LOGIC PROGRAMMING ARCHITECTURES

H. Benker

Rapporteur: Dr. R.P. Hopkins

1. The Programming Language Prolog

Today logic programming is mainly represented by the programming language Prolog ([Clocksin 84, Bratko 86, Sterling 86]) and its many dialects and implementations. Therefore this paper concentrates on the implementation of Prolog as the most important representative of logic programming languages.

The procedural interpretation of a Prolog program allows its execution on current computers. In order to analyse Prolog as a procedural language it is necessary to relate important terms of logic to their corresponding terms describing procedural programs:

- predicate -- procedure
- head of a clause -- procedure declaration
- goal -- procedure call

2. The Problems of Symbolic Computation

Conventional procedural languages are much easier to implement than Prolog. In order to implement Prolog more efficiently it is therefore necessary to anlalyse what is new in Prolog. It actually turns out that these new features of Prolog are problems of symbolic computation in general. Prolog shares many of its implementation problems with other symbolic languages, such as e.g. Lisp.

2.1. Prolog is a non-typed Language

The Prolog programmer never specifies the type of a variable. Therefore Prolog procedures (predicates) can be called with any kind of objects as arguments. The type of an object is not known at compile-time. The type of objects has to be determined dynamically, i.e. at run-time. An arithmetic operation for example first has to test if its arguments are integers, reals or any other kind of object. Only after this test can the corresponding operation be performed.

Typical types of objects that have to be treated by a symbolic language are:

- **Numbers** - either integers or floating point
- **Atoms** - symbolic constants created by the user or the program itself. In Prolog they are represented as strings starting with a lower-case letter. Examples: robert, at2096, this_is_an_atom
- **Structures** - compound terms defined by their functor and arity and the given arguments. Examples: structure(argument1,a), foo(a,b,c)

- **Lists** - Lists are a specific kind of structure. They are however so important to symbolic processing that a special syntax and implementation is used. Using lists chains of objects can be created. A list consists of two parts: the head (also called CAR) and the tail (also called CDR). The head holds the first element of the list, the tail the remaining list. The empty list is represented by the atom nil (Prolog syntax: []). Examples: [a,b,c], [[c,d],e]

- **Variables** - Variables in Prolog are represented by a string beginning with a capital letter or a "_". They are so-called "logic variables". To understand their nature it is necessary to know about unification. Therefore logic variables will be fully introduced in a later section. Examples: X, This_is_a_variable, _x25

2.2. Unification

Unification is the basic procedure calling and argument passing mechanism in Prolog. It checks if two terms are equal or can be made equal by binding variables. For more detail on unification the reader is referred to a textbook on Prolog (e.g. [Clocksin 84, Bratko 86, Sterling 86]).

2.3. The Logic Variable

The character of a logic variable is defined by unification. A value can be assigned to a logic variable only by unification.

Logic variables can be bound or not, i.e. they have a *free* or *unbound* state. Once a variable is bound to a value, unification does not differentiate between a constant and the bound variable. Therefore it simply fails if a variable is bound to a term which does not match. It cannot unbind a variable. Logic variables are therefore single assignment variables.

This is in contrast to variables in conventional procedural languages. They always have a value and can change this value several times in their lifetime.

Another characteristic of logic variables is that they can be bound to each other. Example: if two unbound variables X and Y are unified, none of them gets bound to a value. If however now one of them gets bound to a value, the other one also will be bound to that same value.

2.4. I/O Modes

Procedure declarations in conventional procedural languages define which of the arguments to the procedure are input and which are output. This is not so in Prolog. A Prolog procedure can be called with any argument instantiated or not.

2.5. Recursion

Recursion is the only language construct in Prolog which allows repetetive execution of parts of a program. The programmer cannot specify that it is possible to solve a problem iteratively.

An efficient compiler therefore needs to transform the recursively specified problem into a program using iteration.

2.6. Non-determinism

A Prolog predicate consists of several alternative clauses. The clauses are tried in textual order.

Before a clause is tried, the current state of the computation has to be saved. A pointer to the next alternative has to be memorized.

2.7. Further Features of Symbolic Languages

Symbolic languages usually do not differentiate between code and date. All data can be treated as program and executed.

There are no global variables in Prolog. (In Lisp they were introduced for efficiency, although they are outside the concept of the language.)

The dynamic allocation of memory makes it necessary to introduce garbage collection mechanisms.

In the near future we expect active data structures (e.g. to implement constraint networks) to become more and more important. They are not yet very popular and today only a few systems implement some kind of active data structures.

Very powerful and comfortable programming environments became popular with Lisp. Future implementations of symbolic languages also need to give this kind of support to the user. This implies that it is necessary to implement mixed execution of compiled and interpreted code as well as incremental compilation.

3. Introduction to existing Prolog Systems - The WAM

Most Prolog compilers offered today use an intermediate level of compilation. Nearly all compilers use an intermediate language which is based on the definition of an abstract machine by D. H. D. Warren [Warren 83]. This Warren's Abstract Machine, or WAM for short, is also the basis for most experiments implementing Prolog in hardware. It is necessary to know its basic mechanisms in order to understand how architectural changes to processors can boost Prolog performance.

3.1. Data Formats

Data objects in the WAM hold two fields: the tag and the value fields. Both fields are kept in one machine word if possible. This is a very compact and efficient representation of Prolog objects.

3.2. Memory Organisation

Memory areas:

- **code** - compiled program
- **global stack** - lists and structures created during execution
- **local stack** - environment and choice point stack
- **trail stack** - pointers to variables that need to be unbound on backtracking

Special registers:

- P and CP for the code area
- H and HB for the global stack
- E, CE, and B for the environment stack
- TR for the trail stack
- Ax, argument registers for parameter passing

3.3. Abstract Instruction Set

PUT instructions: put arguments to a procedure into the argument registers

Procedural instructions: procedure call and return instructions; allocation of environments (use of TRO)

Indexing instructions: Selection of clauses and creation of choice points. The branch on the type of the first argument avoids the creation of unnecessary choice points. (*switch*-instructions)

Unification instructions: get and unify instructions

3.4. Possible Optimizations

Creating environments as late as possible avoids unnecessary work in case of failure.

Using shallow backtracking, the next alternative can be tried without restoration of all registers if the unification of the head fails.

The creation of a choice point can be delayed until the call of the first goal in a clause. It might sometimes not be necessary at all.

4. Hardware for Prolog Execution

Advanced features of modern general purpose processors are most useful for the execution of Prolog. A special processor can only be better than a general purpose processor if it can also execute basic instructions as fast as the general purpose processor. The top level organisation of a Prolog processor looks much like a modern general purpose processor.

A large physical memory with high throughput is required. The treatment of symbolic programs is very memory consuming.

The abstract machine defines many state registers to manage the different memory areas. Ideally all these abstract registers are implemented as fast hardware registers.

Unification requires switching on the type of an object. Therefore the data format must include a tag field. The fastest method to do a switch on a type is a multi-way branch in microcode. Therefore CISC architectures have advantages over RISC designs.

Every object in Prolog is accessed via a reference chain. A fast dereferencing mechanism has to be provided.

Some Prolog instructions can be used in different execution modes. The decoding should take this into account and treat different execution modes of the same instruction just like different instructions.

When a variable is bound, three magnitude comparisons have to be performed in order to determine if the variable has to be trailed or not. Special hardware can do this in parallel with the binding.

It is not possible to hold variables in registers. They need to be transformed into references when put into registers. Again special hardware can be provided to perform this operation very quickly.

5. How to use a Prolog Processor - System Architectures

Numeric co-processors execute single instructions of a program for the main CPU. A Prolog program needs to be treated entirely by the co-processor.

Array processors take all input data and produce the result in batch-like manner. Prolog co-processors have to cope with a large amount of data. They produce random accesses to a large data base. Prolog programs are non-deterministic and it therefore cannot be guaranteed that the internal state of the co-processor will not be needed again after the delivery of a result.

Common processing of a Prolog program between the host machine and the co-processor requires transferring the machine state from one machine to the other if one takes over. This makes shared memory co-processors unattractive. Also for bus load and cache consistency problems it seems to be more interesting to use co-processors with their own memory system.

A backend processor is capable of treating an entire problem on its own, without help from the host. It processes programs one after the other in a batch operating mode. It can, for example, sit on a LAN and be used by several workstations. Current applications, however, demand frequent interaction between host and Prolog processor and the handling of non-determinism is difficult.

A stand-alone AI workstation executing Prolog has advantages because it is freed from all the constraints of a host machine. It is however difficult to integrate with an existing computing environment. Also the hardware and software effort to produce an entire system is enormous.

6. Prolog Machines

Performance of Prolog systems is measured in Logical Inferences per Second. One inference, as seen from the logical semantics of Prolog, corresponds to one execution of a procedure in the procedural semantics. Peak performances are obtained with small, deterministic programs.

The degradation from peak to average performance is much worse with general purpose processors than with specialised Prolog processors.

Some Prolog architectures are presented. And their main characteristics described. The interested reader is referred to the literature for details on the architectures.

The ICM3 [Noye and Syre 87] project was the study of a co-processor using the shared memory approach. The project was carried out down to chip-level simulation. A software environment including an optimising compiler is operational.

Nothing has been published about the ICM4 study. It is a backend processor, thus having its own memory. The memory design was such that very fast dynamic RAMs were used in a serial mode in order to avoid the need for caches. The execution unit was based on a two-stage

pipeline. The pipeline performed very well, for the same program as ICM3 around the same number of cycles were used, i.e. the gain in cycle time could be fully exploited. The instruction set was a mix between RISC-like instructions which made all pipeline stages visible to the compiler, and high-level microcoded instructions for unification. The problem with this design is that the RAM chips became not available (announced in 1985, still no samples available). Also the hardware overhead to restore the complex pipeline stages after an interrupt is quite significant. This concerns the pipeline registers in the execution unit, but even more the ones in the memory interface.

The new personal AI workstation designed at ICOT [Nakashima and Nakajima 87] is a fairly conventional machine, but uses many optimizations of the WAM. Therefore it has quite a good performance and the most remarkable thing about it is its full system capability.

One of the early designs of a Prolog machine using the WAM was done at the University of California, Berkeley [Dobry et al. 84, Dobry et al. 85]. The machine was named PLM. It is the design of a shared memory co-processor using a three stage pipeline in the execution unit.

Another very interesting study done at the same university is the SPUR processor [Hill et al. 86, Patterson 87]. It is a RISC chip set designed for the execution of Lisp. The implementation of Prolog on this machine has been studied [Borriello et al. 87]. The comparison with the CISC architecture of PLM shows that SPUR needs around 15 times more instructions and runs at half the speed of PLM. In the opinion of the SPUR team this comparison is in favour of the SPUR architecture, because its hardware is much simpler. On the other hand it also shows clearly the limitations of a RISC approach towards Prolog (in particular the lack of a multi-way branch facility for unification is a significant disadvantage of a RISC).

All the machines introduced above have three things in common: they use the WAM as a base, have special hardware for tag handling, and special attention is given to the design of the memory system. A performance evaluation of these machines shows that for big programs they have a significant advantage over general purpose processors.

The most significant difference between a general purpose machine and a specialised AI processor lies in the use of tag hardware. This is also the most costly addition to general purpose processors. All the other features for Prolog are rather inexpensive (in terms of hardware complexity) little changes to a general purpose machine.

The performance evaluation gave ECRC enough confidence to start a new project aiming at the hardware implementation of a Prolog processor. The characteristics of the Knowledge Crunching Machine KCM are : private memory co-processor; code and data cache; memory management system; 64 bit word length; synchronous prefetch unit.

References

[Abe et al. 87] S. Abe, T. Bandoh, S. Yamaguchi, K. Kurosawa, and K. Kiriyama.
High Performance Integrated Prolog Processor IPP.
In The IEEE Computer Society Press (editor), *The 14th Annual International Symposium on Computer Architecture*, pages 100 - 107. IEEE/ACM, June, 1987.

[Borriello et al. 87]
Gaetano Borriello, Andrew R. Cherenson, Peter B. Danzig and Michael N. Nelson.
RISC vs. CISCs for Prolog: A Case Study.
1987.
To appear in ASPLOS-II 1987.

[Bratko 86] Ivan Bratko.
Prolog Programming for Artificial Intelligence.
Addison-Wesley Publishers, 1986.

[Clocksin 84] W. F. Clocksin and C. S. Mellish.
Programming in Prolog.
Springer Verlag, 1984.

[Dobry et al. 84]
T.P. Dobry, Y.N. Patt, A.M. Despain.
Design decisions influencing the microarchitecture for a prolog machine.
In ACM Sigmicro Newsletters (editor), *MICRO-17*, pages 217-231. U. California Berkeley, New Orleans, 1984.

[Dobry et al. 85]
T.P. Dobry, A.M. Despain and Y.N. Patt.
Performance Studies of a Prolog Machine Architecture.
In IEEE Computer Society Press (editor), *The 12th Annual International Symposium on Computer Architecture*, pages 180 - 190. IEEE/ACM, 1109 Spring Street - Suite 300 - Silver Spring, MD 20910, June, 1985.

[Hill et al. 86] M. Hill, S. Eggers, J. Larus, G. Taylor, G. Adams, B.K. Bose, G. Gibson, P. Hansen, J. Keller, S. Kong, C. Lee, D. Lee, J. Pendleton, S. Ritchie, D. Wood, B. Zorn, p. Hilfinger, D. Hodges, R. Kerz, J. Ousterhout and D. Patterson.
Design Decisions in SPUR.
Computer :8-22, November, 1986.

[Nakashima and Nakajima 87]
H. Nakashima, K. Nakajima.
Hardware architecture of the sequential inference machine PSI II.
1987.
ICOT submitted to SLP87.

[Nakazaki et al. 85]
R. Nakazaki, A. Konagaya, S. Habata, H. Shimazu, M. Umemura, M. Yamamoto, M. Yokota and T. Chikayama.
Design of a High-speed Prolog Machine (HPM).
In IEEE Computer Society Press (editor), *The 12th Annual International Symposium on Computer Architecture*, pages 191 - 197. IEEE/ACM, 1109 Spring Street - Suite 300 - Silver Spring, MD 20910, June, 1985.

[Noye and Syre 87]
Jacques Noye and Jean-Claude Syre.
ICM3: Design and evaluation of an Inference Crunching Machine.
1987.
To be published in the proceedings of the 5th International Workshop on Database Machines.

[Patterson 87] Dave Patterson.
A Progress Report on SPUR: February 1, 1987.
Computer Architecture News 15(1):15-21, March, 1987.

[Sterling 86] L. Sterling and E. Shapiro.
The Art of Prolog.
MIT Press, 1986.

[Tick 84a] Evan Tick.
Towards a multiple pipelined prolog processor.
In U. of Maryland (editor), *Int'l workshop on high level computer architecture*, pages 4.7-4.17. Stanford Lab, Los Angeles, May, 1984.

[Tick 84b] Evan Tick.
Sequential prolog machine: image and host architecture.
In ACM Sigmicro Newsletters (editor), *MICRO-17*, pages 204-216. Stanford lab, New Orleans, December, 1984.

[Tick and Warren 84]
E. Tick and D.H.D Warren.
Towards a pipelined prolog processor.
In IEEE (editor), *1984 International Symposium on logic programming*, pages 29-40. IEEE, Atlantic City, February, 1984.

[Warren 83] David H. D. Warren.
An abstract prolog instruction set.
Technical Report tn309, SRI, October, 1983.

DISCUSSION

Dr. Gries made the observation that indirect reference chains of indefinite length were not unique to logic programing implementations, as had been suggested in the lecture, but were also required for example in the implementation of ALGOL68.

Professor Randell asked whether incremental compilation, which was an important part of the architecture presented, was also included in the Warren Abstract Machine ("WAN") for PROLOG execution. **Mr. Benker** replied that the WAM was concerned with providing an intermediate level for the efficient execution of the basic PROLOG mechanisms and thus did not address issues such as incremental compilation. Then **Professor Randell** also raised the question of parallel addressing of tags as a natural mechanism for improving the performance of this kind of architecture. In response to this **Mr. Benker** commented that such mechanisms require special purpose hardware, whereas the architecture being presented is specifically intended to employ "off-the-shelf" components. Given that the PROLOG machine components and the conventional host machine with which it interfaces both employ a standard 32-bit word length, the two alternatives in implementing tags are: (i) to store the tag of a 32-bit data item in another (adjacent) complete 32-bit word, thus degrading performance by requiring two memory cycles to access a single data item; or (ii) to use some of the bits from a data item word to store its tag, thus reducing the number of bits for the data value and so comprimising data compatibility with the host machine.

DISCUSSION

During the discussion **Professor Randell** raised a number of questions concerning the performance measures for a PROLOG machine and the comparison of its performance with both other PROLOG machines and conventional machines. In these questions and **Mr. Benker**'s replies, the following points were made. There are commonly accepted performance "benchmarks" for PROLOG machines. However these are somewhat inadequate in employing very simple programs, such as appending an item to a list or reversing a list, which under-utilise the specialised mechanisms of a logic machine wherein lie its potential benefits for more sophisticated programs. The resulting performance comparisions have the same limited degree of significance as for using programs with few procedure calls in comparing machines all optimised for efficient procedure calling. Also the "Logical Inferences Per Second" performance measure used is misleading since the amount of computation required for a single "Logical Inference" in these simple programs is very much less than would be required for a single "Logical Inference" in more sophisticated programs. Another problem in comparing performances is in factoring out the effects of different implementation technologies used in different implementations, such information not having been published for many of the machines being compared. In fact, most performance results for these machine designs are obtained from simulation studies or hardware implementations using TTL technology. The necessity for experimental implementations of novel special purpose architectures to use such "outdated" technology makes it difficult to justify the fuller development of such architectures since in evaluating their performance against the conventional architectures with which they would have to compete, the latter have the architecturally irrelevant advantage of implementations using production quality custom chips. Despite this, it can be hoped that in the long term the work on special purpose architectures will make a significant impact on the commercial market. To acheive this it is increasingly important to integrate these novel architectures into the existing computing environment as co-processors to be hosted by conventional machines. One particular commercial product that could arise from novel architecture research would be microprocessors handling tagged data items as this is a requirement common to the efficient realisation of most such architectures.

In response to a question from **Dr. Gries** concerning which aspects of PROLOG execution would benefit from special purpose hardware, **Mr. Benker** mentioned tag handling and also garbage collection. The latter is currently being looked into by a group at the European Computer Industry Research Centre (ERCC), but nothing has yet been implemented. **Professor Randell** raised the relationship between PROLOG machines and the anticipated future use of very large "knowledge" databases. The approach proposed by **Mr. Benker** is that the data base and PROLOG machine would both be co-processors for a conventional host machine. There is a group at ERCC investigating the interactions between such co-processors and how tightly coupled their connection needs to be. The question of the handling of Input/Output within a PROLOG system was raised by **Dr. Schneider. Mr. Benker** agreed that this was a problematic area, emphasising the importance of off-loading input/output work from the PROLOG processor by the use of an input/.output co-processor.

Finally, **Professor Randell** introduced the question of the extent to which PROLOG programs exhibit exploitable parallelism, particularly in view of the initial emphasis in the Japanese Fifth Generation Project on the parallel execution of logic programs. **Mr. Benker** made the observation that before a parallel machine could become a commercial possibility there must be a good performamce sequential machine since the former will substantially be a multiplicity of the latter. However for research purposes it is not unreasonable to

build a parallel machine based on a sequential machine of poor performance. The source of potential parallelism for logic programs is in the parallel execution of the alternative clauses for a goal (OR parallelisn) and/or parallel execution of the sub-goals of a particular clause (AND parallelism). Most current logic programming implementations are sequential, partly because programmers are trained in sequential paradigms. He also commented that the examples, such as solving the eight queens problem, usually used to illustrate the potential for parallelism are not of much practical significance, and in fact it prooves difficult to find useful applications of parallelism.