

Supporting more types in the WAM: the hProlog tagging schema

Bart Demoen, Phuong-Lan Nguyen

Report CW 366, August 2003



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

Supporting more types in the WAM: the hProlog tagging schema

Bart Demoen, Phuong-Lan Nguyen

Report CW 366, August 2003

Department of Computer Science, K.U.Leuven

Abstract

Starting from dProlog, we developed hProlog to become a back end for HAL. We incorporated attributed variables, the data types *string* and *character* and support for *bigints*. We describe how we managed to cater for all these, while not further restricting the heap address space and without efficiency loss. We explain the rationale behind some decisions, the effect on indexing, some new low-level built-in predicates and abstract machine instructions. The tagging scheme has room for extending it to other basic types.

Supporting more types in the WAM: the hProlog tagging scheme

Bart Demoen^{*} Phuong-Lan Nguyen[†]

August 30, 2003

Abstract

Starting from dProlog, we developed hProlog to become a back end for HAL. We incorporated attributed variables, the data types *string* and *character* and support for *bigints*. We describe how we managed to cater for all these, while not further restricting the heap address space and without efficiency loss. We explain the rationale behind some decisions, the effect on indexing, some new low-level built-in predicates and abstract machine instructions. The tagging scheme has room for extending it to other basic types.

1 Introduction

We assume knowledge of Prolog and its WAM based implementation. For a good introduction to the WAM [17], see [1].

The WAM originally defined only the data types ATOM, LIST, STRUCT and REF. Since heap addresses are typically a multiple of four, the encoding tags a heap cell in its two lowest bits, e.g. 0x1 for ATOM, 0x2 for LIST, 0x3 for STRUCT and 0x0 for REF. The heap address range is not restricted by this. However, Prolog systems typically also need to support integers (INT) and floating point numbers (FLOAT). Often such support is given without reducing the heap address range, but it typically restricts the range of integers (from 32 bits to 29 in dProlog) and

^{*}Dept. of Computer Science, K.U. Leuven, Belgium, bmd@cs.kuleuven.ac.be

[†]Inst. de Mathématiques Appliquées, UCO, Angers, France, nguyen@ima.uco.fr

even the precision of floating point numbers. The latter is rather repulsive and dProlog [7] choose to half the available heap addresses in favor of supporting the C *double*. That was the situation end 2000. At that moment, it was decided to let dProlog evolve into a back end for HAL [6] and rename it to hProlog. Chronologically, we abandoned the use of the compiler based on the XSB compiler [15] in favor of the compiler written by Henk Vandecasteele for ilProlog (see for instance [2]) end 2000; we incorporated *delayed variables* in hProlog (beginning of 2001), replaced them by *attributed variables* (See [10, 5] - mid 2002), implemented the types *character* and *string* natively (end 2002), and started implementing *bigints* (Febr 2003 ¹). That means that compared to dProlog, the hProlog implementation has support for four more native types, totaling now nine ² and we want this without further restricting the heap addresses, without performance loss, and without changing the deref operation:

```
#define deref(p) \
    while (! (p & 0x3) && (p != *p)) p = *p
```

We particularly did not want to change the deref operation, because it happens so often in the WAM and so it needs to stay as simple as it is.

Before continuing, let us explain briefly why we wanted to support natively the data types string and character in hProlog: the Prolog syntax for strings is in the form of double quotes and the implementation is often by a list of character codes. We wanted to do better than character code lists, because of the wastefulness of that representation and also because we do not wish to make the unification of [97, 98, 99] and "abc" succeed. The other - and main - reason was that we inherit code from Mercury, which uses strings where Prolog programs typically use atoms: for instance, in Mercury the atom type does not exist as such, but string is a built-in type. Mercury code consequently uses strings in a predicate that defines the operators needed for reading a term. Such a predicate needs indexing and with lists of character codes as the internal representation of strings, this is a bit awkward. A similar story goes for characters, although one cop-out for characters in a Prolog system would be to represent a character as an atom of length one.³ This would however confuse two separate types, which in a back end for a typed language, we cannot tolerate.

¹at the moment of writing, the implementation lacks long division, full integration with the garbage collector and source files cannot yet contain bigints

²ref, atom, list, struct, smallint, bigint, float, string, character, attributed variable

³GNU-Prolog would use such a solution - private communication with D. Diaz

In Section 2 we explain the hProlog tagging scheme, in Section 3 the interaction with the garbage collector, in Section 4 the additions to the built-ins and abstract machine instructions. Section 5 contains the experiments and Section 6 finished with a conclusion.

2 Three tag bits

We are developing hProlog almost exclusively on Intel machines. Pointers to the heap are aligned on a 4 byte boundary and the highest order bit of pointers returned by malloc, is 0. This means that we can use 3 bits for the tagging. We chose to use three lower bits and to shift the word when necessary. This resulted in the following tags in bits 1 to 3:

REFERENCE	000 and 100	(**)
STRING	001	(*)
SIMPLE	010	
NUMBER	011	(*)
ATT	101	(*)
STRUCT	110	(*)
LIST	111	(*)

(**) means that the contents of the word is to be interpreted as a pointer. As usual in the WAM, a self-reference means an undef - the end of a reference chain.

(*) means that the word should be (logically) shifted to the right, and masked with $\sim 0x3$, to obtain a pointer⁴ to a sequence of cells that contain more data. The case LIST is implemented as in traditional WAM: the pointer points to two consecutive cells on the heap and these are the head and tail of the list. In the other (*) cases, the (derived) pointer points to a *header* which also must have a tag because of the heap garbage collector: more on this in Section 3.

Before getting into more details, it is worth noting that hProlog keeps its atoms and functors in the same array: the array index is simply used for accessing the string value of atoms and functors, their arity, their associated global value, their entry point as a predicate and any debugging information associated to them. This is of course a bit wasteful for atoms (which always have arity 0) or atoms and functors that are not predicates, but the implementation effort was deemed not worth

⁴[4] describes an alternative to shifting: negation of the bits can be used for one of the pointers

the effort of minimizing the space. The fact of using an array is not essential: the index in the array could also have been chosen to represent an index in another (more flexible) data structure without restricting the number of atoms/functors that can be represented.

We describe first the tag SIMPLE, as it is the most complicated one and used by more than one basic type.

2.1 SIMPLE

The tag SIMPLE is overloaded and could mean that the rest of the word encodes an atom, a character, a small integer or is a struct header (i.e. a functor descriptor) or a string header. Atom, character and small integer are *atomic* in the Prolog sense of the built-in. Let a cell have the SIMPLE tag - remember it is only 3 bits.

- If the fourth bit is zero, the remaining 28 bits encode a small integer in the usual two's complement. This choice ensures that detecting whether some cell contains a small int is fast.
- If the fourth bit is one, bits 5 to 11 (7 bits) often contain an arity, so we name this field the *generalized arity* or *genarity* for short.
 - If the genarity equals 0x7f, the cell is an atom and the remaining 21 bits denote an index in the atom/functor table.
 - If the genarity equals 0x7e, the remaining 21 bits encode a character.
 - If the genarity equals 0x7d, the cell contains a STRING header, and the 21 bits indicate the length of the string.
 - In all other cases the cell contains a functor, i.e. the header of a structure on the heap. The 21 bits denote again an index in a functor table, where the name, the real arity and (if present) the code entry point of the functor can be found. If the genarity is different from zero, it represents the actual arity of the functor, otherwise the arity must be looked up in the atom/functor table. It means that functors with an arity smaller than 124 (0x7d) do not need this general look-up and that covers most occurrences of functors and it allows support for very large arities.⁵

⁵during an experiment with very large arities, we noticed that this distinction does not really pay off

Section 2.3 gives overview in the form of pictures. Note that general unification (and abstract machine instructions) does not need to distinguish between these cases: a header is only reachable by its tagged pointer and the case ATOM, SMALLINT and CHAR need no distinction.

2.2 NUMBER and STRING

A NUMBER tagged word is a pointer to the header of either a floating point number (a C double) or a bigint. As usual, the pointer is extracted from the word w by the code

```
pointer = (((unsigned long)w) >> 1) & ~0x3
```

The same is true for LIST, FUNCTOR and ATT tagged pointers.

The header of a bigint (BIGINT_HEADER) has the value 0 and the REAL_HEADER has the value 1. At first we tried to pack more info in the BIGINT_HEADER, but later refrained from doing so mainly for simplicity reasons.

The REAL_HEADER is followed by two cells: the double. Its precision and range is not affected. Loading and storing must take into account the ability of the processor to load and store non-aligned doubles.

The BIGINT_HEADER is followed by a cell that indicates the length (in words) of the bigint, its sign and a bit telling whether to find the value as the next cells or not. The latter is related to heap overflow. Execution that does not overflow the heap during a computation, always produces the value cells immediately after the information cell. We will come back to this point in Section 3.

A STRING tagged pointer points to a cell which is identified as a STRING_HEADER because it has a SIMPLE tag and a generarity that equals 0x7d. The remaining 21 bits indicate the length of the string in bytes: the length in words is easily derived from that. The characters are represented as C does: one byte per character. There is no trailing zero byte and in principle, a string can contain zero bytes. It would be easy to incorporate strings with wide characters in this scheme.

2.3 Summary of the tags and outlay on the heap

Figure 1 shows the atomic types with a SIMPLE tag and which are tagged-on-data, as well as the two kind of variables in hProlog: ordinary (Herbrand) variables and attributed variables. A reference has the same tag as a free variable.

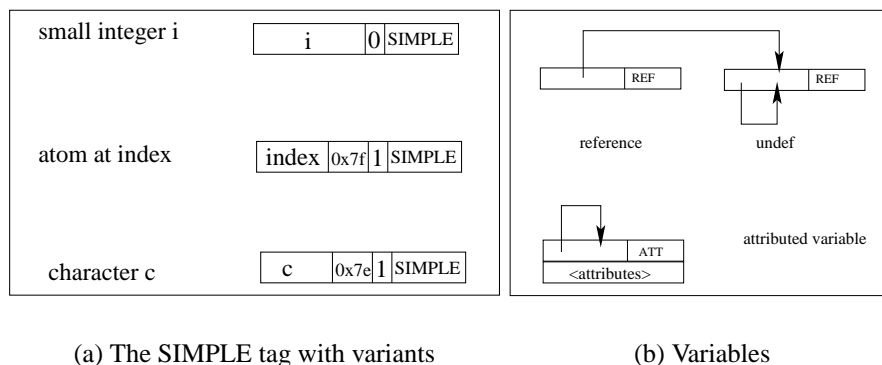


Figure 1: Simple tag and variables

Figure 2 shows the Prolog types which use tag on pointer: LIST, STRING, STRUCT and NUMBER. In the case of STRING, STRUCT and NUMBER, the pointer points to a header. The LIST pointer points directly to other Prolog terms.

3 What garbage collection has to do with it

Most garbage collection schemes for Prolog are based on the algorithm of Morris [14] or on the algorithm of Cheney [3]. Both require the ability to scan in a linear fashion the heap while recognizing the type of each cell. Morris' algorithm even needs this while scanning the heap backwards. One or more spare bits in each heap word or in the extra mark array can make this easier, but since in hProlog garbage collection is based on Cheney without marking (see [8]) and since there are no spare bits in a heap word, we must carefully choose the tags so that scanning the heap (from lower to higher addresses) is possible. For this reason, the header for bigints (0) and doubles (1) can not carry extra information besides the tag: indeed, 0 is also the tag for a reference, and 1 for a string pointer, but 0 is not a legal pointer, so cells with 0 or 1 are unambiguously identifiable as number headers. Also, because of the SIMPLE tag, cells that the scan must skip can be

recognized easily - also the cells containing a functor. The need for such tagging of a functor cell is also discussed in [1].

A second point related to garbage collection is related to bigints: the production of a bigint can be in the middle of a chunk of code in which it is - according to hProlog standards - not safe to perform garbage collection. The following code is part of an example:

```
a(X,Y,Z,T) :- Z is X*X, functor(T,_,N), Y is Z+N.
```

Depending on the size of X, the computation of X*X or of Z+N can *overflow* the heap. hProlog cannot call the garbage collector at that moment (but see later for an alternative that was explored), so if X*X is too large to be put in the heap, it is put temporarily in an on-demand malloc-ed zone, the overflow flag is set and as soon as possible, garbage collection is performed. There are (at least) two alternatives;

- enhance the information needed for precise garbage with information about the temporary variables
- generate a rather different kind of code for arithmetic

The former was deemed too brittle (it depends on the register allocator) and the code explosion was high; the latter would have a too high performance penalty.

4 New built-in predicates and instructions

4.1 New built-in predicates

The new built-in predicates for strings and characters are rather obvious:

- `ilist_to_string/2`: transforms a list of integers into a string
- `string_to_ilist/2`: transforms a string into a list of integers
- `string/1`: succeeds if the argument is a string, otherwise fails
- `char_to_int/2`: transforms a character into an integer
- `int_to_char/2`: transforms an integer into a character
- `char/1`: succeeds if the argument is a character, otherwise fails

We have deliberately chosen predicates **_to_** which act *in one direction* only. For completeness, we also give the built-in predicates related to attributed variables:

- `attvar/1`: succeeds if the argument is an attributed variable, otherwise fails; a specialized version exists for use inside the if-then-else construct
- `put_attr_lowlevel/2`: the second argument replaces the attributes of the first argument
- `get_attr_lowlevel/2`: the second argument is unified with the attributes of the first argument
- `put_attr_lowlevel/3`: replaces the attribute of argument one corresponding to argument two by argument three
- `get_attr_lowlevel/3`: unifies the attribute of argument one corresponding to argument two with argument three
- `make_new_attvar/1`: turns a variable into an attributed variable with an empty list of attributes
- `bind_attvar/2`: binds an attributed variable to a value without wake-up; this is needed in the XSB and SICStus approach to attributed variables
- `attvar_method/1`: a fast test to decide on which basic attributed variable method is used; the return value is `hProlog`, `SICStus` or `XSB`

[5] has more details on the built-ins for attributed variables.

4.2 New instructions

The instructions related to characters are just variants of the instructions for atoms; a simple enumeration suffices: `build_char`, `unify_char`, `get_char`, `put_char`, `get_charv`, `put_charv`, `ifequal_char_goto`. The **v* variant work on environment slots (permanent variables) instead of on argument registers (temporary variables). The `ifequal_char_goto` is used in the condition of an if-then-else.

The instructions for strings are similar, but one must cater for the fact that not every string has the same length: `set_string`, `unify_string_untagged`, `get_string`, `put_string`, `get_stringv`, `put_stringv`, `bld_string`, `ifequal_string_goto`.

The `bld_string` instruction indicates the situation where a string is constructed on the heap while the corresponding string pointer has previously been constructed by `set_string`, as in the code for the goal `b(f(a,"asdasd"),c)` which is

```
put_structure A1 f/3
build_atom a
set_string 2
build_atom c
bld_string 6 asdasd
```

The *untagged* instruction is bit of a misnomer ⁶: it has an extra heap offset for constructing the string in case S ⁷ is zero. The float instructions are similar. Ideas from [13] and [12] are used here.

4.3 Indexing

The instruction `switchonbound` ⁸ uses a hash table for fast access of the appropriate clause(s). It is trivially extended to also work on input and head arguments that are of the type character or string. However, in HAL - just as in Mercury - a set of facts like

```
foo(asd) .
foo("hello") .
```

cannot be typed correctly, because a union type with string and some other data is not possible. The same applies to integers, floating point numbers and characters. We have therefore disabled indexing on arguments which are a mixture of values of these basic types and something else. We have not (yet) taken advantage of that by specializing the indexing instructions, but it has a detrimental effect on at least one benchmark: `cal.pl` contains the predicate

```
cal_key( 1, 6, 1) .
cal_key( 2, 2, 1) .
cal_key( 3, 2, 0) .
...
cal_key(jan, 6, 1) .
```

⁶thanks to Henk Vandecasteele :-)

⁷the WAM structure pointer in unify instructions

⁸inherited from XSB

```
cal_key(feb, 2, 1).
cal_key(mar, 2, 0).
...
```

to which hProlog does no longer apply indexing. The benchmarks show therefore timings for cal.pl and for simple_cal.pl, which is a version of cal from which we eliminated all the facts for cal_key with an atom in the first argument: they are not activated during the benchmark.

One more change was made to one classical benchmark: sdda contained

```
% "A" = [A],
```

which fails for native strings and so we replaced it by the more reasonable

```
A = 0'A,
```

4.4 Differences between dProlog and hProlog

Because our Prolog systems are in constant evolution, some differences in timings must be attributed to changes in the compiler, instruction merging ... When this is the case, we will indicate this in the table with benchmarks and discuss it.

5 Experimental evaluation

Extending the tag scheme - and the instruction set - of a system that is among the fastest around, runs the risk of slowing it down: we would not have been happy with a substantial slowdown and actually, we aim at showing that there is no need for a slowdown if more types are supported natively. We use the same set of benchmarks as in [7] and we simply compare the performance of dProlog1.0 which has the original tagging scheme with hProlog2.3.9 which has the new tagging scheme with the extra types supported: see Table 1. We also show the performance of SICStus Prolog 3.10.0 and Yap-4.4.2, just to make it clear that hProlog is not slow. One must keep in mind that the benchmarks are old, mostly badly written and that although hProlog supports (a superset of) Clocksin-Mellish Prolog (except for dynamic predicates), hProlog's ultimate aim is to support a language far superior to Prolog.

We have previously compared the performance of our attributed variables to the SICStus Prolog ones: see [5]. The bigints in hProlog are based on Section

4.3.1 in [11], and so are the bignums in SICStus Prolog: the comparison is therefore not interesting, i.e. the results are basically the same. Mainstream Prolog systems do not support strings or characters, so a specific comparison is not possible.

benchmark	dProlog	hProlog	SICStus Prolog	Yap
boyer	<i>7410</i>	5645	6115	5087
browse	7552	5290	<i>11667</i>	5270
simple_cal	1495	1000	1370	<i>1515</i>
chat	807	740	<i>1007</i>	840
crypt	3850	2460	<i>4237</i>	3902
ham	977	977	<i>1252</i>	852
meta_qsort	<i>1372</i>	1095	1132	1072
nrev	6472	6450	<i>11605</i>	6605
poly_10	557	385	490	470
queens	1807	1435	2385	1715
queens_16	1365	632	<i>1592</i>	700
reducer	<i>5362</i>	3517	4130	3382
sdda	<i>460</i>	372	370	340
send	6547	4332	<i>9345</i>	5240
tak	1235	837	<i>1282</i>	1142
zebra	4210	4560	<i>5097</i>	3485
cal	1480	<i>1637</i>	1357	1510
comp	1347	1300	<i>1755</i>	1442

Table 1: Timings for some benchmarks

The table shows in bold the figures that are best amongst the four and the worst figure is indicated in italic.

We have singled out two benchmarks: *cal* for reasons explained in Section 4.3, and *comp*, because it is not a traditional benchmark.

Of the 16 traditional benchmarks, hProlog is the faster on 9 and never the slowest, neither ever slower than dProlog. On modern hardware ⁹, the traditional benchmarks are no longer well suited for performance assessment, because they need to be repeated a million times in order to obtain timings that are close to a second. The last benchmark (*comp*) is an old (and adapted) version of the XSB compiler compiling itself (without dumping the generated code). It is the most realistic benchmark of all and hProlog also performs best for it.

hProlog is slower on *cal*, because it does not index on a badly typed argument. Note that these benchmarks were run without some optimizations that hProlog also can perform (like in-lining and switch improvement).

All together, Table 1 seems to show our point.

We must end however with a caveat: the choice of the version of gcc with which hProlog and dProlog is compiled is crucial to the findings. We have easy access in our department to gcc 2.95.4 and 3.0.4. hProlog runs slightly faster when compiled with 3.0.4 while dProlog benefits significantly from 2.95.4. So we have compiled each with the gcc version that gives best results. The benchmark that is most affected by the choice of compiler is ... *nrev* :-)

6 Discussion

[9] gives a very thorough discussion of *tagging*: it is not difficult to classify our tagging scheme according to the terminology used there. We have made particular choices with the following goals in mind:

- the address space is not further restricted
- performance is not compromised
- it does not make garbage collection more complicated

One can wonder whether there is still room for more types in hProlog. One can of course also overload even more any of the current types and a good candidate

⁹the benchmarks were run on an Intel 686, 1.8 GHz

is the STRING data type (which is not overloaded at the moment) and we have in experiments used for a native array type.

Even more types are more easily incorporated by using the tag-on-data scheme as in [16]. This is probably a good idea if a lot of types are needed. Another approach - followed by ECLiPSe as far as we know - is to use on a 32-bit machine 8 bytes for representing an atomic piece of information. ECLiPSe itself *proves* that such an approach is feasible and that performance need not be hampered too much by it.

Acknowledgments

Part of this work was conducted while the first author was a guest at the Institut de Mathématiques Appliquées of the Université Catholique de l'Ouest of Angers, France. Sincere thanks for this hospitality. We also thank Henk Vandecasteele for his work on the ilProlog compiler used within hProlog. This work was partly supported by project G.0144.03 funded by Fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen).

References

- [1] H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990. See also: <http://www.isg.sfu.ca/~hak/documents/wam.html>.
- [2] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ILP. In J. Cussens and A. Frisch, editors, *Inductive Logic Programming, 10th International Conference, ILP2000, London, UK, July 2000, Proceedings*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 60–77. Springer, 2000.
- [3] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.
- [4] V. S. Costa. Optimising Bytecode Emulation for Prolog. In *Proceedings of PPDP'99*, volume 1702 of *LNCS*, pages 261–277. Springer-Verlag, Sept. 1999. See also <http://www.ncc.up.pt/~vsc/Yap/>.

- [5] B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Computer Science, K.U.Leuven, Belgium, Oct. 2002. unpublished.
- [6] B. Demoen, M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999.
- [7] B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.
- [8] B. Demoen, P.-L. Nguyen, and R. Vandeginste. Copying garbage collection for the WAM: to mark or not to mark ? In P. Stuckey, editor, *Proceedings of ICLP2002 - International Conference on Logic Programming*, number 2401 in *Lecture Notes in Computer Science*, pages 194–208, Copenhagen, July 2002. ALP, Springer-Verlag.
- [9] D. Gudeman. Representing type information in dynamically-typed languages. Technical Report TR93-27, Department of Computer Science, The University of Arizona, Tucson, Arizona, 1993.
- [10] C. Holzbaur. Meta-structures vs. Attributed Variables in the Context of Extensible Unification. In M. Bruynooghe and M. Wising, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, number 631 in *Lecture Notes in Computer Science*, pages 260–268. Springer-Verlag, Aug. 1992.
- [11] D. J. Knuth. *Semi-numerical Algorithms*. Addison-Wesley, 1981.
- [12] A. Mariën and B. Demoen. A new scheme for unification in WAM. In *Proceedings of The International Symposium on Logic Programming, San Diego, October 1991*, pages 257–271, 1991.
- [13] M. Meier. Compilation of Compound Terms in Prolog. In S. K. Debray and M. V. Hermenegildo, editors, *Proceedings of North American Conference on Logic Programming*, MIT Press, pages 63–79, 1990.

- [14] F. L. Morris. A Time- and Space-efficient Garbage Compaction Algorithm. *Communications of the ACM*, 21(8):662–665, Aug. 1978.
- [15] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proc. of SIGMOD 1994 Conference*. ACM, 1994.
- [16] P. Tarau and U. Neumerkel. A novel term compression scheme and data representation in the binwam. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 73–87. Springer-Verlag, Sept. 1994.
- [17] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.

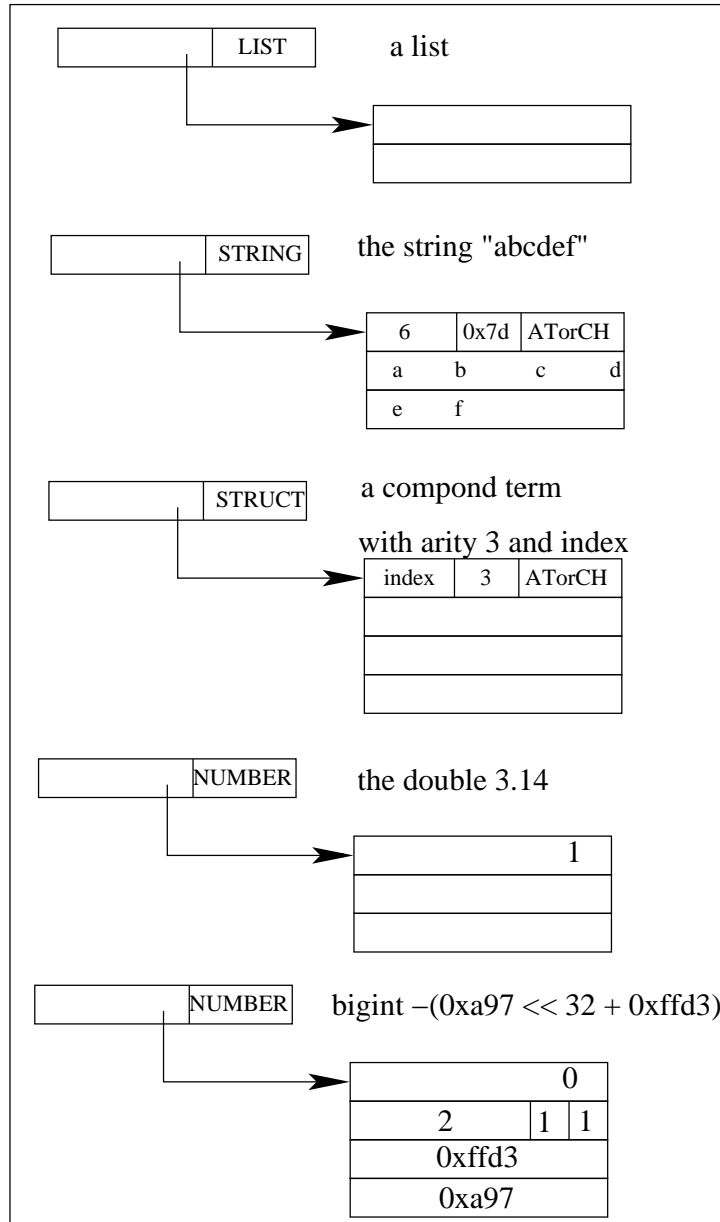


Figure 2: Other types