UPMAIL Technical Report No. 42

Integrating Complex Data Structures in Prolog

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

Electronic mail: `JONAS@AIDA.UU.SE`, `HAKANM@AIDA.UU.SE`

Abstract. Computer Science has produced several data structures and algorithms on them to efficiently solve computational problems. Some algorithms require destructive operations for their efficient implementation or have other properties which make them difficult to implement efficiently in Prolog. If they are to be introduced in Prolog, these data structures and algorithms must be implemented at a lower level.

We investigate how such data structures, possibly with a complicated internal representation, can be naturally incorporated as first class terms in Prolog. Implementation problems are discussed for a number of alternative solutions. Four examples of data structures: (hash) tables, arrays, characters, and strings are examined individually.

An earlier version of this paper was presented at the 1987 Symposium on Logic Programming in San Francisco.

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

1. PROBLEM

Edinburgh Prolog is a small and elegant language, providing few but general constructs. For example, the only data are general terms: atoms and structures (with a little sugar to make certain binary structures look like lists).

The disadvantage is that although this is sufficient to represent any other data structure, all operations on these data structures may not be very efficient. In particular, modifying large data structures is very expensive. One remedy for this, which is the approach taken in some LISP systems (e.g., Common Lisp [Steele 84]) is to add new primitive data structures, such as arrays, strings, and hash tables, and built-in operations on them.

Unfortunately, doing this with Edinburgh Prolog would destroy the simplicity and uniformity of the language. Indeed, "rich" languages such as Common Lisp and Ada have been accused of becoming too large and incomprehensible.

The paper attempts to solve this problem. We will propose a number of data structures, primitive operations on them (which could not have been efficiently coded in Prolog) and various ways of integrating them in Edinburgh Prolog without destroying its uniformity.

Tricia is an implementation of Prolog on the DEC-2060 computer, developed at Uppsala University. It is based on a modified WAM [Carlsson 86; Warren 83] and the built-in predicates are roughly a superset of Edinburgh Prolog. Some of the constructs in this paper are implemented in version 0.63 of Tricia [Barklund *et al.* 86], all of them are intended to be implemented in version 1.0.

By "Edinburgh Prolog" we mean the language and semantics of Prolog described in [Bowen 81], not to the actual implementation developed at Edinburgh University. Below we will call this simply "Prolog".

2. HAIRY DATA STRUCTURES

In Prolog there are only two kinds of data: *atoms* (symbols) and *structures*. Structures are usually implemented as sequences of memory words where the first word contains a reference to the functor of the term and the following words contain the arguments of the structure. The only exceptions are structures with the functor `'.'/2` (lists) whose functors are implicit. This format allows direct access to arguments of the structure and makes it cheap to access the arguments sequentially, e.g., when unifying terms. It does not allow cheap non-destructive modification of arguments which requires copying the whole structure. However, we can imagine other ways of implementing some structures for the benefit of other operations.

In [Eriksson & Rayner 84] and [Barklund & Millroth 87] implementation methods are given for adding arrays and hash tables to Prolog. The key idea is a way to hide the side effects which are necessary for efficient updating of structures. This is accomplished by constructing *exception chains* for older versions of the data structure. One consequence of this is that the internal representation of certain arrays and hash tables can be quite complicated. In the rest of this paper, *ordinary structures* stands for structures implemented the normal way as described in the previous paragraph and *hairy structures* stands for other non-straight-forward implementations of structures.

3. TERMS IN PROLOG

In Prolog, all terms are expected to

1. have a syntax (both for input and output) on the form *atom* or $functor(arg_1, \ldots, arg_k)$ (although some sugared syntax may also exist),
2. be composable and decomposable by the built-in predicates `=../2`, `functor/3` and `arg/3` (referred to below as the *structure predicates*) and
3. unify with terms being syntactically equivalent or which can be made syntactically equivalent by instantiating variables in either or both terms.

If hairy structures are added to Prolog, there are at least three ways to do it, as described below.

3.1. Second-class Terms

Let hairy structures be "second-class" terms without syntax which can only be manipulated through built-in predicates. This is easiest to implement but has several disadvantages.

Firstly, the uniformity and simplicity of Prolog is broken. This is not only a philosophical point but it also complicates understanding of the language for novice programmers.

Secondly, imagine as an example a predicate which tests if a term is ground. This can be easily done having base cases for variables and atoms and a recursive case for structures using `=../2`. With second-class terms it would be necessary to add one case for each hairy structure.

Thirdly, when debugging programs it helps if arrays, hash tables etc. can be in- and output.

3.2. Abstract Data Structures

It has been suggested that using types and abstract data could be a way to handle the situation. The idea behind abstract data structures [Allen 78] is to provide (functions or relations acting as) recognizers, selectors, constructors and predicates on a data structure without revealing its actual representation. This is a beautiful idea but it stands in conflict with the principle of Prolog that all terms have a syntax.

After a major restructuring of the way terms are supposed to behave, it is possible that abstract data structures can fit nicely in Prolog.

3.3. Conservative extension

The third way is to require that hairy data structures meet all requirements for first-class terms in Prolog. The rest of the paper discusses how this can be done and at which cost.

4. LIMITED REPRESENTATION OF TERMS

Some hairy structures are only limited representations of terms in the sense that the implementation requires certain components of the terms to be instantiated or even ground. For example, it is possible to imagine strings where some characters are replaced by uninstantiated variables. Such strings cannot be implemented as packed arrays of characters, which is the common technique.

Hairy structures are either created by specialized built-in predicates or by structure predicates. When a hairy structure is created by a built-in predicate the problem can be handled by documenting that certain arguments of the predicate must be ground.

It is necessary to decide for structure predicates what should happen if an attempt is made to construct a hairy structure where some critical component is uninstantiated and what should happen if the component later becomes instantiated. Four possibilities follow below.

4.1. Construction Fails

The easiest solution is to let such attempts fail or generate an exception. This is easy to implement but makes Prolog even more incomplete. For experienced programmers this may in practice be only a small inconvenience.

4.2. Construction Suspends

In an implementation of Prolog with primitives for delaying goals until certain terms become instantiated or ground, the structure predicate could suspend until the critical component becomes instantiated. If this approach is taken, there are other built-in predicates which should have the same behaviour and those modifications to Prolog are beyond the scope of this paper.

4.3. An Ordinary Structure is Created

The previous approaches share the advantage that terms implemented as ordinary structures and terms implemented as hairy structures form mutually disjunct sets. If it is acceptable that two equivalent terms may have different internal representations, then a structure predicate can create an ordinary structure when the requirements for a hairy structure are not fulfilled. The next chapter discusses what problems will arise from this.

4.4. Always Create Ordinary Structures

Again assuming that different representations of equivalent terms are acceptable, structure predicates can always create ordinary structures, even when it would have been possible to create a hairy structure instead. This simplifies the structure predicates but of course affects performance when the created structure is used.

5. DIFFERENT REPRESENTATIONS OF EQUAL STRUCTURES

What happens if there are different representations of equal structures? Let us first state in which parts of a Prolog system terms are actually composed or decomposed.

1. The general unification routine.
2. Open-coded unification in compiled code.
3. Structure predicates.
4. Built-in predicates on certain terms.
5. Interning of atoms.

We will cover each of these separately.

5.1. General Unification

The general unification routine must be augmented with three new cases for each hairy structure type (one case for unifying two hairy structures and two cases for unifying a hairy structure with an ordinary structure).

For example, if an attempt is made to unify a string and an ordinary structure, the ordinary structure should be inspected to find out if it could possibly be an alternative representation of a string. If this is the case, both the string and the ordinary structure must be traversed to unify their components. An easy but not so efficient solution in this situation is of course to convert the string to its corresponding ordinary structure and then call the general unification routine again with the two ordinary structures.

5.2. Open-coded Unification

Every sensible representation of a hairy structure as an ordinary structure will reserve either a finite number of functors or all functors with a certain name (but varying arities). When the compiler finds a structure with such a functor in the head of a clause it cannot generate a `get_structure` instruction (we assume the implementation is based on some abstract machine similar to WAM). This is because at runtime the predicate may be called with a hairy structure as argument. We propose four alternative ways of extending WAM to cope with hairy structures.

5.2.1. Failure. All attempts to compose or decompose a hairy structure or its corresponding ordinary structure fail or generate an exception. This is in accordance with the behavior of structure predicates described in section 4.1.

5.2.2. Copying Hairy Structures. For each hairy structure *foo* a new instruction `get_foo Xi` is introduced. At runtime it dereferences `Xi` and takes one of the following actions:

1. If `Xi` dereferences to a variable then `get_foo Xi` is equivalent to `get_structure F,Xi` where `F` is the appropriate reserved functor, entering write mode. This will always create an ordinary structure, in analogy with the behavior of structure predicates described in section 4.4.
2. If `Xi` dereferences to an ordinary structure with a functor `F` which is one of the reserved functors then `get_foo Xi` should be equivalent to `get_structure F,Xi`, setting up the `S` register and entering read mode.
3. If `Xi` dereferences to a hairy *foo*-structure then a corresponding ordinary structure is created on the heap, the `S`-register is set to point to its first argument and read-mode is entered.
4. If `Xi` dereferences to something else backtracking is initiated.

The copying of structures in case 3 is admittedly not very elegant but it has the advantage that the compiler can generate `unify`-instructions following the `get_foo`-instruction just as after a `get_structure`-instruction.

5.2.3. Loading Arguments in Variables. For each hairy type of structure *foo* whose corresponding ordinary structure has a functor `F` and `k` arguments, an instruction `get_foo Xi,Xj1,...,Xjk` is introduced. If `Xi` dereferences to a variable or an ordinary structure, this instruction should be equivalent to the following sequence of WAM-instructions

```
get_structure F,Xi
unify_variable Xj1
...
unify_variable Xjk.
```

If `Xi` dereferences to a hairy `foo`-structure this instruction should be equivalent to creating a corresponding ordinary structure on the heap, placing a pointer to this structure in `Xi` and then executing the sequence of instructions above. This has the advantage that for many hairy structures the corresponding ordinary structure does not have to be created. The disadvantage is that it may be impractical to implement instructions with a large number of arguments using a byte-code emulator. Also, this may be a waste of temporary registers.

5.2.4. Do Not Open-code Unification. The simplest way to handle open-coded unification is not open-coding unification of a source term which may unify with a hairy structure. For example, if in the ith argument position of some clause appears a term `bar(a,Xj,c)` which may unify with a hairy structure, it is not compiled to

```
get_structure bar/3,Xi
unify_constant a
unify_value Xj          % or unify_variable Xj if first occurrence of Xj
unify_constant b
```

but rather to

```
put_structure bar/3,Xi
unify_constant a
unify_value Xj
unify_constant c
get_value Xi,Xj
```

thus creating an ordinary term and relying on the general unification routine to handle the case when `Xi` is a hairy structure. In good programs, head unification of hairy structures will be very rare so we do not think this will affect space or time performance significantly.

5.3. Structure Predicates

In chapter 4 above we discuss different approaches to how structure predicates should compose possibly hairy structures. Let us now see how hairy structures can be decomposed.

The predicate `functor/3` is probably quite trivial to implement since it only involves finding the functor of the corresponding ordinary structure. `arg/3` and in particular `=../2` may be more difficult to implement.

For some hairy structures (e.g., arrays which are similar to ordinary structures) accessing components by `arg/3` will be almost as easy as for ordinary structures. In general, however, `arg/3` will be as difficult as `=../2`.

The simplest general solution is of course to create a corresponding ordinary structure and then decompose it. On the other hand, we think that for most hairy structures it is not difficult to write primitive routines for handling them. `arg/3` and `=../2` would then only have to dispatch to the correct routine.

5.4. Built-in Predicates

For every primitive built-in predicate operating on some hairy structure (e.g., modifying a hash table) it is necessary to define an equivalent operation on the corresponding ordinary structures. If corresponding ordinary structures are reasonably well chosen it will be easy to define these operations in Prolog with bad but acceptable efficiency. For an example of how this is done for hairy hash tables, see [Barklund & Millroth 87].

5.5. Interning of Atoms

If some hairy structures correspond to atoms (such as `empty_ht` below in section 6.1.3) they may have to be recognised when atoms are interned. In this case it is obvious that predicates on atoms must recognise them and treat them as atoms. Alternatively the built-in predicates on hairy structures (not only the Prolog equivalents of the predicates mentioned in the previous section) should handle these atoms. The latter approach is probably easiest to implement.

6. FOUR EXAMPLES OF HAIRY STRUCTURES

Logical terms as provided in Prolog are a very elegant data type which quite easily can be used to represent any other data type. Programs operating on small data structures usually have no problem representing their data efficiently, simple lists and trees work fine in most cases. When programs need large data structures, however, the programmer often finds that general logical terms are too inefficient for the particular problem. The key point is that it is easy in Prolog to say that two terms `T1` and `T2` have one or a few components in common, this can be done by, e.g.,

```
arg(I,T1,X), arg(I,T2,X),
```

but it is very difficult to say that two terms have all elements except one or a few in common. In imperative languages this is solved by destructively changing one term to obtain the other, causing a side effect of the computation. The most important hairy structures are proposed to overcome this problem.

6.1. Tables

We define a table to be a term which pairs together ground keys with values. (In Prolog a table can alternatively be represented by a set of binary clauses. Updating such tables requires manipulating the data base and we do not consider them here.)

6.1.1. Motivation. Tables are very common in Prolog programs. Small tables are often implemented as A-lists, larger tables as trees. If accessing a table is much more frequent than adding to or deleting from it, it is realistic to implement the table efficiently in Edinburgh Prolog as a k-order tree structure. Since destructive updating of the tree is impossible, insertion or deletion of an element in a tree with n nodes requires allocation of between $\log_k(n)$ and n nodes, depending on how well the tree is balanced. This becomes expensive when the tables get large, as they often do.

Sometimes a Prolog coding trick is employed: representing the table by a tree whose fringe consists of uninstantiated variables, to facilitate the addition (but not replacement or deletion) of nodes. This is elegant for certain purposes [Warren 80], but is generally awkward and programs using such tables often need to use meta-logical concepts.

Hashing is another efficient implementation method for tables. If hash tables are represented by ordinary Prolog structures, programs for accessing values by keys and adding or removing key-value pairs can be written in Prolog. Unfortunately, if addition and deletion of key-value pairs is to be implemented efficiently, destructive modification of the terms becomes necessary. If the implementation of hash tables is made below the Prolog user level, it is possible to let insertion and deletion invisibly use destructive operations, hiding their side effects. From the outside the operations will appear to be completely free of side effects. In this way it is possible to let most insertions and deletions allocate an amount of memory equivalent to only one tree node.

We consider an efficient and clean implementation of tables in Prolog so important that it motivates extending Prolog with hairy structures.

6.1.2. Implementation. See [Barklund & Millroth 87] for an implementation method for *hash tables* in Prolog.

6.1.3. Integration. Finding a suitable correspondence between hash tables and ordinary structures is difficult, since the components of a hash table have no meaningful observable order even in the actual hairy structure. Also the components must be pairs of keys and values where the keys are ground.

One approach is letting a hash table containing n pairs of keys and values correspond to a list

$$[k_1\text{-}v_1,k_2\text{-}v_2,\ldots,k_n\text{-}v_n],$$

where for every i, k_i is ground and k_i`@<`k_{i+1} (`@</2` is some total ordering on ground terms). The ordering criterion is necessary to give a unique syntax for a hash table. The syntax for hash tables is not likely to be used very often so a clumsy syntax may be acceptable. Incidentally, to obtain a hash table without worrying about the order of the keys, a goal

$$\texttt{keysort}([k_1\text{-}v_1,\ldots,k_n\text{-}v_n],\texttt{T})$$

could be used.

The problem with this correspondence (and indeed all correspondences between hairy structures and lists) is that lists are used in very many places in a Prolog system. It would be difficult and expensive to allow hash tables or other hairy structures to appear in every place where a list is expected. It would also be very expensive to create a list, checking if the list could actually be implemented as a hairy structure. In other words, the functor `'.'/2` is already reserved for other purposes.

(This can be seen as an argument for structure-based Prolog against list-based Prolog, where the only non-atomic first-class terms are lists. Integrating hairy structures in such a system would give very many internal representations of lists and this would be very expensive to maintain.)

Viewing a hash table such as the above as a structure

$$\texttt{ht(}k_1\texttt{-}v_1\texttt{,}k_2\texttt{-}v_2\texttt{,}\ldots\texttt{,}k_n\texttt{-}v_n\texttt{)},$$

(which could be created by the goal

$$\texttt{keysort([}k_1\texttt{-}v_1\texttt{,}\ldots\texttt{,}k_n\texttt{-}v_n\texttt{],S), T=..[ht|S])}$$

is better but our reason for not choosing this view is that predicates for accessing, adding, or removing elements in hash tables are much easier to express if hash tables are viewed as recursive structures.

We have therefore chosen the view expressed in [Barklund & Millroth 87]: a hash table containing n pairs of keys and values corresponds to the structure

$$\texttt{ht(}k_1\texttt{,}v_1\texttt{,ht(}k_2\texttt{,}v_2\texttt{,}\ldots\texttt{ht(}k_n\texttt{,}v_n\texttt{,empty\_ht)}\ldots\texttt{))}$$

where for every i, k_i is ground and k_i`@<`k_{i+1}. The necessity of ordering is the same as above. Most built-in predicates on hash tables can be expressed quite easily in Prolog using this view [Barklund & Millroth 87].

The greatest disadvantage with this view is that it may encourage people to write programs recursing over a hash table using clauses such as

```
size(ht(K,V,T),M,O) :- N is M+1, size(T,N,O)
```

It is possible to obtain the "rest" `T` of the hash table in the first argument using the primitives for removing one element. In this case, recursing over the whole table may become expensive. This is one reason why we suggest in chapter 5 above that when a hairy structure is decomposed, the components should become ordinary structures. In the example above, at the first level of recursion, `T` would get instantiated to the ordinary structure corresponding to the rest of the table. Further recursion would only have to decompose this ordinary structure.

6.1.4. Built-in Predicates. The following are some predicates on tables which should be implemented at a low level to be really efficient, but with corresponding versions for ordinary structures. A + sign before an argument means that the argument must be instantiated. Other arguments may be instantiated or uninstantiated.

`gethash` *key +hash_table value*
: This is true if *key* is associated with *value* in *hash_table*. If *key* is ground this goal succeeds zero or one time. If *key* is not ground it succeeds one time for each key in *hash_table* which unifies with *key*. The value returned for *key* is not a copy of the value found in the table (as it would have been if the table was implemented using the internal data base). To implement Prolog data bases using these hash tables, it is therefore necessary to have separate primitives for freezing and melting clauses [Nakashima *et al.* 84]. (The internal data base in Tricia is indeed implemented in this way [Barklund 87].)

`puthash` *+key +hash_table value new_hash_table*
: *new_hash_table* is exactly like *hash_table*, except that *key* is associated with *value*. This does not change *hash_table*.

`remhash` *key +hash_table new_hash_table*
: *new_hash_table* is exactly like *hash_table*, except that *key* is not associated with any value. This does not change *hash_table*. This may succeed zero, one or several times, in analogy with `gethash`.

`size` *+hash_table size*
: The integer *size* is the number of key-value pairs in *hash_table*.

For efficiency of some applications, predicates for testing if some table is actually implemented as a hairy structure and for converting between hairy and ordinary structures could be provided.

(In [Barklund & Millroth 87] two predicates `addhash/4` and `modhash/5` are given. They can easily be defined in terms of `puthash/4` and `remhash/3` or vice versa.)

6.2. Arrays

Arrays are a special case of tables, where the keys are always integers in a limited range, but they are used in a quite different manner. Because of their fixed set of keys, integrating them in Prolog should be done in a different way.

6.2.1. Motivation. Many computations, numerical as well as symbolic, use fixed-size tuples, i.e., arrays. In Prolog, structures are used for this purpose. This works fine except for large tuples. Changing a few elements in a large structure is expensive since the whole structure must be copied. Introducing mutable arrays is an attempt to solve this problem. Since arrays are a special case of tables they could easily be implemented as hash tables. This would at the same time remove the condition that the range of the keys for an array must be predetermined. Implementing arrays this way is quite efficient since a trivial hash function can be used for integer keys and it is often possible to dimension the physical hash table to avoid hash collisions. In spite of this, we have chosen to implement arrays separately, because there are several operations on arrays which are meaningless on general hash tables. By making arrays a separate data type, the domain of those operations is clearly defined.

6.2.2. Implementation. In [Eriksson & Rayner 84] an implementation of "mutable arrays" is described (which was the primary source of inspiration for the implementation of hash tables in Tricia).

6.2.3. Integration. Mutable arrays have much in common with ordinary structures, the main difference being that updating an element is usually much cheaper and accessing an element is usually a little more expensive for arrays.

Recursing over the elements of a structure is an unnatural operation and is not provided in Prolog, instead `=../2` is used to obtain a recursive term. We think that the corresponding ordinary structures for arrays should also be non-recursive so we let an array of length k correspond to a structure with functor `array/`k. As for structures, the implementation of arrays does not require the elements to be instantiated so there are no particular difficulties in integrating arrays this way. Most operations on arrays are obtained using the general structure predicates.

Array elements are accessed with the predicate `arg/3`.

Arrays are created either by calls to `functor/3` (a goal `functor(A,array,N)` instantiates `A` to an array containing `N` uninstantiated variables), or with `=../3` to create them with some initial contents (`A =.. [array,a,b,c,d]` instantiates `A` to an array of length 4, containing the atoms `a`, `b`, `c` and `d`). `functor/3` can also be used to get the size of an array.

The normal way to recurse over an array `A` viewed this way would be to decompose it with `=../2`, i.e., `A=..[array|L]` and recurse on `L`.

The remaining array operation, i.e., updating elements, requires a new built-in predicate since there is no corresponding structure predicate.

6.2.4. Built-in predicates. The most important built-in predicate on arrays is

`argh` *+Index +Array Element New_array*

This is true if *Index* is a valid index in *Array* and *New_array* is exactly as *Array* except that the argument specified by *Index* is *Element*. A specification (not runnable in Prolog) of `argh/4` could be

$$\begin{aligned}&\forall I,A,X,B\{argh(I,A,X,B)\leftrightarrow\\&\qquad\exists K\{arg(I,B,X)\wedge functor(A,array,K)\wedge functor(B,array,K)\\&\qquad\quad\wedge\forall J\{1\leq J\wedge J\leq K\wedge I\neq J\rightarrow\exists Y\{arg(J,A,Y)\wedge arg(J,B,Y)\}\}\}\}.\end{aligned}$$

Alternatively, we could imagine a predicate `gra/3` which is the complement of `arg/3` i.e., `gra(I,A,B)` is true if the structures `A` and `B` are identical except for their `I`th argument (which may or may not be identical). Using `gra/3` it is easy to define `argh/4` as

```
argh(I,A,X,B) :- gra(I,A,B), arg(I,B,X).
```

6.3. Characters

Characters are the minimal units for communication with text I/O devices. Also, text strings are composed of characters.

6.3.1. Motivation. In Edinburgh Prolog a character is represented by an integer whose value happens to coincide with the ASCII-code of the character. Sometimes this may be desirable:

1. Some operations, e.g., case conversion, can easily be defined in terms of standard arithmetic operations.
2. The set of constants in Prolog is kept down.

On the other hand there are some reasons for introducing characters among Prolog's terms:

1. The common operations on characters are few and can be handled by a small set of primitives. If the ASCII code of a character can be easily obtained, any other operation can be coded almost as easily as if the characters were represented directly as integers.
2. If predicates on characters are introduced, a character data type naturally delimits the domain of such predicates and also delimits what can be put in a string.
3. It is practical to have a syntax for character constants in Prolog. In some Prologs this is solved by introducing a notation hack where, e.g., `0'X` (radix zero) denotes the ASCII code of `X`.
4. The output routines do not know when an integer should be written as a character constant or a sequence of digits.
5. On modern I/O devices, characters may have other properties than their ASCII codes, e.g., a font. Such attributes are difficult to add to characters represented as integers.

We think the latter arguments take precedence over the former and propose a character data type in Prolog, similar to that of Common Lisp.

6.3.2. Implementation. A character has three disjoint parts: a *character code*, a *font* and *shift bits*. In Tricia the code may occupy 8 bits and the font 6 bits, but there is no reason why this could not be increased to care for larger character sets. There are 4 shift bits which are mainly intended for communication with keyboards (named *hyper*, *super*, *meta* and *control*) [Steele 84].

Any practical implementation of characters as hairy structures will require them to be ground.

6.3.3. Integration. Unlike hash tables and arrays, characters always have a certain number of components. This makes it easy to integrate them in Prolog. We propose that a character should have a corresponding ordinary structure `char(`*code*`,`*font*`,`*bits*`)`, where all arguments are integers in suitable ranges.

6.3.4. Built-in Predicates. All predicates below are easy to implement in Prolog, but some computers provide efficient primitives for them, it is desirable that programmers do not introduce different versions of them and it is desirable to enhance portability.

`alphabetic` +*Character*
: True if *Character* is an alphabetic character.

`digit` +*Character*
: True if *Character* is a digit.

`upcase` +*Character U_case*
: True if *Character* is alphabetic and *U_case* is the same character, in upper case.

`downcase` +*Character L_case*
: True if *Character* is alphabetic and *L_case* is the same character, in lower case.

For convenience, it may also be desirable to introduce a syntactic sugar for characters with null font and bits.

6.4. Strings

Strings are text terms. Most computers provide efficient representations of ground strings.

6.4.1. Motivation. In Edinburgh Prolog atoms work as text constants. There is also a convention that a list of integers sometimes may be interpreted as a string where each integer corresponds to a character in the string. Most of the arguments above for and against characters as a data type can be applied to strings as well and we introduce strings as a separate data type for the same reasons as for characters.

6.4.2. Implementation. Strings can be implemented on top of arrays if the implementation of arrays can be optimised for byte arrays. Otherwise, it is easy to make separate implementation of mutable strings similar to that of arrays. The syntax "$C_1C_2\ldots C_n$" can be modified to denote such a (ground) string rather than a list of integers. For ground strings, a packed representation is the most efficient but if the implementation is made on top of arrays it should be possible to resort to using the general arrays, so the components need not be ground.

6.4.3. Integration. There are several ways to view a string: as an "uninterned" atom (requiring strings to be constant), as a list of characters or as an array of characters. We are somewhat ambivalent towards the view of strings. We could require them to be ground, so that viewing them as constants would work. On the other hand, even with ground strings, one does often want to inspect the constituent characters or recurse over the string. Therefore strings should correspond to non-atomic terms.

Let us investigate the pros and cons of some possible integrations of strings in Prolog.

1. A string could be viewed as a list of integers (or characters, but that is not important here). The advantage would be one of backwards compatibility and compact syntax but the disadvantages of using lists for this purpose are discussed in section 6.1.3 and we dismiss this solution.
2. A string of length k could be viewed as a structure with the functor `string/`k and character arguments, similar to arrays. This stresses the non-recursiveness of the actual implementation of strings. As for ordinary structures and arrays, it forces users to create recursive terms explicitly if they want to recurse over them.
3. A hybrid solution is viewing a string as a structure with functor `string/1` whose argument is a list of characters. This has the advantage of reserving only one functor but may require structure predicates etc., to inspect not only the functor but also the first argument to determine if a hairy structure can be created.
4. A more interesting solution is to view a string as a binary structure with functor `string/2`, whose components are the first element of the string and the rest of the string. An empty string would be a term `empty_string`. Like the first approach, this is a recursive structure, but this approach is better since Prolog has less prejudice towards the functor `string/2` than towards `'.'/2`. The disadvantage is that the efficient implementation of strings is not recursive. If an attempt is made to destruct a string, the best solution is probably to return the first element of the string and an ordinary structure corresponding to the rest of the string.

In Tricia we have chosen the second approach, but the fourth approach also has certain advantages. Using the second approach, strings are accessed (`arg/3`), created (`functor/3` and `=../2`), and modified (`argh/4`) in the same way as arrays.

6.4.4. Built-in Predicates. Some other predicates on strings should be built-in to take advantage of the efficient representation and string handling primitives available on many computers. For example:

`upcase` *+String U_string*
: *U_string* is *String* in upper case. This and the next relations are supposed to hold between strings or between characters (see section 6.3.4).

`downcase` *+String L_string*
: *L_string* is *String* in lower case.

`capitalize` *+String C_string*
: *C_string* is *String* with its first letter in upper case and the following letters in lower case.

`substring` *+String +Start +End Substring*
`substring` *+String Start End +Substring*
: *Substring* is a segment of *String*, beginning at subscript *Start* and ending at subscript *End*. The second mode is a string search.

7. EFFICIENCY

Implementing the changes in a Prolog system as suggested, will not have negative consequences on performance if hairy structures are not used, except if structure predicates try to create hairy structures when possible. In this case, creating a structure will involve looking up the functor in a table to see if it is reserved.

This is already implemented in most Prolog systems, to ensure that structures with the reserved functor `'.'/2` are always implemented as lists.

The consequences will also be small if hairy structures are only used through the built-in predicates. For example, if hash tables are only created empty and changed through the predicates `puthash/4` and `remhash/3` then performance will not be affected.

The suggested modifications only have effect if hairy structures are used in other ways than they are intended for. For example, recursing over a hash table is an expensive operation, but everything will work as expected, the only penalty is in execution time and possibly memory consumption. This is just an instance of a well known principle: no data structure is efficient for all kinds of operations.

8. ALTERNATIVES

The following are some other ways which have been proposed to solve some or all of the problems mentioned in the introduction.

8.1. Why Not Let All Structures Be Mutable?

It would be possible to make any structure mutable using the technique described in [Eriksson & Rayner 84]. This would be elegant since then predicates such as `gra/3` or `argh/4` (see section 6.2.4) could be applied to any structure and there would be no need to make arrays a special data structure.

The disadvantage of doing this is that it becomes necessary to modify all parts of the Prolog system which assume anything about the internal representation of *any* structure. The most serious effect of this is for open-coded unification in WAM which would get several times more complicated by taking into account the possibility that a hairy structure may appear anywhere. In our solution above we avoid this by localizing such effects to structures with certain reserved functors.

8.2. Cdr-coded Lists

Cdr-coded lists [Baker 78] have been proposed as an alternative to arrays and are implemented in some Prolog systems (e.g., PLM [Dobry *et al.* 85]). These are lists where one or two bits per word are used to store information about how the tail of the list can be found. Many lists with k elements can then be stored in k words of memory instead of $2 \times k$ words for normal implementation. This does not solve the problem of how to update components of non-atomic terms efficiently. It saves memory but may well increase the execution time of programs since traversing lists will involve more computation, although the number of memory references is decreased.

8.3. Mappings

We are investigating another quite different way of integrating arrays and tables in Prolog. The idea is to replace structures as the only non-atomic data structure by a more general kind of term: *mappings*, of which structures are a special case.

A mapping is a functor together with a table. A mapping with a table whose set of keys is $\{1, \ldots, n\}$ is a structure. The generalisation of `functor/3` is `domain/3` whose arguments are a mapping, its functor and its set of keys. `=../2` is generalized as `compose/4` taking as arguments a mapping, its functor, its set of keys and its set of values. `apply/3` corresponds immediately to `arg/3`, but the first argument could be any key, not just integers. Two mappings unify if they have the same functor and set of keys and the values corresponding to each key unify.

General mappings would be implemented by hash tables but the special case by ordinary Prolog structures or arrays. A program not using the general mappings could be run in ordinary Prolog.

Connection Machine® Lisp [Steele & Hillis 86] has a similar device, called *xappings*. We will investigate this subject further.

9. RELATED WORK

Several authors (e.g., [Eriksson & Rayner 84] and [Cohen 84]) have studied how arrays could be incorporated in Logic Programming but little attention has been paid to how they should be made first class terms in Prolog, although some authors may have had a solution like ours in mind.

Strings are implemented in some Prolog systems but as far as we know, they are either assumed to be constant (i.e., atoms) or implemented inefficiently.

[Pereira 85] introduces directed acyclic graphs—a data structure related to hash tables—in logic programming, but does not integrate them among ordinary terms. We are not aware of any other work on hash tables in Logic Programming, even less so on how they should fit into an otherwise austere language like Edinburgh Prolog.

10. CONCLUSIONS

In [Barklund & Millroth 87] an implementation method was given for hash tables in Prolog. An interpretation of them as first class terms was given for the purpose of showing that they belong in a logic programming language. We have investigated and suggested solutions to the problems appearing when arrays, hash tables, and other data structures which have proved useful in Computer Science are to be incorporated as first class terms of Prolog. This is to make unification, composition and decomposition of these structures work exactly as for ordinary structures with the straight-forward implementation.

We think mutable data structures are so important that they should be considered for inclusion in any serious Prolog system. Making them first-class structures is non-trivial, but the methods suggested in this paper preserve both the efficiency and the simplicity of terms in Prolog.

It may not be worth the trouble to let structure predicates and interning of atoms try to create hairy structures when possible. This simplifies structure predicates since they will only have to worry about decomposition of hairy structures. As hairy structures are not normally intended to be created by structure predicates, this does not harm performance significantly. Also, since it is difficult to let open-coded unification in compiled code construct hairy structures, this make the behaviour of structure predicates and compiled code consistent (although this is not visible to users).

It sems that hairy structures which are implemented recursively or without any useful order between its components, should in general have recursive corresponding ordinary structures. On the other hand, hairy structures of a fixed size, or with a certain order between the components, should have non-recursive corresponding ordinary structures. Examples of the former are lists and hash tables, examples of the latter are characters, arrays, and strings. In some cases, such as strings, recursive ordinary structures would be desirable, in such cases it may be worthwhile to search alternative recursive implementations of the hairy structures.

ACKNOWLEDGEMENTS

We are grateful to our colleagues at UPMAIL for interesting discussions and nice tea breaks, particularly to Åke Hansson whose criticism did much to improve this paper. The comments from the referees of SLP '87 were also valuable. We would also like to thank Martin Nilsson at the University of Tokyo for giving us access to computer equipment for the final editing of the SLP '87 version of the paper, and Yoshihiko Futamura at Hitachi Advanced Research Laboratory for his hospitality. This work would not have been possible without the continous support from our families.

REFERENCES

[Allen 78] J. Allen, *Anatomy of LISP*, McGraw-Hill, New York, 1978.

[Baker 78] H. G. Baker Jr., *List Processing in Real Time on a Serial Computer*, Communications of the ACM, vol. 21, no. 4, pp. 280–293, April 1978.

[Barklund 87] J. Barklund, *Efficient Interpretation of Prolog Programs*, ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, St. Paul, Minn., June 1987.

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

[Barklund & Millroth 87]
J. Barklund, H. Millroth, *Hash Tables in Logic Programming*, 4th International Conference on Logic Programming, Melbourne, May 1987.

[Bowen 81] D. L. Bowen, *DECsystem-10 Prolog User's Manual*, University of Edinburgh, Dept. of Artificial Intelligence, Edinburgh, 1981.

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

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

[Dobry *et al.* 85] T. P. Dobry, A. M. Despain, Y. N. Patt, *Performance Studies of a Prolog Machine Architecture*, University of California, Berkeley, 1985.

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

[Nakashima *et al.* 84] H. Nakashima, K. Ueda, S. Tomura, *What Is a Variable in Prolog?*, International Conference on Fifth Generation Computer Systems 1984, pp. 327–332, Tokyo, 1984.

[Pereira 85] F. C. N. Pereira, *A Structure-Sharing Representation for Unification-Based Grammar Formalisms*, 23rd Annual Meeting of the Association for Computational Linguistics, pp. 137–144, Chicago, July 1985.

[Steele 84] G. L. Steele Jr., *Common Lisp, the Language*, Digital Press, 1984.

[Steele & Hillis 86] G.L. Steele Jr., W.D. Hillis, *Connection Machine® Lisp: Fine-Grained Parallel Symbolic Processing*, 1986 ACM Conference on Lisp and Functional Programming, pp. 279–297, Cambridge, Mass., Aug. 1986.

[Warren 80] D. H. D. Warren, *Logic Programming and Compiler Writing*, Software—Practice and Experience, vol. 10, no. 2, pp. 97–125, Feb. 1980.

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