Implementation of Tag Representation in Prolog Virtual Machine

Guillaume Autran and Xining Li
University of Guelph
Computer and Information Science, The IMAGO Laboratory
Guelph, Ontario, Canada, N1G 2W1

Abstract

Even if it is understood that Prolog execution environment performs dynamic type checking on data being processed, it is often unclear how implementation of the language internally represents the information used to verify the type compatibility. In this paper we describe a Prolog tag encoding mechanism used in the Logic Virtual Machine (LVM) implementation of Prolog. The LVM is a Prolog engine that uses a merged heap/stack architecture and a virtual memory manager. We show that it is possible to retain the integrity of the data being tagged without requiring much extra memory and we explain the steps taken for proper unification.

Keywords: Prolog, logic programming, type representation, virtual machine

1. Introduction

For some, the origins of logic programming are quite mysterious. But for many people in Europe, logic programming was the most interesting research area. In the early 70's, there was a split between pure logic advocated by Robert A. Kowalski [7] in Edinburgh, Scotland and a more "hands-on" approach of logic that was going on on the other side of the channel in Marseille, France by Alain Colmerauer and his team [3]. Both university's extensive research and cooperation in their respective areas led to the birth of several logic machines such as the Q-systems or Algo-W, but most importantly it was there and then that "Prolog" [2, 3] was born. Prolog was at the time (and probably still is) the most practical implementation of logic programming (horn-clause). The language itself became very popular when Warren developed his famous Abstract Machine (WAM) a few years after his one-year stay in Marseille (1973-74). His WAM was best described much later by Hassan in [1] and the document uncovered the mystery behind the unification algorithm and the internal representation of data used by the interpretor. In one section of the document, Hassan describes the term representation in the way it interracts with the Prolog heap. Intuitively, he introduces the notion of "tag" used to differentiate the various cells within the heap. He also shows us the 4 different tag types that can complement the internal cell data structures, although he does not describe the binary encoding used.

In this paper, we explain how we extended the tag scheme described in [1] to provide more flexiblility and expand the encoding range in order to accept the most typical datatype required for today's Prolog systems. To this end, we briefly present the Logic Virtual Machine (LVM) [9] used to implement our ideas. This system's particuliar approach to memory management makes it unique in the sense that it is capable of providing a linear 4 Gb address space [1] to any running Prolog program. The merge Stack/Heap architecture ([8]) groups all runtime information and dynamically allocates data including choice frame, variable, *etc*. The advantage of such a scheme is that the garbage collection (GC) only needs to be invoked once on the stack for all dynamic memory. Constant table data and heap data will be reclaimed all at once by the GC along with any unused runtime data. Additionaly, a multi-threaded architecture allows the interpretor to run multiple applications concurrently without restricting their respective resource needs.

This article is organized as follows: Section 2 provides some background information necessary to understand the remainder of this paper. Continuing Hassan's description, section 3 provides a detail explanation of our tagging strategy by first explaining variables and structures representation in section 3.1, then discussing constants and list representation in section 3.2. Finally, we propose a summary of our implementation, offering some insight on the pros/cons of such a scheme and conclude this article.

1 The internal of the memory management of Prolog applications in the LVM interpretor is to be discussed in a later paper.

2. Overview

In his 1991 WAM book "A tutorial reconstruction", Hassan AÏT-KACI [1] discuss the internals of the not-so-clear Warren Abstract Machine. He explains in great detail the complete instruction set and shows most of the implementation theory required to understand and implement a WAM interpretor. In his discussion oriented toward the internal data structure of the abstract machine, Hassan introduces a set of 4 term types that can be encountered in the WAM heap [2] of a Prolog program. Typically, those terms are differentiated using one of the 4 tags described below:

- *<VAR> or <REF>* a pointer to another cell location containing any other type (possibly another *<VAR>* value).
- *<STR>* a structure tag which value points to a functor storage location. The first cell containing the functor name and *arity* (number of argument in the functor), while successive cells contain the arguments of that particular functor (if any).
- *<LST>* a pointer to the head of a 2-cell optimized list element. The first cell contains the head element while the second cell contains a *<LST>* pointing to the list tail.
- *<CON>* a constant value (such as a single lower case character).

Although no extensive comment is made regarding the bit encoding used to represent those 4 tags, one can assume that: **the smallest addressable item in a Prolog heap (a cell) is no smaller than an integer (32-bit); therefore, the lowest 2 bits can be used to encode one of 4 distinct tags.** In practice, the use of the lower 2-bit as a tag encoding does not disrupt the addressing capabilities of the program.

In later expansion of the original WAM (aka: many commercial versions of Prolog) the maximum of 4 term types became too restrictive and generated a real challenge for Prolog systems. To permit efficient support of a wide variety of terms needed by many programs such as integers, floating point value, character string, and binary data, a different cell tag encoding had to be used. A representation that would allow for a greater number of tags to be encoded within the data so dynamic type checking could be performed at run time with minimum impact on the performances. The most widely used solution (also the simplest to implement) is to reserve extra bits in addition to the original 2 in the cell. These extra bit representing an extended tag can be allocated either at the upper part of the cell (MSB) or right next to the original 2 bit in the low side of the cell (LSB).

In practice, these techniques are not without disadvantages, the worst of which may be the reduced precision of integer values, or the constraint afflicted to pointers with less than 30 significant bits. Regular pointers, used in the **VAR**, **LST**, and **STR** tags, are no longer referencing just "any" location in memory. This constraint forces the memory regions, such as the heap, to be located below a fixed boundary and reduces the flexibility of the dynamic loader.

Many commercial Prolog systems evolved and adapted to such restrictions while others have tried to overcome them by developing alternative options. The most notable is the use of the twin-cell model that allocates 2 regular cells (2 x 32-bit) as the basic size for a single Prolog cell. Obviously, the addition of 32 extra bits allow much more flexibility in term of tagging and does not impose memory location constraint on the program. Joachim Schimpf [10] also demonstrates that garbage collection can be further improved and optimized in such conditions and reduced tagging/untagging over-head effective. However, as Joachim states clearly in his article, "Obviously, the space consumption is higher, ...". In fact, the extra space required to store a twin-cell is exactly double the size of a single-cell scheme, and the tagging/untagging over-head is not proven to be reduced on the contrary of Joachim's claim.

Although literatures do not describe in detail many schemes used for Prolog term tag encoding purposes, it is our belief that most systems use a tag encoding represented on multiple bits (greater than 2). Some systems may allocate the least significant bits for that purpose, others may use a mixture of most significant bit(s) plus least significant ones. In either case, integer precision will be reduced unless some additionnal "tricks" are used.

3. Term Representations

The essence of logic programming applied to horn clause [7] rejects any data types, since logical operations performed on data unit do not depend upon types. However, implementation of logic programming concepts like Prolog do require some notion of types in order to perform efficient computations.

The Prolog language is said to be a "*typless and a purely declarative*" [6] language. Declarative languages describe relationships between variables in terms of functions or inference rules, letting the language executor apply some fixed algorithm to these relations to produce a result. The emphasis is made on the logic of a program in contrast to procedural languages, when the programmer has to specify an explicit sequence of steps to follow to produce a result.

---

2 In [1], Hassan use the word "HEAP" to refer to the memory area containing Prolog cells. The LVM merged STACK/HEAP architecture allows for a single memory region containing both HEAP and STACK data. However, for clarity purposes, we use the word HEAP to identify the memory containing the Prolog cells.

In Prolog a predicate argument (or variable) can be anything whatsoever. However, the relationship between variables can only be satisfied when the arguments possess particular properties, and some evaluable functors are defined only when the operands possess some particular property. Note also that although the control constructs built-in predicates and evaluable functors are defined for all arguments and operands, it is often an error unless an argument has a particular sort of value.

Therefore, the Prolog standard defines a set of basic types used to control the proper program execution and avoid useless and erroneous computations. For example "Z is X + Y" is not semantically meaningful if X, Y are instanciated to non-integer type variables and Z is not an unbound variable. It is therefore convenient when defining a declarative language to classify any term as belonging to one of several disjoint types.

At its origin, the WAM contained 4 different data types: variables (**VAR**) (named **REF** in [1]), constants (**CON**), lists (**LST**), and structures (**STR**). The advantage of such rudimentary schemes was the fact that only 2 bits are needed to encode all possible types. On a 32-bit architecture, using 32-bit cell aligned properly, the lower 2 bits of a pointer could be reused for type storing without conflict while keeping the address part of the pointer intact. As the language evolved over the years and "escaped" from the hands of its creators [3], more complex variable types were needed. Integers and floats allow for faster mathematical computations to be implemented and constant string grew to more than a single character (as in the original WAM). Nowadays most (if not all) Prolog implementations offer a variety of datatypes to the programmers to help them design more efficient programs and more complete algorithms.

Any Prolog implementation defines some sort of a bit pattern that allows the internal virtual machine to differentiate at run time the type of instanciated variable. This bit pattern size varies greatly according to the architecture. GNU Prolog implementation [4], for example, uses a 4-bit tag embedded within the cell to represent a particular type. 4-bit represents a maximum number of $2^4 = 16$ possible types. Among those 16 possibilities, various types such as integers, floats, strings, variables, optimized list, *etc*, have to cohabit happily, hoping not to interfere with one another. Unfortunately, this is not so true. Almost every type will suffer from losing 4-bit of information, such as 28-bit signed integer values or variable pointers. The programming community is so used to having standard datatypes and upper range values simmilar to C datatypes that reducing the range of the most basic types is hardly ever well accepted. Also, 28-bit pointers reduce the memory space available or impose constraints upon segment memory location for the Prolog program (such as quintus). To resolve those issues, particular implementation of Prolog language allocate twin cells where one word is used to store the datatype (along with some GC information) and the other word used for the value. This is the proposal made by Joachim Schimpf in [10] and tested with the SEPIA Prolog interpretor.

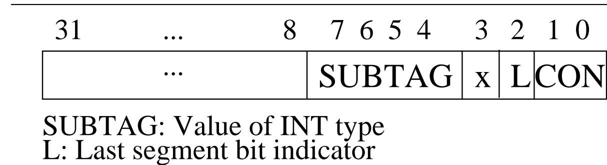

Figure 1. Heap representation of constant.

The approach taken by the LVM engine differs from all others in the sense that it assumes by default that only the 4 basic types are required (namely: VAR, LST, CON and STR). Those types are used in the same way as in the original WAM [1] proposal in order to facilitate and improve the engine's performance. The only difference between the original WAM and the LVM comes with the processing of the **CON**stant tag. In the original WAM, the **CON** tag is used for constant string only, while the LVM expands the idea of constant to any data type that cannot be modified on the spot. Therefore, integers, floats, octet string and multimedia content are all considered as constant falling into this category. The problem with a "one-size-fits-all" approach of this type, is to prevent association of incompatible types occurring and to help the engine identify such heresy prior to starting more costly computations. Therefore, a means to identify sub types is required.

The original WAM does not explain the meaning of the data part in the constant tagged cells. One may assume that it may be a single character stored or a reference pointer to a string of characters stored into a different location in memory. Regardless of the method used, sufficient information should be contained in the remaining bits of that tagged word to express the storage strategy. In the LVM implementation of constant, the data part (the upper 30 bits) are further decomposed into a 4-bit sub-tag, one special processing bit and one all-purpose bit as described in Figure 1. The subtag contains a 4-bit value providing up to 16 different data types. This representation leaves 24 bits of useful data storage.

3.1. Variables, Structures

A variable will be identified to a reference pointer and represented using a single heap cell. A variable cell will be identified by the tag **VAR**, as denoted by $\langle VAR, k\rangle$, where *k* is an address into the HEAP. Upon binding, the address part of the **VAR** cell points to the address this variable is bound to. By convention, *unbound* variables have their address part of the **VAR** cell set to their own address so to

0	$e/1$	
1	VAR	1
2	$b/2$	
3	VAR	3
4	VAR	1
5	$a/3$	
6	STR	2
7	VAR	3
8	STR	0

Figure 2. Heap representation of $a(b(C, D), C, e(D))$.

identify easily *bound* from *unbound* variables. Therefore an unbound variable is a self-referencial **VAR** cell.

Structures are not variable term. Thus, the HEAP format used for representing a structure $f(t_1, \ldots, t_n)$ will consist of $n+2$ HEAP cell. The first two of these $n+2$ cells are not necessarily contiguous since the first of the two acts as a reference pointer to the latter. This particular cell contains a tag $\langle STR, k\rangle$, where *k* is the HEAP address to the second cell used to represent the *functor* f/n. The n other cells (the arity) are destined to represent the root of the n subterms in proper order. The functor cell and its subterms must be contiguous in the HEAP storage area and each subterm must occupy no more than one single cell space. That latter property seems obvious in most implementations of Prolog, but it will be made clear later on that multiple cell term can also be represented in our implementation. To summarize, a possible representation of the functor $a(b(C, D), C, e(D))$ starts at address 7 in Figure 2 and occupies 11 cells total. Although Figure 2 is a compact representation example of a WAM based interpretor, the BinWAM implementation discussed in [11] propose an even more compact representation. However, the LVM interpretor implements the more traditional approach.

3.2. Constants and Lists

3.2.1. Lists Often used in Prolog programming is the special functor $'.'/2$ also known as list. A list contains a *Head* and a *Tail* represented as the two subterms of the list functor written using the notation $'.'(H, T)$, or the more common list notation $[H|T]$. It is possible to represent a list in the HEAP, the same way as any other functor as shown in Figure 2. Since lists in prolog are frequently used, an optimization is being made to improve the memory requirements and processing speed of the datastructure. To this extent, we introduce a new tag $\langle LST, k\rangle$, where *k* is a reference to a two cell structure containing the head part followed by another $\langle LST, k\rangle$ cell pointing to the tail of the list. A special *nil* terminator, written [], is used to end the list. For example, the following list $[A, B, [C, B]]$ can be re-written as $'.'(A,'.'(B,'.'('.'(C,'.'(B, [])), [])))$ and Figure 3, starting at cell 4 is its possible HEAP representation.

0	VAR	0	C
1	LST	2	
2	VAR	2	B
3	[]		
4	VAR	4	A
5	LST	6	
6	VAR	2	B
7	LST	8	
8	LST	0	
9	[]		

Figure 3. Heap representation of $[A, B, [C, B]]$.

0	0xFFFFFF	INT	x	0	CON
1	0xFFFFFF	INT	x	0	CON
2	0xFFFF	INT	x	1	CON

Figure 4. Heap representation of a 64-bit integer.

3.2.2. Integer Constants The basic type needed in most computer programs is the integer. In standard programming, the range of a signed integer is dictated by the maximum number of bits in a half-word for a 16-bit architecture, a word for a 32-bit architecture and a double-word for a 64-bit architecture. Numerically speaking, -32768 to 32767 for a 16-bit integer, -2147483648 to 2147483647 for a 32-bit integer and -9223372036854775808 to 9223372036854775807 for a 64-bit integer. If we had to use the WAM **CON** type alone, integer range would be reduced to 30-bit integer minus any bits allocated to differentiate the types. Instead, the LVM machine allocates the minimum amount of space required for the information stored.

The HEAP representation for the different length integers follow the concept shown in Figures 4, 5 and 6. The integer is broken down into 3-byte units stored individually in each successive cell. The last cell is marked using a special bit, denoted *L*, in the subtag that identifies the end of the sequence.

This method of representation allows us to compress the integer value to a minimum storage requirement by removing the leading 0's in the value. A given integer value n has

0	0xFFFFFF	INT	x	0	CON
1	0xFF	INT	x	1	CON

Figure 5. Heap representation of a 32-bit integer.

0	0xFFFF	INT	x	1	CON

Figure 6. Heap representation of a 16-bit integer.

for binary representation b, where b contains 0's and 1's. If all the leading bits in b have the same value (for example 0), therefore, they can be omitted in the stored representation of n. For example, the functor $f(64, g(1234567890, A))$ contains two positive integer number, 64 and 1234567890. The binary representation of the number 64 is 0100 0000 with 24 leading 0's that can be removed while writing the value into the cell. Therefore, only 8 bits of useful storage are required to preserve the value. Figure 7 gives a possible HEAP representation of the functor $f(64, g(1234567890, A))$ starting at address 5.

In practice, no integers wider than 64-bit are supported by this particular implementation. However, wider integer support could easily be implemented using this scheme, assumming that a special arithmetic unit was implemented to support wider number arithmetic.

Another aspect of integers is the representation of signed values. We saw before the basic representation used for positive integers, so let us demonstrate how this scheme can be used effectively with the negatively signed integer. A common representation of a negative number uses a 2's complement. It is achieved by inverting all the bits of the absolute part of the number and adding 1 to the result.

0	0x9602D2	INT	0	0	CON
1	0x49	INT	0	1	CON
2	$g/2$				
3	0				VAR
4	4				VAR
5	$f/2$				
6	0x40	INT	0	1	CON
7	2				STR

Figure 7. Heap representation of $f(64, g(1234567890, A))$.

0	0x9602D2	INT	0	0	CON
1	0x49	INT	0	1	CON
2	$g/2$				
3	0				VAR
4	4				VAR
5	$f/2$				
6	0x3f	INT	1	1	CON
7	2				STR

Figure 8. Heap representation of $f(-64, g(1234567890, A))$.

$$0001\ 0001_{(binary\ 17)} \Rightarrow 1110\ 1111_{(two's\ complement\ -17)} \tag{1}$$

$$NOT(0001\ 0001) = 1110\ 1110\ (Invert\ bits) \tag{2}$$

$$1110\ 1110 + 0000\ 0001 = 1110\ 1111\ (Add\ 1) \tag{3}$$

This form of representation permits arithmetic between signed and unsigned numbers to be performed the same way as operations between the same sign numbers without further consideration. Therefore, the presence of a 1 in the Most Significant Bit (MSB) position indicates a negatively signed number, while a 0 at this location means a positively signed number. Looking at the binary representation, one can see that our nicely setup compression scheme no longer works as the high order bits are always set to 1 for negative number, even for the small numbers which normally would benefit most from the compression.

To resolve this issue, we decided to modify the number representation such that we could benefit from the compression scheme discussed earlier. In fact, we only need to invert (again) the binary representation of negative numbers so that the high order bits are reset to 0 while not losing precision. The obtained value is handled the same way as before and the all purpose bit is used to indicate the sign.

$$1110\ 1111_{(binary-17)} \Rightarrow 0001\ 0000_{(invertbits)} \tag{4}$$

Encoded $\Rightarrow$

0x000010	INT	1	1	CON

Decoding the integer is done in a reverse way where we first reassemble the value and then apply the inversion if the sign bit is present and set. Similar to Figure 7, Figure 8 is the HEAP representation of the functor $f(-64, g(1234567890, A))$, where -64 (in this case) is the negative value of the integer number 64.

This method allows us to compress any type of integers, signed or unsigned, and to save space in the interpretor's HEAP. The precision is not affected by the compression and

0	0xFFFFFF	FLT	x	0	CON
1	0xFFFFFF	FLT	x	0	CON
2	0xFFFF	FLT	x	1	CON

Figure 9. 64-bit double precision float

the range remains equivalent to the native datatype. Unification is left simple and performed on the encoded value, taking care to unify the extra cells allocated. Decoding numbers is only made necessary while performing mathematical operations such as comparison or addition.

3.2.3. Float Constants Similar in essence to the integer representation, the floating point type uses the same scheme to store the actual value of the data. However, not as much flexibility is allowed. Floats value are represented using the IEEE Standard [5] representation using the double precision storage method. The standard offers several storage formats for single precision or double precision using 32-bit (1-bit sign, 8-bit exponent and 23-bit fraction with a bias of 127) or 64-bit (1-bit sign, 11-bit exponent and 52-bit fraction with a bias of 1023) respectively. Additionaly, a possible extended double precision format exists increasing the range of the floating point number. For practical reasons, only the double precision floating point number implementation is considered in the LVM engine. Figure 9 shows the memory layout of floating number in a very similar way to the 64-bit integers of Figure 4. In fact, the representation is identical for simplicity purposes. A total of 3 words are required to store a complete 64-bit double precision floating point value. Again, the 'L' bit is only set in the last word terminating the value.

This type representation is oriented toward a fast and simplified unification while keeping the range of numbers similar with those of standard C programming. It also helps to reduce the memory requirements by providing more compact representation of integers without sacrificing precision. The simplistic unification process for **CON**stant type is as simple as comparing each successive integer until the 'L' bit is set or the result of a comparison fails. If all words perfectly match, both terms are identical (type and value) and unification succeeds. It will fail otherwise. Mathematical evaluation is, however, a little more complex due to the fact that the value must be reassembled from the storage format to a format suitable for native computation, then converted back for storage.

3.2.4. Octet string and Binary Constants In addition to basic integer and float types, the LVM engine defines two more complex types: octet strings and binary content. Octet strings are strings of characters (regardless of the character encoding) usually dynamically created along the execution of the Prolog program. Binary content type contains a much broader set of data that can vary from streaming video to still images, the content of a text file, or just a binary file. In general, binary content type can hold any data that does not fit in any other predefined datatype and that does not require operations to be performed other than unification. The entire content of those two types is stored within the HEAP in successive order.

0	'a','b','c'	OCT	0	CON
1	'd','e','f'	OCT	0	CON
2	'g','h','i'	OCT	0	CON
3	'j', 0, 0	OCT	1	CON

Figure 10. Heap representation of the ASCII string "abcdefghij".

0	nb of	BIN	0	CON
1	bytes	BIN	1	CON
2	BINARY DATA			
3	BINARY DATA			
4	BINARY DATA			

Figure 11. Heap representation of a binary object.

Octet string (tag **OCT**) encoding differs significantly from binary (tag **BIN**) content, based upon the assumption that strings have reduce length (in general) and binary content may be several mega bytes in size. The encoding of an octet string, shown in Figure 10, follows the basic construction principle as explained earlier in this paper, with a 1-byte tag used to indicate the type of the data and the implicit length (using the 'L' bit). The encoding of multimedia content however, contain two sections: first the tag cells containing the length of the object, then the object itself as illustrated in Figure 11.

From the encoding structure used by the constant type, the first-order term unification is kept relatively simple. A simple comparison of integers followed by a bit-wise test of the 'L' bit is enough to cover most of the constant types (except for the binary type). The exception is made for the binary type that requires more processing. For this type, the header should first be unified and then, if there is a match, a word comparaison of the number of bytes divided by the size of a word (in this case 4 bytes) must follow to insure the identicality of both terms. Optionally, memory reduction can be performed when two identical terms of non negligible size are unified by simply replacing one of them with a reference to the other term.

4. Conclusion

To allow dynamic type checking, Prolog engines encode datatype along with their data. This information is retreived at runtime to perform operations on the data according to the algorithm described in the program. Our solution to type encoding implemented in the LVM engine is a tradeoff between memory requirement and computation needs while not affecting the actual data to be stored. Basic types, such as variables (**VAR**), lists (**LST**) and structures (**STR**) require only a 2-bit tag while constants (**CON**) uses a 2-bit tag along side a 4-bit subtag. The actual constant data is compressed, encoded and stored in the remaining useful bit within the cell without loss of precision (for numerical values) nor loss of information for more data oriented types. The actual computation cost required by the compression and encoding process are being currently evaluated. It is our belief that the simplicity of the unification process and the amount of memory saved, are enough to justify the extra processor cycles. This solution is being implemented in the LVM Prolog interpretor and is being currently extensively tested for computational efficiency and memory efficiency.

References

[1] H. Aït-Kaci. *Warren's abstract machine: a tutorial reconstruction*. MIT Press, 1991.

[2] A. Colmerauer. Prolog in 10 figures. *Commun. ACM*, 28(12):1296–1310, 1985.

[3] A. Colmerauer and P. Roussel. The birth of prolog. In *The second ACM SIGPLAN conference on History of programming languages*, pages 37–52. ACM Press, 1993.

[4] D. Diaz and P. Codognet. The GNU prolog system and its implementation. In *SAC (2)*, pages 728–732, 2000.

[5] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[6] International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England. *PROLOG. ISO/IEC DIS 13211 — Part 1: General Core*, 1994.

[7] R. A. Kowalski. *Logic for Problem Solving*. Prentice Hall PTR, 1979.

[8] X. Li. Efficient memory management in a merged heap/stack prolog machine. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 245–256. ACM Press, 2000.

[9] X. Li. The logic virtual machine (lvm) specification. 2000.

[10] J. Schimpf. Garbage collection for Prolog based on twin cells. In *2nd NACLP Workshop on Logic Programming Architectures and Implementations*. MIT Press, 1990.

[11] P. Tarau and U. Neumerkel. A novel term compression scheme and data representation in the binwam. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming: 6th International Symposium (PLILP'94)*, pages 73–87. Springer, Berlin, Heidelberg, 1994.