PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

BENCHMARKING OF PROLOG PROCEDURES FOR INDEXING PURPOSES

Micha Meier

European Computer-Industry Research Centre GmbH
Arabellastr. 17, D-8000 Muenchen 81
West Germany

ABSTRACT

Indexing, i.e. restricting the number of Prolog clauses selected for the unification with a call, is based on a preselection using the type and value of one or more of the call arguments. A good indexing scheme is one of the key features for a fast Prolog system. Since for the design of an efficient indexing scheme it is necessary to take into account the form of real-life Prolog clauses, we have analyzed a large number of medium to large Prolog programs mainly with respect to indexing on the first 9 arguments.

1 INTRODUCTION

Prolog is a logic programming language based on the concept of resolution and unification (Robinson, 1965). One resolution step consists of selecting one literal (the leftmost one), choosing a clause and then unifying the literal with the clause head. As the Prolog program can contain a large number of clauses, for an efficient execution it is necessary to reduce the number of clauses that are chosen to match the selected literal, otherwise the system executes too many redundant backtracking steps.

The simplest and obvious way is to select only clauses whose head predicate match the predicate in the literal. For realistic applications, however, it is necessary to introduce other concepts to refine the selection mechanism. *Indexing* in the WAM (Warren, 1983) is of a great help here: the first argument of the predicate is used as another key to search for the matching clause - if the type and/or value of the literal argument matches the one of the clause, the clause is selected. Although it looks only like moving a part of the unification to the indexing step, it means more than that - it helps to avoid unnecessary backtracking and often it will select only one clause so that the overhead of managing a choice point frame is avoided.

Various current Prolog systems use various indexing strategies, but all in all, there has been no systematic information available about the form of the head of Prolog clauses and their suitability for indexing, or for one or another indexing type. Some systems index on the first argument, others on another chosen argument or even on more arguments. It is clear that for each indexing type there are real-life examples for which it is the most suitable one, but we are lacking a common basis for them.

Based on (Meier, 1985) we present here the results of static benchmarking of a number of large Prolog programs, whose aim was to obtain enough information to decide which of the indexing methods are generally applicable. We are well aware of the limitations of a static analysis for runtime behavior of Prolog programs, but even such analysis can contribute to better understanding of real-life Prolog applications.

These measurement were performed as part of the SEPIA project (Meier et al., 1988) and SEPIA did benefit from its results by having a concise and powerful indexing scheme.

The rest of this paper is divided as follows: in section 2 we describe some basic concepts, in section 3 we describe the benchmarked figures, in section 4 we report the general properties of indexable and not indexable procedures, section 5 presents the statistics about the occurrences of various argument types, sections 6, 7 and 8 contain the results concerning the various types and the section 9 presents a summary of the main results.

2 TERMINOLOGY

We first define some basic concepts that will help to explain the benchmarking approach and results. A *type* of a clause argument is either *variable*, *constant*, *list* or *functor* with

the usual Prolog meaning. A clause is of type T (w.r.t. the i-th argument) if the i-th argument of its head is of type T, similarly a goal is of type T if its corresponding argument has type T. We will call a clause of type T a *T clause*, e.g. a variable clause. A *block* is a sequence of clauses with the same type (the largest such sequence). A *T block* is a block consisting of clauses of type T; a *nonvariable block* is either a constant, list or functor block. A *section* is a sequence of consecutive nonvariable blocks. All these notions are defined always with respect to a given argument position.

Each procedure can therefore be uniquely divided into sections and blocks w.r.t. an argument position. The first block in a procedure is called a *leading block*, the last one is a *trailing block*, other blocks are called *internal blocks*.

The meaning of the introduced concepts is obvious - for a nonvariable goal it is necessary to match at most all clauses of the same type (i.e. blocks) as well as variable blocks. A variable goal must try to match all blocks. In terms of (Warren, 1977), the *special section* corresponds to our *section* and a *general section* is a *variable block*. Since we are going to handle different block types, our terminology seems clearer.

The indexing can take place at several levels - it may choose sections, blocks or single clauses. The most desirable one, which we will concentrate upon is of course the latter.

3 BENCHMARKED ITEMS

We collected 52 medium to large Prolog programs and we ran this test suite through a series of analyzing Prolog and C programs to collect the required data. A detailed description of the benchmarked programs is beyond the scope of this paper, 19 of them were developed at ECRC, the remaining ones came from outside, they include database programs, theorem proving, expert system-like, natural language processing, several Prolog compilers, programs from the Stanford Prolog library etc.

In our previous report (Meier, 1985) we performed an analysis of a lower number programs and only with respect to the first argument. We have now generalized these results to more programs and up to the 9th argument, separately for each argument, however the results for arguments higher than 6 are not significant since there was only a small number of procedures with a sufficiently high arity. In contrast to (Meier, 1985) we have paid more attention to procedures that cannot be indexed and to compound arguments, several parameters are presented in a graphic form. Some of the results would deserve a more thorough analysis which would go beyond the scope of this paper, we hope that the concise graphic form will give enough start momentum for an interested reader. In the presented figures, the x-axis represents the given predicate argument position. The y-axis often represents the percentage of procedures with the respective property. Usually the percentage is relative to all procedures indexable on the given argument, sometimes it is taken as relative to procedures with another property, e.g. the type of the first element of a list argument is represented relatively to procedures that have some list clauses on that argument.

When presenting the figures, we have tried to consider the influence of both large and smaller[1] programs. For all figures we have computed the absolute average value as well as the relative one. For example, for the number of clauses in a procedure, the absolute average is the sum of clauses in all programs divided by the number of procedures in all programs, the relative average is the sum of averages in each program divided by the number of programs. When describing the results, we always use the relative average since it gives the same weight to all programs, independent of their size, otherwise the longer programs would prevail even if they might be less important than the short ones.[2]

Our analysis was static and we have not taken into account possible *mode declarations* since they were available only in very few programs and therefore the modes, i.e. input or output for the procedures were not known. However, from the programming practice it is known that most of the procedures were written with a specific mode in mind, only very few procedures use Prolog's ability to be called with different instantiations of arguments. This means that our results are at best upper bounds of the realistic figures, many of the procedures which we classify as indexable on a certain argument are in fact not, since that argument is used for output. Despite this bias, our results are still significant, be it nothing else than to state these upper bounds.

4 INDEXABLE PROCEDURES

We have benchmarked 52 Prolog programs, consisting of 8245 procedures and 25471 clauses, and we have performed

[1] or as well large programs in which only a small percentage of procedures were indexable

[2] The important results were confirmed in both of the averages, though.

the same analysis on each of the first nine procedure arguments, for procedures with sufficiently high arity.

Some procedures cannot be indexed on a given argument for one of the following reasons:

- the procedure has lower arity
- the procedure contains only one clause
- the procedure contains only a variable block

Most of the measurements were performed only for the indexable procedures, the whole set of procedures occurs only in the figure 4-1.

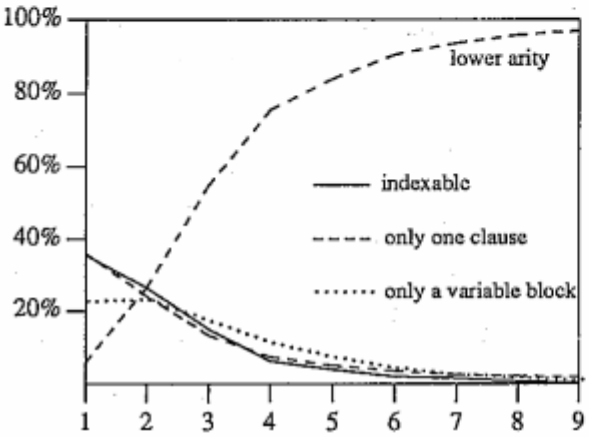

Figure 4-1: Percentage of Indexed and not Indexed Procedures

As it can be seen in figure 4-1, the number of procedures decreases almost linearly with increasing arity up to 4, and among the remaining procedures, the number of indexable and not indexable procedures (only one clause or only a variable block) decreases in the same extent. This picture helps to realize how the number of representative procedures decreases with increasing arity.

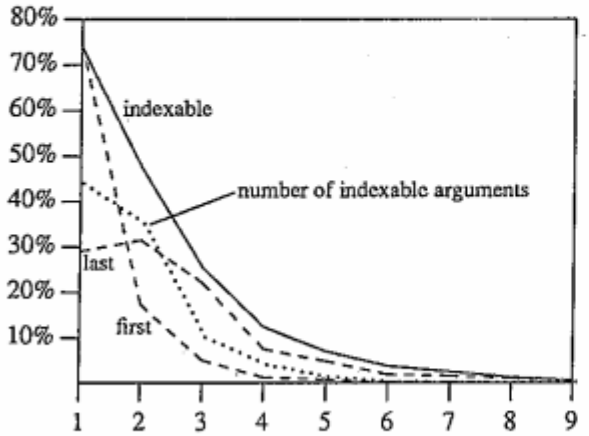

Figure 4-2: Indexable Arguments

In figure 4-2 we show the main results concerning the indexable arguments, all the figures are relative to the indexable procedures. The dotted line represents the percentage of procedures that are indexable on the given number of arguments, for all the other curves the x-axis represents the argument position and the y-axis the percentage of procedures. It includes the procedures indexable on the given argument, percentage of procedures for which this is the first (i.e. lowest) and last (highest) indexable argument. There are several important results here:

- Only 47% of all procedures are indexable.
- On any given argument, only less than 40% of procedures that have a sufficiently high arity are indexable.
- Not indexable procedures are quite frequent, procedures containing only a variable block or only one clause represent about 35% each (of procedures with a sufficiently high arity). This means that shallow backtracking (Meier, 1986) for the former and in-line expansion (partial evaluation (Venken, 1984)) for the latter might be significantly helpful to improve the performance. It is interesting to note that from this *statical* analysis it follows that these two mechanisms are as important as indexing.
- Most of the indexable procedures can be indexed on one or two arguments.
- 26% of indexable procedures cannot be indexed on the first argument.
- The ratio of procedures which are not indexable because they have only a variable block is lower in the first one than in the following arguments (figure 4-1). This suggests that the first argument, even for programmers that do not know about indexing, is considered different from the others.

The very low ratio of indexable procedures is indeed surprising; if we look at the number of *clauses* that belong to these procedures, the situation is different: for the first argument, 67% of clauses belong to the indexed procedures and only 33% to the not indexable ones. This is due to the fact that procedures with only a variable block are short, their average number of clauses is 2.7 for the first argument and it grows to 3.3 in the 5th argument. Single-clause procedures contribute to this figure as well.

For indexable procedures, we can see the following consequences:

- The most likely selected argument is the first one (the *first* curve sinks rapidly after the 1st argument) which means that the intuitive strategy of many

Prolog systems is correct. However, restricting the indexing to the first argument rules out 26% of the indexable procedures.

- If the first or second argument is not indexable, then the chances decrease rapidly that another one will be found.
- Except for the first argument, if an argument is indexable it is likely that no higher indexable argument exists.

5 ARGUMENT TYPES

Now we are going to describe some general properties of blocks and their types.

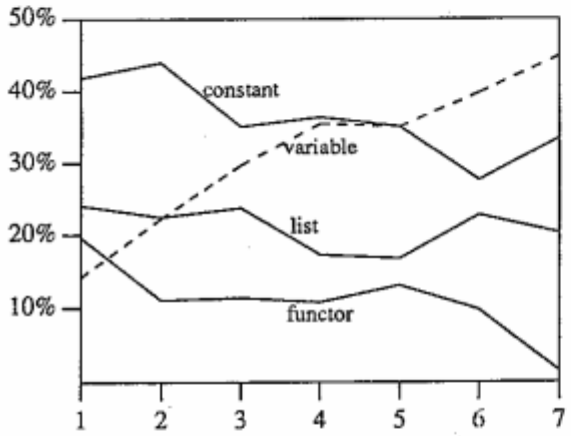

Figure 5-1: Percentage of Clause Types

From the figure 5-1 it follows that the first argument is mostly a constant, in higher arguments the ratio of variables is increasing whereas all other types become less frequent. This confirms the assumption that lower arguments contain more information, higher arguments are likely to be output ones and are bound later in the clause.

The average length of variable and list blocks is relatively stable in all arguments (figure 5-2), the length of constant and functor blocks is higher and it differs in different arguments. It means that indexing on the value of constant and functor arguments is indeed helpful. The average length of nonunit list blocks is about 3, which suggests that indexing on list elements may improve the performance.

The figure 5-3 shows the combinations of various types in procedures, without considering variable clauses, e.g. the 'list' curve describes what percentage of procedures consists

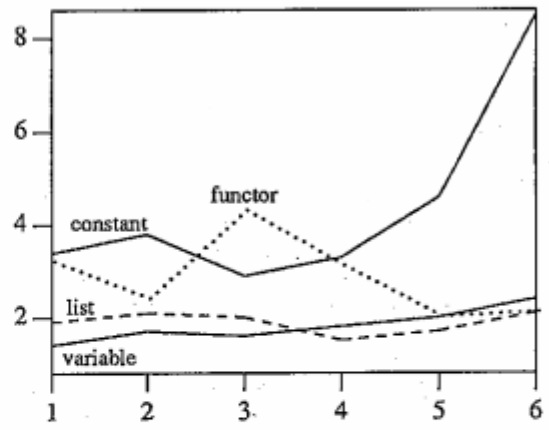

Figure 5-2: Blocks Length

only of list and maybe variable clauses. In most of the procedures only very few different types occur, the most frequent ones are list and nil, constants, or functors. **About 95% of all procedures (for all arguments) contain only these type combinations.**

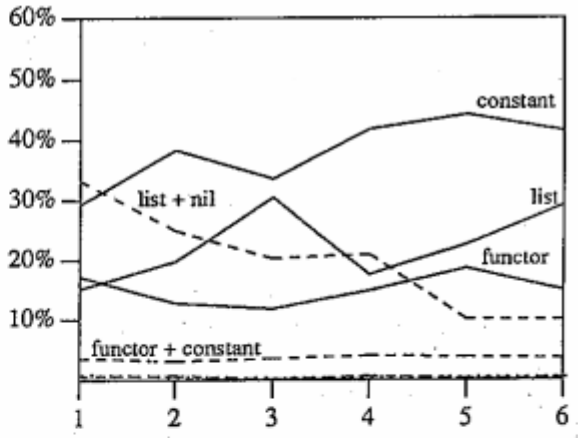

Figure 5-3: Combinations of types in procedures

We have not classified *nil* as a special type, but we have measured the number of procedures with both list and constant blocks in which the constant is not nil; it was negligible, e.g. from 913 procedures that contain a list and constant block in the first argument only 13 contain a constant different from nil, for other arguments it is similar.

This implies that the usual WAM method of indexing, i.e. executing the **switch_on_term** instruction that selects a path for the argument type is too general in most cases - usually only one or two of its labels are different from *fail*. For

constants and functors another switch instruction is executed which tests the value of the argument.

We suggest introducing new indexing instructions which consist of a combination of these two *switch* instructions:

list_switch A_i, LList, LNil, LDefault
if the dereferenced value of A_i is a list, jump to LList, if it is nil, jump to LNil, if it is a free variable then continue in sequence, otherwise jump to LDefault. Such an instruction can be generated for almost 50% of procedures indexable on the first argument.

atom_switch A_i, Tab, LDefault
if the dereferenced value of A_i is an atom, use the hash table Tab for further dispatch, if it is a free variable then continue in sequence, otherwise jump to LDefault.

integer_switch A_i, Tab, LDefault
functor_switch A_i, Tab, LDefault
these instructions are similar to the **atom_switch** instruction

Constants others than atoms or integers almost do not occur (a special case is *nil*, see section 7) and so these instructions are sufficient. The original **switch_on_term** instructions must be of course kept since some procedures do contain several types after all.

About 50% of the indexed procedures have lists or nil in the first argument, this highlights the importance of lists in Prolog.

6 VARIABLE BLOCKS

Another important figure concerns variable blocks. In the original WAM, variable blocks in a procedure can cause creation of an additional choice point for each section, especially for internal variable blocks. Such a scheme minimizes the space used for the code but in terms of execution time it does not perform well. Other systems (Van Roy, 1984, Bowen et al., 1986, Turk, 1986, Carlsson, 1987) use different strategies where the code for the variable blocks is less or not at all shared which makes the second choice point obsolete, but the generation of indexing instructions is more complicated.

From the figure 6-1 we can see that in the first argument, most procedures contain no variable blocks. If they contain some, it is mostly a trailing variable block which is often used as catch-all clause(s) at the procedure end. Leading variable blocks occur far less frequently and internal blocks are rare (at least in the first argument).

The length of trailing and leading variable blocks is increasing with higher arguments, the length of internal blocks

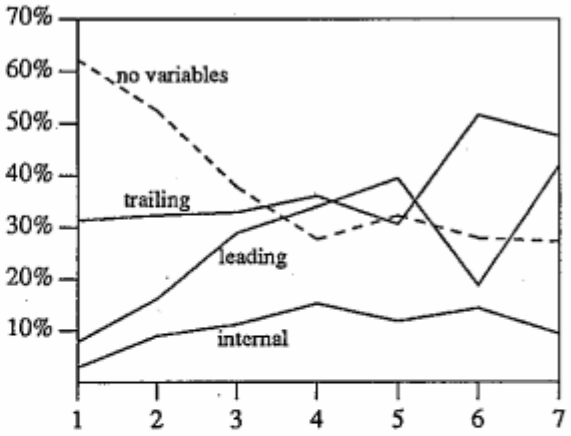

Figure 6-1: Percentage of Variable Blocks

is first decreasing and then it increases as well (see figure 6-2). Many of the trailing blocks contain more than one clause.

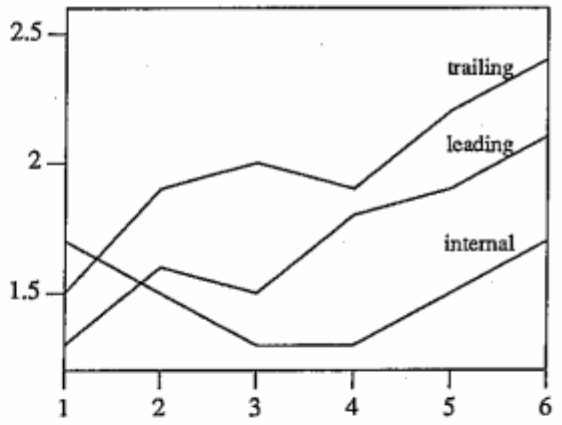

Figure 6-2: Length of Variable Blocks

A consequence of these figures is, that in most cases, a very simple algorithm can be used to generate the indexing instructions, since in the first argument of more than 90% of procedures there are either no variable blocks or only a trailing variable block. However, since trailing variable blocks represent the most frequent ones, it is preferable that the code for them is shared among different possible paths. For a procedure

```
p(a).
p(b).
p(X).
p(Y).
```

the usually generated code could be

the variable path:

```
        try_me_else L2
C1:     <clause 1>
L2:     retry_me_else L3
C2:     <clause 2>
L3:     retry_me_else L4
C3:     <clause 3>
L4:     trust_me_else_fail
C4:     <clause 4>
```

code for a:

```
        try     C1
        retry   C3
        trust   C4
```

code for b:

```
        try     C2
        retry   C3
        trust   C4
```

The code that executes the last two clauses is generated three times (if there are more atoms in the procedure it is even more), although it does exactly the same job - backtrack through the trailing variable block. We have introduced in SEPIA the instruction **try else** that is a mixture of the **try** and **try_me_else** instructions in that it executes the code at one label (like **try**) and sets the alternative clause address to another label (like **try_me_else**), similarly for the **retry** instructions. This is extremely simple to include into a WAM emulator; then the previous example can be coded as

variable path:

```
        try_me_else L2
        <clause 1>
L2:     retry_me_else L3
        <clause 2>
L3:     retry_me_else L4
        <clause 3>
L4:     trust_me_else_fail
        <clause 4>
```

code for a:

```
        try     C1 else L3
```

code for b:

```
        try     C2 else L4
```

With no runtime and compile time overhead, the generated code is shorter.

The same approach is not as straightforward for the leading variable blocks, but fortunately they do not occur so often so that such modification is not necessary. Since internal blocks do not occur frequently (and most of their occurrences could be removed by reordering the clauses without changing its semantics), a conservative approach, e.g. that one creating two choice points is still acceptable[3].

[3]Note however, that having two choice points for one procedure might make the *cut* implementation more difficult, since instead of using one bit to indicate whether the current procedure has a choice point or not, we need two bits or the address of the appropriate choice point.

7 CONSTANT ARGUMENTS

Constants are, at least in the first arguments, the most frequent type. A vast majority of the constants are atoms and integers, *nil* is presented much less frequently, floating numbers are very rare. No other constant types were encountered in the benchmarked programs (e.g. strings or database references). This justifies that only the instructions **atom_switch** and **integer_switch** are necessary.

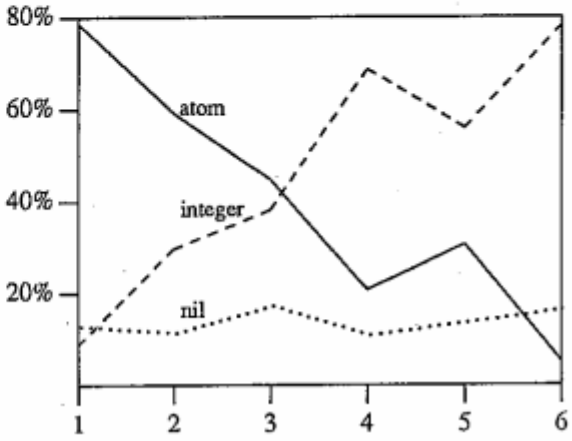

Figure 7-1: Different Constant Arguments

8 COMPOUND ARGUMENTS

Apart from measuring the occurrence of compound types, we have also followed the *arguments* of compound terms, their arity and size. These figures may be useful not only for indexing but also for the compilation of the head unification. We have made a distinction between lists and other compound terms; for lists we have measured the number of elements and the type of the first elements, for other compound terms it was the arity of the main functor, the arity of the subarguments, the size of the whole compound term which is 1 + arguments size (i.e the functor + arguments), size of a constant is 1, and the number of functors, i.e. of compound subterms including the main one. The size roughly corresponds to the number of WAM instructions generated for its unification. Apart from the results in the figure 8-1 it is interesting to note that

- Most lists have only one element, but in higher arguments the lists are longer.
- The arity of functors is usually 1 or 2 and the arity of the main functor is similar to the arities of the subarguments. Top functor arity was 9 which is surprisingly lower than the top arity of a predicate which was 23 (1% of procedures had arity greater than 9).
- The number of functors, i.e. the number of

compound terms is mostly 1, which implies that the compound subarguments are rare.

- The size of compound terms is usually 3 or 2 which means that terms are mostly of the type f(X, Y) or f(X), the top size was 39 (CHAT-80).

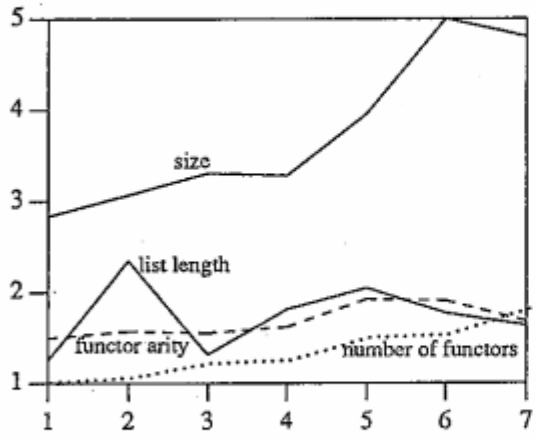

Figure 8-1: Mean Parameters of Compound Arguments

To be able to index nonunit functor blocks, the functors must be different. In the first argument, about 70% of them contain at least two different functors, in further arguments this figure decreases to 50%.

Generally, in higher arguments the compound terms are larger, have greater arity and contain more compound subarguments. This is an interesting feature whose reason might be that higher arguments represent output of the procedure.

8.1 List Arguments

Since many of the list blocks contain more than one clause, with the usual WAM-like indexing it is not possible to select only one clause. We have analyzed the possibility to index as well on the elements of the list.

Among procedures that contain a nonunit list block, as far as the first list elements are concerned, in about 50% of the procedures they are only a variable, in about 30% they are unifyable terms or variables, e.g. *X* and *a*, in about 7% they are completely identical and only 20% of procedures contain some really different values in the first list element. These figures are fairly consistent over all the arguments.

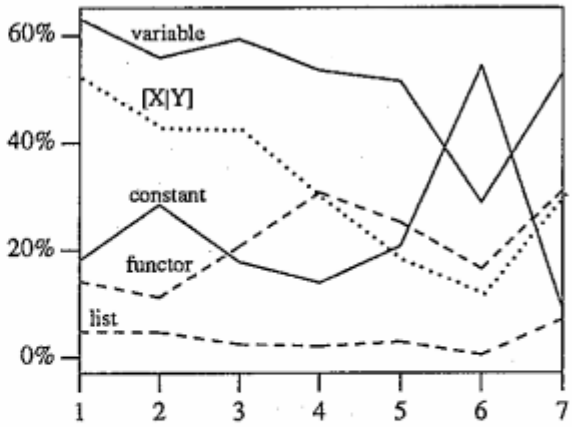

Figure 8-2: Percentage of Types in the First List Element

The figure 8-2 shows the types in the first list element relative to all list (i.e. even unit) blocks. It can be seen that most of them are variables, many lists are only pairs [X|Y], lists as list arguments are rare. In higher list elements this is different, variables in the list tail are less frequent.

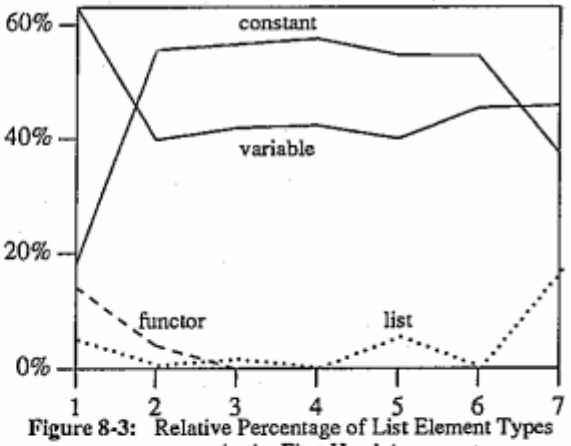

Figure 8-3: Relative Percentage of List Element Types in the First Head Argument

The figure 8-3 shows the type of lists elements in the first head argument (in the x-axis *i* is the *i-th* list element). It can be seen that the subsequent elements are different from the first one - most of them are constants.

9 CONCLUSION

We have presented the results of a static analysis of a large number of Prolog programs. Despite the fact that our analysis was static and that the mode of the arguments was not

known, i.e. silently supposed to be input, the analysis has shown some new results and confirmed some intuitive assumptions about Prolog programs:

- The percentage of indexable *procedures* is very low, less than 50%, although in terms of clauses it is about 67%.
- Procedures that cannot be indexed have only one clause, in which case they can be partially evaluated, or they consist only of a variable block and then their execution could be optimized using shallow backtracking.
- The most likely indexable arguments are the first two, higher arguments can be usually indexed only when a lower argument is indexable. Most procedures are indexable either on one or two arguments.
- In about 95% of procedures only a limited number of different type arguments is present: lists + nil, constants or functors. For such procedures we recommended new abstract instructions to be used, which, being more specialized, are more efficient than the WAM ones.
- In the first argument, which is the most important one, most procedures have no variable block, or they have a trailing variable block. Leading variable blocks are less frequent and internal variable blocks are rare. We have shown that for a majority of procedures an efficient and space-saving indexing scheme can be used.
- In the first argument, most constants are atoms, whereas in higher arguments the ratio of integer numbers is growing.
- Among the nonunit list blocks, indexing on the first element of the list can be used in 20% of procedures to restrict the number of matching clauses, in other 30%, it may restrict the number of matching clauses at least for some instantiantions of the call.

The SEPIA system (Meier et al., 1988) was implemented in accordance with the results of this paper. The indexing algorithm itself was fairly straightforward to write and its efficiency, in terms of both compilation and execution time is superior to current indexing schemes.

ACKNOWLEDGEMENTS

We thank to Herve Gallaire and Alexander Herold for reading and commenting previous versions of this paper. We further thank to the colleagues at ECRC who have provided us with many Prolog programs that have made this analysis possible, and we also appreciate the stimulating multi-national working environment at ECRC.

REFERENCES

(Bowen et al., 1986)
Kenneth A. Bowen, Kevin A. Buettner, Ilyas Cicekli and Andrew K. Turk.
The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler.
In *Third International Conference on Logic Programming*, pages 650-656. London, July, 1986.

(Carlsson, 1987)
Mats Carlsson.
Freeze, Indexing and Other Implementation Issues in the WAM.
In *Proceedings of the 4th ICLP*, pages 40-58. Melbourne, May, 1987.

(Meier, 1985)
Micha Meier.
Analysis of Prolog Procedures for Indexing Purposes.
Internal Report IR-LP-7, ECRC, August, 1985.

(Meier, 1986)
Micha Meier.
Shallow Backtracking in Prolog Programs.
Internal Report IR-LP-1113, ECRC, November, 1986.

(Meier et al., 1988)
M.Meier, P.Dufresne and D.Henry de Villeneuve.
SEPIA.
Technical Report TR-LP-36, ECRC, March, 1988.

(Robinson, 1965)
J.A.Robinson.
A Machine-oriented Logic Based on the Resolution Principle.
JACM 12(1):23-41, Januar, 1965.

(Turk, 1986)
Andrew K. Turk.
Compiler Optimizations for the WAM.
In *Third International Conference on Logic Programming*, pages 657-662. London, July, 1986.

(Van Roy, 1984)
Peter Van Roy.
A Prolog Compiler for the PLM.
Technical Report UCB/CSD 84/203, Computer Science Division, University of California, November, 1984.

(Venken, 1984)
Raf Venken.
A Prolog Meta-Interpreter for Partial Evaluation and its Application to Source to Source Transformation and Query-Optimisation.
In *Proceedings of ECAI*, pages 91-100. September, 1984.

(Warren, 1977)
David H. D. Warren.
IMPLEMENTING PROLOG - compiling predicate logic programs.
D.A.I. Research Report 39, University of Edinburgh, May, 1977.

(Warren, 1983)
David H. D. Warren.
An Abstract Prolog Instruction Set.
Technical Note 309, SRI, October, 1983.