regarding the implementation of the physical hash tables, is the computation of *hash functions* (not to be confused with the functions in the semantics section above) for keys.

We suggest that the hash functions for atoms and functors should be computed when they are interned and that the computed values be stored in their cells. Applying the hash function on an interned atom or functor will then only amount to fetch the value stored in its cell.

We think this is necessary to implement logical theories as outlined below, since without it the process of finding the clauses for a certain predicate would probably be too expensive.

5. Applications for Multiple Version Hash Tables

We have suggested several applications above where the hash tables replace lists or trees by a more efficient data structure.

Replacing use of the internal database by use of hash tables has several advantages, e.g.:

1. Data can be modularized.
2. It is possible to refer to different versions of one's data base.
3. The values retrieved from hash tables will be identical to the values stored, while values retrieved from the internal database will be copies of the stored values (although the latter behaviour may sometimes be a feature).
4. Hash tables can be made more efficient since they need not create representations of the terms which are preserved on backtracking, although obviously the contents of hash tables are lost when backtracking.

5.1 Multiple Version Sets

Once having implemented MVHT, we get an implementation of sets (of ground terms) for free: as hash tables where the keys represent the members of the set and the values of the keys are ignored. Writing predicates for the usual operations on sets (union, intersection etc.) is straightforward and is left as an exercise for the reader.

When hash tables are used to represent sets we can save half the storage by not storing the values for the keys at all.

5.2 Multiple Version Logical Theories

In logic programming languages with meta-logical capabilities, MVHT can be used to implement multiple version logical theories. They can be implemented as mappings from functors to lists of clauses. Having logical theories as data structures makes it possible to reference them explicitly when reasoning about knowledge.

Almost regardless of other issues in meta-logical reasoning, efficiently implemented logical theories will be an important part of a system with meta-reasoning capabilities [Weyhrauch 80; Bowen & Kowalski 82; Bowen 85].

We also think that logical theories and meta-reasoning may play an important part in formalizing the concept of *frames*, commonly used in knowledge representation [Hayes 79].

Acknowledgements

We wish to express our gratitude to our colleagues at UPMAIL for interesting discussions and nice tea breaks, particularly to Amelie Banks who has substantially improved the language of the paper. The comments from one referee were also particularly valuable. This work would not have been possible without the continous support from our families.

References

[Baker 78] H. G. Baker, *Shallow Binding in Lisp 1.5*, Communications of the ACM, vol. 21, no. 7, pp. 565–569, July 1978.

[Barklund *et al.* 86] J. Barklund, L. Oestreicher, Å. Hugosson, M. Nylén, *Tricia User's Guide*, Computing Science Dept., Uppsala University, Sept. 1986.

[Barklund & Millroth A] J. Barklund, H. Millroth, *Code Generation and Runtime System for Tricia*, UPMAIL Technical Report 36, Uppsala, in preparation.

[Barklund & Millroth B] J. Barklund, H. Millroth, *Integrating Complex Data Structures in Prolog*, forthcoming.

[Bowen 85] K. A Bowen, *Meta-Level Programming and Knowledge Representation*, New Generation Computing 3, 1985.

[Bowen & Kowalski 82] K. A. Bowen, R. A. Kowalski, *Amalgamating Language and Metalanguage in Logic Programming* in K. L. Clark, S.-Å. Tärnlund, *Logic Programming*, London, 1982.

[Carlsson 84] M. Carlsson, *LM-Prolog — the Language and Its Implementation*, UPMAIL Technical Report 30, Uppsala, Oct. 1984.

[Carlsson 86] M. Carlsson, *Compilation for Tricia and its Abstract Machine*, UPMAIL Technical Report 35, Uppsala, Sept. 1986.

[Cohen 84] S. Cohen, *Multi-Version Structures in Prolog*, The International Conference of Fifth Generation Computer Systems, Tokyo, 1984.

[Eriksson & Rayner 84] L-H. Eriksson, M. Rayner, *Incorporating Mutable Arrays into Logic Programming*, Second International Logic Programming Conference, Uppsala, July 1984.

[Hayes 79] P. Hayes, *The Logic of Frames* in D. Metzing, *Frame Conceptions and Text Understanding*, Berlin, 1979.

[Kahn 83] K. Kahn, *Unique Features of LISP Machine Prolog*, UPMAIL Technical Report 15B, 1983 (revised by M. Carlsson Feb. 1985).

[Knuth 73] D. E. Knuth, *The Art of Computer Programming*, vol. 3, Reading, Mass., 1973.

[Miyazaki 85] T. Miyazaki, A. Takeuchi, T. Chikayama, *A Sequential Implementation of Concurrent Prolog based on the Shallow Binding Scheme*, 1985 Symposium on Logic Programming, Boston, July 1985.

[Naish 85] L. Naish, *Negation and Control in PROLOG*, Dept. of Computer Science, University of Melbourne, 1985.

[Pereira 85] F. C. N. Pereira, *A Structure-Sharing Representation for Unification-Based Grammar Formalisms*, 23rd Annual Meeting of the Association for Computational Linguistics, Chicago, Ill., July 1985.

[Standish 80] T. A. Standish, *Data Structure Techniques*, Reading, Mass., 1980.

[Steele 84] G. L. Steele Jr., *Common Lisp, the Language*, Burlington, Mass., 1984.

[Turk 86] A. K. Turk, *Compiler Optimizations for the WAM*, Third International Conference on Logic Programming, pp. 657–662, London, July 1986.

[Warren 83a] D. H. D. Warren, *An Abstract Prolog Instruction Set*, SRI Technical Note 309, Oct. 1983.

[Warren 83b] D. H. D. Warren, Logarithmic Access Arrays for Prolog, unpublished program, 1983.

[Weyhrauch 80] R. W. Weyhrauch, *Prolegomena to a Theory of Mechanized Formal Reasoning*, Artificial Intelligence, vol. 13, pp. 133–170, 1980.

Appendix

This appendix contains three examples of pseudo-coded algorithms for hash table lookup, modification and unification.

The following assumptions are made:

- Value blocks contains the slots `key`, `value` and `mvht`.
- The termination block in the value block chain contain the reserved constant `END-OF-CHAIN` in its `key` slot.
- The reserved constant `EMPTY` represents an empty entry in the table. In practice, anything that can be distinguished from all Prolog objects will do.

Hash Table Lookup

```
function lookup(Table,Key):   % For gethash/3.
% Search value block chain of Table for Key.
  while Table.key ≠ END-OF-CHAIN do {
    if Table.key = Key then
      if Table.value ≠ EMPTY then return(Table.value) else fail;
    Table := Table.mvht};
% Search physical table for Key.
  Table := Table.mvht;
  physical_table_lookup(Table,Key);
```

Hash Table Modification

```
function add_or_modify(Table,Key,Value):   % For puthash/4.
  New := alloc(3);
  Result := New;
% Copy value block chain, omit blocks containing Key.
  while Table.key ≠ END-OF-CHAIN do {
    if Table.key ≠ Key then {
      X := alloc(3);
      New.key := Table.key;
      New.value := Table.value;
      New.mvht := X;
      New := X};
    Table := Table.mvht;}
  X := alloc(3);  % Allocate termination block of the new chain
  P := Table.mvht;  % Physical table
% Allocate the old element in Table's termination block.
  Table.Key := Key;
  if physical_table_lookup(P,Key)
    then Table.value := physical_table_lookup(P,Key)   % Key had a previous value
    else Table.value := EMPTY;                         % Key had no previous value
  Table.mvht := X;     % Point to termination block of new chain
% The termination block of the new chain.
```

```
  X.key := END-OF-CHAIN;
  X.mvht := P;          % Physical table
```
If adding to table: `physical_table_update(P,Key,Value);`
If removing from table: `physical_table_update(P,Key,EMPTY);`
```
  return(Result)
```

Hash Table Unification

```
procedure table_unification(Table1,Table2):
  if number_of_elements(Table1) ≠ number_of_elements(Table2) then fail;
  T := Table1;
```
% Unify elements found in chain of `Table1` *with elements from* `Table2`*.*
```
  while T.key ≠ END-OF-CHAIN do {
    if T.value ≠ EMPTY then {
      V := lookup(Table2,T.key);
      unify(T.value,V)};
    T := T.mvht}
```
% Unify elements found in physical table of `Table1` *with elements from* `Table2`*.*
```
  T := T.mvht;     % Physical table
  for each Key and Value in T do {
    if not_in_chain(Table1,Key) then
      V := lookup(Table2,Key);
      unify(Value,V)}
```

Make most recent version

```
procedure recent(Table):
  <P,B> := rev(Table);
  Table.mvht := B;
  while P ≠ Table do {
    K := P.mvht.key; V := P.mvht.value;
    P.key := K;
    P.value := physical_table_lookup(B,K);
    physical_table_update(B,K,V);
    P := P.mvht}
  Table.key := END_OF_CHAIN;
```

Utility functions

`function alloc(N):`
Allocate a block of `N` *slots in memory and return a pointer to it.*

`function physical_table_lookup(Physical_table,Key):`
Standard hash table lookup using `Physical_table`*. Returns value if found, or fails.*

`procedure physical_table_update(Physical_table,Key,Value):`
Update `Physical_table` *so that it maps* `Key` *to* `Value`*.*

`function number_of_elements(Table):`
Returns the number of non-EMPTY elements in `Table`*.*

`function not_in_chain(Table,Key):`
Succeeds if there exists no block `T` *in the chain of* `Table` *such that* `T.key=Key`*.*

`function rev(Table):`
Destructively reverse chain of Table. Returns a pair `<R,B>` *where* `R`
is a pointer to the reversed chain and `B` *is a pointer to the physical table.*