11-1985

The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler

Kenneth A. Bowen
Syracuse University

Kevin A. Buettner

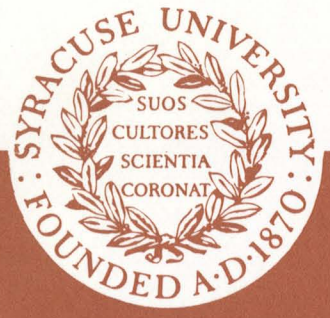
Ilyas Cicekli

Andrew Turk

Technical Report CIS-85-4
School of Computer & Information Science
Syracuse University

The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler

Kenneth A. Bowen, Kevin A. Buettner, Ilyas Cicekli, Andrew Turk



SCHOOL OF COMPUTER
AND INFORMATION SCIENCE
SYRACUSE UNIVERSITY

Technical Report CIS-85-4 November 1985
School of Computer & Information Science
Syracuse University

The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler

Kenneth A. Bowen, Kevin A. Buettner, Ilyas Cicekli, Andrew Turk

Logic Programming Research Group
School of Computer & Information Science
Syracuse University
Syracuse, NY, 13210 USA
CSNET: kabowen%syr

Abstract

The design and implementation of a relatively portable Prolog compiler achieving 12K LIPS on the standard benchmark is described. The compiler is incremental and uses decompilation to implement retract, clause, and listing, as well as support the needs of its four-port debugger. The system supports modules, garbage collection, database pointers, and a full range of built-ins.

1. Introduction

In the course of exploring implementation techniques for metalevel extensions of Prolog (cf. Bowen and Kowalski [1982], Bowen and Weinberg [1985], Bowen [1985]), it became apparent that a fast flexible Prolog compiler would be a useful tool to serve as a starting point for developing experimental implementations of the extended systems. Consequently, in late 1984 we began exploring just such a project. We planned to base the system on the designs of Warren [1983], implementing a byte-code interpreter for the abstract machine in C, while implementing the compiler itself in Prolog. We worked initially in C-Prolog on the Data General MV/8000 which was the machine available to us at that time. We were fortunate to join forces with the group working at Argonne National Laboratory (Tim Lindholm, Rusty Lusk, and Ross Overbeek) who were interested in the implementation of Prolog on multiprocessor machines. They had already implemented a byte-code interpreter for a system which would support multiple versions of Warren's abstract Prolog machine (WAM), different machines running on different processors, but using shared physical memory and implementing appropriate logical memory spaces. The system was parameterized as to the

This work supported in part by US Air Force grant AFOSR-82-0292 and by US Air Force contract F30602-81-C-0169. The authors are very grateful to the following people for numerous valuable conversations on the topics of this paper: Hamid Bacha, Aida Batarekh, Jim Kajiya, Kevin Larue, Jacob Levy, Tim Lindholm, Rusty Lusk, Jon Mills, Hidey Nakashima, Ross Overbeek, Karl Puder, and Toby Weinberg.

number of physical processors, so that we could run a version with that parameter set to one, yielding a sequential byte-code interpreter for the abstract machine. Thus, in principle, we could focus our efforts on the construction of the compiler. Naturally, life being what it is was not quite that simple. The Argonne group had implemented their byte-code interpreter in C on a VAX 780. While they had striven for portability, one serious hardware assumption had crept into the code, namely that the underlying machine was byte-addressable. Since the MV/8000 is not a byte-addressable machine, we found that we had to devote considerable energy to porting the Argonne WAM to the MV/8000. However, the changes necessary to achieve this port were propagated back into the original Argonne code, so that the present Argonne WAM is in all liklihood an extremely portable system. The Argonne system includes a "WAM assembler" which will assemble and load "WAM assembly code". (This was revised by Cicekli to remove limitations on the sizes of programs which could be assembled and loaded.) Thus we were able to hand-compile and run test examples. We were disappointed in the resulting performance, the naive reverse benchmark (nrev) performing at only about 4K LIPs. We concluded that the relatively slow speed was due to a combination of the portability requirements and the data structures necessary for multi-processor implementation (even though we were making no use of those facilities)! Performance improved somewhat when we moved to a newly acquired VAX 780 running Berkeley UNIX 4.2, but was still disappointing. This disappointment, coupled with an interest in implementing a Prolog system on 68000-based machines, led Turk to begin exploring a new implementation of a byte-code interpreter written in C, while as a group we continued work on the compiler.

The need to devote resources to the port to the MV/8000 had slowed our development of the compiler, so it was not until late February of 1985 that we had a first version of the compiler itself constructed and operational in C-Prolog. While writing the compiler in Prolog was of course a joy, we found ourselves somewhat hampered by C-Prolog's restricted memory size and apparent lack of significant tail recursion optimization and garbage collection. Consequently, we were forced to somewhat unnaturally segment parts of the compiler, store intermediate results in files, etc. The compiler itself had grown fairly large, reflecting our explorations of various optimization techniques. When we began to attempt to boot the compiler on itself, we were frustrated to discover that we immediately overran the maximum allowable local and global stack spaces. While we found that by a combination of breaking the compiler into many small files and using Prolog assert/retract hacks to reclaim stack space we could begin jamming it through, we were quite upset by the butchery this was performing on what we originally regarded as relatively clean code. At this time, Buettner had been devoting some time to exploring the implementation of a Prolog compiler on 16-bit machines, in particular the design of a byte-code interpreter for that environment. In a burst of enthusiasm, he roughed out a new byte-code interpreter for the abstract machine coupled with an implementation of a moderately sophisticated compiler, all written in C, in the space of a month. We now found ourselves in the (perhaps enviable) position of possessing three distinct implementa-

tions of the abstract machine (all written in C) and two compilers, one written in C and the other in Prolog.

While there were some differences in structure between the compilers, they both operated on basically the same principles. On the other hand, our two homegrown implementations of the abstract machine appeared to use significantly different techniques, and of course differed markedly from the Argonne implementation. Since both of our local WAMs executed nrev at better than 6K LIPS and both authors asserted that not all opportunities for optimization had been exploited, we decided to pursue development of both machines and compilers in parallel. In the course of the summer of 1985, we saw both machines evolve towards a more common structure, and begin achieving speeds in nearing 10K LIPS on nrev. We also had the interesting experience of booting the Prolog-based version of the compiler using the C-based Prolog compiler. We were able to do this without introducing any of the ugly adjustments we had found necessary when using C-Prolog. Since the two abstract machines seemed to be evolving towards a common structure, we decided in July (at a breakfast meeting at the Logic Programming Symposium) to coalesce the two efforts, making a final incorporation of the remaining clever techniques of Turk's machine into Buettner's. From that point on, we focused most of our efforts on developing the C-based Prolog compiler and abstract machine. We did complete the Prolog-based version of the compiler and delivered a copy to the Argonne group in late August. It is expected that this version will be made publicly available along with the Argonne WAM sometime in the near future. The rest of this paper will be devoted to describing the design, structure, and facilities of the C-based system.

2. Organization of the System

We will assume familiarity with Byrd, Pereira and Warren [1980], Pereira, Pereira, and Warren [1978], and Warren [1983]. To the user, our system presents the appearance of a standard Edinburgh-style interactive interpreter. However, it is really an incremental compiler. Thus we have no need to support a separate interpreter with all the difficulties of consistency between compiler and interpreter which are normally entailed. Briefly, the major services provided by the system are as follows:

- The compiler is resident in the system, incrementally compiling original and added program clauses (including those added by assert) as well as goals.

- Programs may be organized into modules which are relatively independent of file structure in that multiple modules may be included in a single file (a single module can also be spread over several files); visibility of procedures is controlled by use of import/export declarations; clauses not appearing within a module declaration are stored in a default global module; constants and functors are globally visible; modules may appear as submodules within oth-

er modules;

- Garbage compaction of the global stack (heap) and trail is provided using a pointer-reversal algorithm of Morris[1978]; no garbage collection is provided for the code space;

- Run-time use of retract, clause, and listing is accomplished via a general decompilation technology (described in detail in Buettner [1985]); this technology is also used to support the debugging subsystem;

- A full four-port debugging model (cf. Byrd[1980]) is provided; it relies on the decompilation technology mentioned above and accomplishes its task by constructing linked lists representing local stack frame entry and exit on the global stack (heap); it is largely complete, though some standard commands remain to be implemented;

- Database pointers are supported; these exist as Prolog terms which can occur in other terms and predicates;

The system supports the full range of built-ins standard in Edinburgh-style Prolog systems. Some are implemented in C, with the rest being written in Prolog and compiled by the system.

The system occupies approximately 135K bytes of virtual memory (and 76K bytes of physical memory) when loaded. Performance of the system on the naive reverse benchmark is shown in Table 2.1 (measured in LIPS) for lists of length 100 and 1000. The slower figures for lists of length 1000 of course reflects the need to perform garbage collection.

	Unoptimized	Optimized
100	9.6K	12.0K
1000	8.5K	10.5K

Table 2.1. Benchmark performance.

The "unoptimized" column represents the performance of the system running with the output of the UNIX 4.2 C compiler unchanged. The "optimized" column represents the performance of the system with the output of the C compiler slightly hand optimized. The only optimization specific to a Prolog system is a tightening of the dereference loop. All of the rest of the optimizations are of a generic sort that could be performed by a highly optimizing C compiler, such as shortening branches to branches (to branches...). Another such optimization involves reclaiming poorly used machine registers. In the compiler output, the low numbered machine registers are only used for scratch values and are not saved on procedure entry/exit. The usage of these registers was reorganized and

code added before calls and exits to render them safe. Most of these optimizations were performed on the code simulating abstract machine instructions. A native code compiler could get it right from the beginning, while of course performing many other optimizations. It would not surprise us to see a speed increase factor of 3-4 resulting from native code compilation.

3. Compiler Organization

While the principles on which the two compilers operate are quite similar, their internal organization is somewhat different.

3.2 Clause Compilation

The overall action of the Prolog-based compiler is divided into three major passes:

(1) compilation of individual clauses to intermediate code,
(2) organization of groups of intermediate clause code into procedures, and
(3) generation of instructions for the abstract machine.

During the first pass, the compiler treats each clause for a procedure separately, producing intermediate code representing the action of that clause. This pass is organized into three phases: lexical analysis, parsing, and intermediate clause code generation. The lexical analysis phase outputs a list of annotated tokens. The parsing phase processes this list, more or less in a definite clause grammar style, to produce a complex Prolog term representing the clause; a considerable amount of variable analysis is also performed during this phase. The third phase processes this term, producing another Prolog term representing the required sequence of abstract machine instructions. Considerable use of difference lists and uninstantiated logical variables representing machine addresses is made during these phases. During the second pass, the intermediate code for the individual clauses constituting a procedure is connected using the indexing instructions. Our method of indexing, which differs from Warren [1983], will be described later. The output of the second pass is a complex Prolog term representing the procedure. Consequently, assembly amounts to a traversal of this term, calculating symbolic addresses as necessary, and linearizing the entire structure; loading is then straight-forward.

The C-based version of the compiler utilizes a standard Prolog reader to read the clauses as terms. It makes one pass through the term, performing its variable analysis and building appropriate tables. On a second pass through the term, this compiler generates and loads the instructions for the clause, linking them into the naive try-me-else indexing chain for the procedure (see Section 3.2). Full indexing for the procedure is generated when the module containing the procedure is sealed.

Examination of the examples supplied in Warren [1983] shows that the required *get-* and *put-* instructions occur in the order corresponding to the left-to-right ordering of the corresponding terms in the source clause. In an effort to minimize the number of instructions generated and to optimize A-register usage, our compilers reorder these instructions. They also make a very serious attempt to set up the arguments to the first call in the body while carrying out the head matching. They also perform the now-standard Warren-style optimization of permanent variable allocation by trimming environments. (The permanent variables used in implementing cut are also included in this optimization -- cf. Section 4.)

3.2. Indexing

Access to the block of clauses constituting a procedure is handled in the usual way with hash tables, though provision for modules and hiding of local procedures complicates this a bit. Within the list of clauses constituting a procedure, it is desirable to minimize the number of clauses attempted but failed due to failure to match the head of the selected clause against the incoming goal. Such failure can occur only when the incoming goal contains instantiated variables; if all variables of the incoming goal are uninstantiated, the goal will match the head of each clause of the given procedure. Consequently, the indexing process has two major tasks to accomplish:

(a) When the incoming goal contains uninstantiated variables in designated indexing argument places, it must provide a means of trying each clause of the procedure in order.

(b) When the incoming goal contains instantiated terms in the argument places designated for indexing, it must provide a means of selecting only those clauses whose heads satisfy the following: for each argument position designated for indexing, the term occuring in the clause head must match the term occuring in the corresponding position of the incoming goal.

As with all other current Prolog systems known to us, ours only supports (or designates) indexing on the first argument of procedures. (However, our plans for the future include relaxing this restriction.) We have not modified the indexing instructions of Warren [1983], but we do employ them in a different manner. Focusing on the first argument of procedures, a *block* of clauses is a maximal subset of the clauses for a procedure, contiguous in the given clause ordering, all of whose first head arguments are of the same type, where the allowable types are:

constant, compound term (other than list), variable, and list.

Roughly, one uses indexing instructions at the lowest level to control access to each block, coupling these with second-level indexing instructions to control transfers between blocks. A sequence of instructions of the form

try - retry - ... - retry - trust -

specifies a group of clauses to be tried in sequence, as does a sequence of the form

try_me_else - ... retry_me_else - ... retry_me_else - ... trust_me_else fail.

In the second case, the branch instructions must be physically interleaved with the code of the individual clauses, while in the first, the collection of branch instructions can be physically quite removed from the code of the clauses controlled. We refer to these as *try chains* and *try_me_else chains*, respectively. Note that in a try_me_else chain, the label of each instruction is the address of the succeeding retry_me_else or trust instruction. Consequently, this succeeding instruction and its following clause code need not physically follow the code of the preceding clause. Consequently, we can regard a try_me_else chain as a linked list of clauses. In the case of try chains, while the actual try-retry-trust instructions must physically follow one another (they constitute a vector of instructions), the actual code blocks of the clauses they control can be distributed in memory in any manner whatsoever. These code blocks need bear no physical relationship to one another nor to the controlling try chain, other than the fact that the try chain instructions reference the addresses of the clause code blocks. We exploit both of these observations in the implementation of assert and retract. Our method of indexing runs as follows. To cater to requirement (a) above, we create one master try_me_else chain linking all of the clauses of the procedure. In catering to requirement (b), we avoid using the ..._me_else instructions, restricting ourselves to try-retry-trust to control sequential access to both clauses and blocks. Constant and compound term blocks are of course accessed using switch instructions, and overall access to the upper-level indexing is initiated with the switch_on_term instruction. Sequential ordering of groups of clauses as well as groups of blocks of clauses is indicated with try chains; no use of try_me_else chains is made in the upper-level indexing meeting requirement (b). Consequently, the indexing meeting requirement (a) is totally separated from the indexing meeting requirement (b). We feel this provides great flexibility for insertion and deletion of clauses (by assert/retract or by a run-time editor) while minimizing the number of choice points which must be created. Figures 3.1 and 3.2 schematically indicate the structure of this scheme.

4. Abstract Machine Organization and Cut

The layout of the various machine regions is shown in Figure 4.1.

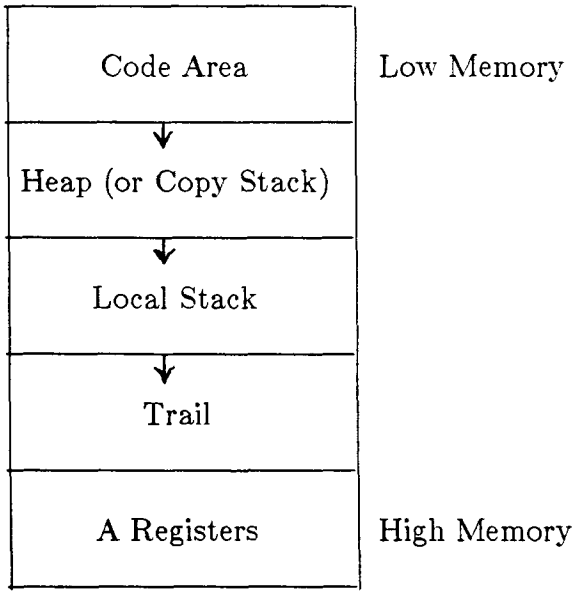


Figure 4.1. Abstract Machine Organization.

A bit table for garbage collection is also permanently allocated. For the most part, we have implemented the instruction set of Warren [1983] with only minor modifications. The most significant extension to date is the addition of a new machine register (called *cutpt*) and new instructions to allow us to compile cut. These instructions and their effects are listed below:

Instruction	Action
set_B_from_cutpt	B := cutpt
set_B_from Yn	B := Yn
save_cutpt_in Yn	Yn := cutpt
save_B_in Yn	Yn := B

Figure 4.2. Instructions Necessary for Cut.

The last instruction is only necessary for compiling the so-called "soft cut".

The difficulty in dealing with cut is that at compile time, it is impossible to know how many choice points will be created for a procedure before a clause of that procedure is entered. Consider the following trivial program.

```
f(a).
f(b).

f/1:    switch_on_term    C1a,L1,fail,fail
```

```
L1:     switch_on_constant  2,[a:C1, b:C2]

C1a:    try_me_else         C2a          % f(
C1:     get_constant        a,A0         %  a)
        proceed                          %  .

C2a:    trust_me_else       fail         % f(
C2:     get_constant        b,A0         %  b)
        proceed                          %  .
```

When the first clause C1 is executed, there can be one or zero choice points for the procedure f/1, but this cannot be detected at compile time because it depends on the incoming value in the first argument register A0. If the incoming value in A0 is the constant *a*, there will be no choice point created for the procedure f/1, but if *a* is an unbound variable, there will be one choice point created for the procedure f/1.

The new register *cutpt* is treated in the abstract machine as follows. The value of the last choice point register B is automatically stored in the cutpt register by a *call* or an *execute* instruction to record the address of the last choice point before the procedure is invoked. The current value of the cutpt register is saved in a choice point when the latter is created. The cutpt register is reset from the value stored in the last choice point when backtracking occurs.

The following examples illustrate how the compiler uses these instructions to compile cuts.

Example 1.

```
p :- q1, !, q2.
```

Code for the clause:

```
allocate            1
save_cutpt_in       Y0
call                q1/0,1
set_B_from          Y0
deallocate
execute             q2/0
```

Example 2.

```
p :- !.
```

Code for the clause:

```
set_B_from_cutpt
proceed
```

Notice that the clause doesn't have an environment and that the cutpt register contains a pointer to the last choice point before the procedure p is invoked.

Example 3.

```
p :- q1, q2, !.
```

Code for the clause:

```
allocate            1
save_cutpt_in       Y0
call                q1 ,1
call                q2 ,1
set_B_from          Y0
deallocate
proceed
```

This approach can be optimized.

5. Conclusions

The abstract machine design of Warren [1983] together with the compilation techniques suggested by his examples are a sound piece of software engineering. We have filled in some gaps such as the implementation of cut which were omitted in his discussion, and have introduced modifications in the pursuit of refining and optimizing performance. The present system provides an excellent basis for our primary goal, the pursuit of implementations of meta-level Prolog systems. Our approach will be to introduce modifications to the abstract machine providing the required functionality, the primary one being a change in the treatment of the code space. This will be coupled with appropriate changes in the compilers. We expect this to lead to efficient implementations of the experimental systems.

6. References

Bowen, K.A., and Kowalski, R.A., Amalgamating language and metalanguage in logic programming, in *Logic Programming*, ed. K. Clark and S.-A. Tarnlund, 1982, pp 153-172.

Bowen, K.A., and Weinberg, T., A meta-level extension of Prolog, *1985 Symposium on Logic Programming*, Boston, IEEE, 1985, pp. 48-53.

Bowen, K.A., Meta-Level programming and knowledge representation, *New Generation Computing*, 3, 1985, pp. 359-383.

Buettner, K.A., Decompilation of compiler Prolog clauses, submitted.

Byrd, L., Prolog debugging facilities, in Byrd, Pereira, and Warren, 1980.

Byrd, L., Pereira, F., and Warren, D., *A Guide to Version 3 of DEC-10 PROLOG*, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1980.

Morris, F.L., A time- and space-efficient garbage collection algorithm, *Communications of the ACM*, 21, (1978), pp. 662-665.

Pereira, L.M., Pereira, F.C., and Warren, D.H.D., *User's Guide to DECsystem-10 PROLOG*, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1978.

Warren, D.H.D., An abstract Prolog instruction set, SRI Technical Report, 1983.

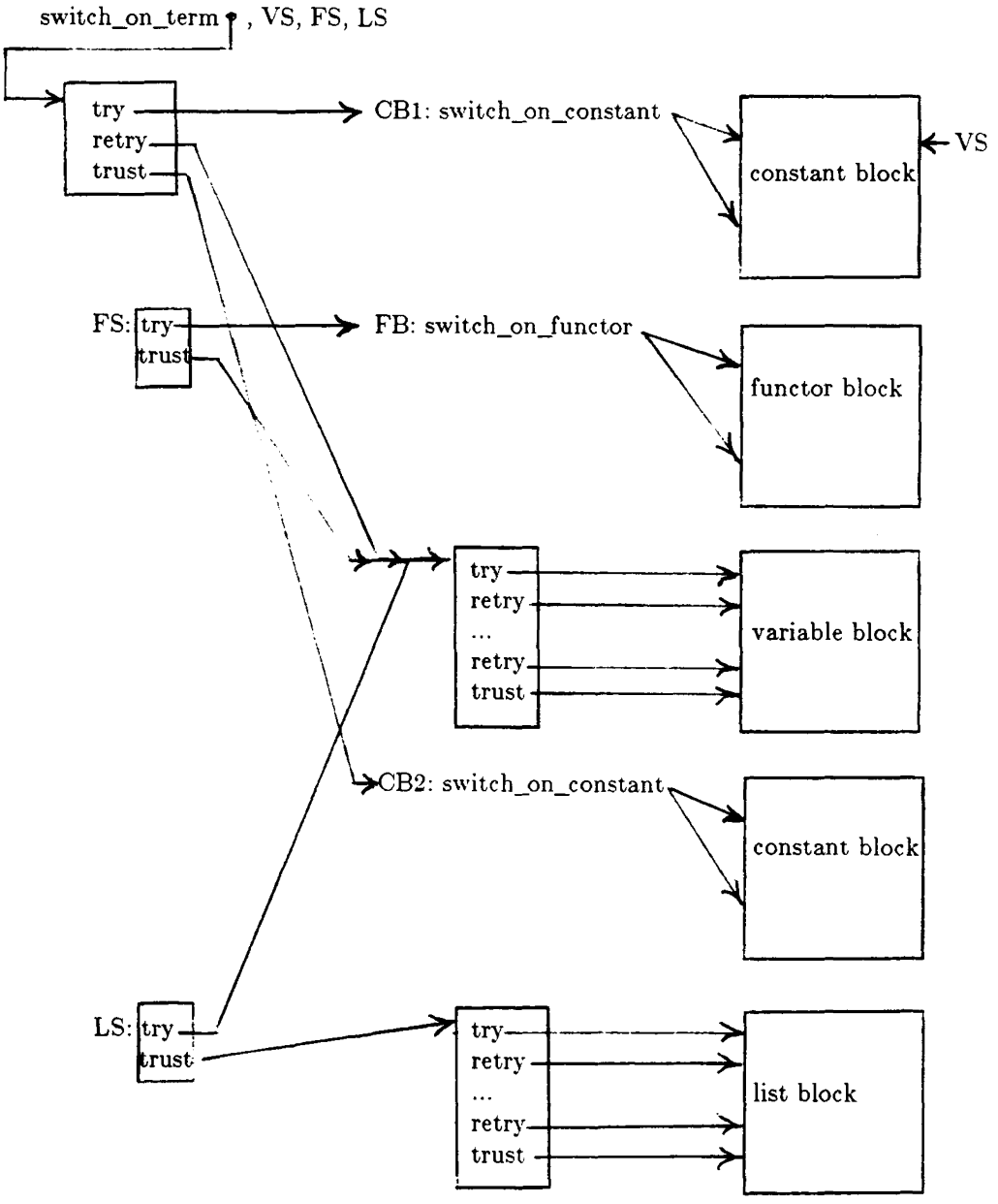


Figure 3.1. Overall indexing structure.

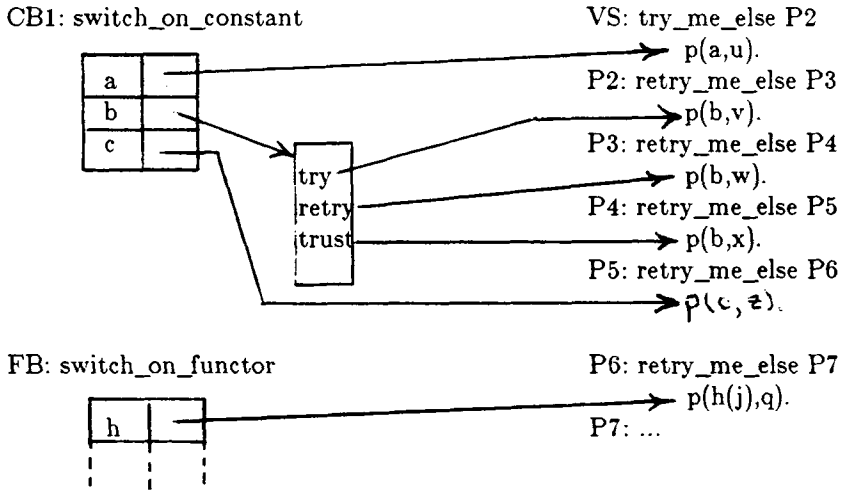


Figure 3.2. Detail of indexing structure.