84

Methodology of Logic Programming

by

Ehud Shapiro

Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, ISRAEL

In this session I would like to discuss research methodology, rather than programming methodology, of logic programming. As a basis for discussion I propose the following statements, interspersed with text that attempts to justify or explain them.

1 Goals of research in logic programming

Statement: *Logic-programming share the goals of computer science at large.*

As I see it, there are no major differences between the goals of computer science at large and the goals of logic-programming. Both want to solve the problem:

(*) How to make computers do what we want them to?

This problem has two derivatives:

(1) How to make it easy for us to make computers do what we want them to?
(2) How to make computers do fast what we want them to?

Using a more respectable jargon, the two derivative questions become:

(1') How to program computers?
(2') How to make computers run the programs fast?

Many of us beleive that logic-programming may provide better solutions to these problems than the more conventional approaches to computer science.

Statement: *The basic method of computer science is bootstrapping.*

Computer science offers one encompassing methodology for solving these two questions, namely bootstrapping. Apart from brilliant new ideas (which no methodology can promise to provide), the crucial factor dictating the ease in which we can program computers and build faster computers is the computerized support available to these tasks. More concretely, the ease of programming is determined mostly by the quality of the programming environment available; and the possibility of building cheaper and faster computers is determined mostly by the quality of the CAD/CAM systems available.

Logic programming has a lot to contribute to the general thrust of bootstrapping, and also to provide some briliant new ideas of its own.

2 Prolog

Statement: *Prolog is an expressive and efficient programming language, which we are still learning how to use.*

Prolog is the first practical logic programming language. It is acquiring a growing group of users, who develop for it a rich set of programming idioms and techniques, and a refined programming style. Inspite of the initial dissatisfaction of Prolog's inventor, Alain Colmerauer, and others with the language, it turns out that its expressiveness is far greater than what was expected. It is surprising that such a simple language can lend itself to so many sophisticated and powerful programming techniques. Even after programming in Prolog for the past three years, Iam still learning new methods and techniques of Prolog programming.

Prolog falls short of the aspirations of the founders of logic programming in several respects: it has a rather inflexible control, and, to implement substantial systems, must resort to features that have only procedural meaning (cut, I/O, side-effects). Nevertheless, many of us beleive that Prolog, as it is, is a good programming language for many applications. To increase its effectiveness, Prolog requires improvements in its speed of execution and programming environment.

Statement: ***Prolog should be made to run faster.***

The desire for greater speed needs no justification. If we had as many MegaLIPS as we have MIPS, then almost no programming task will have to be carried in a lower programming language.

The speed of Prolog on a von Neumann machine can be increased in several ways. One is to improve the basic cycle of Prolog, the unification, by providing faster (cached, pipelined, multiported) memory access, by supporting the basic unification instructions in hardware or microcode, and by parallelizing the unification of subterms. Another is to incorporate in Prolog an abstract data-types mechanism, that will support in a logical way interface to efficient data-structures. Abstract data-types are needed in order to make more efficient use of the resources of the underlying hardware. If substantial systems are to be implemented in Prolog, then, given current technology, we cannot afford to represent everything as a list of elements. Mutable arrays, strings, and other efficient data-structures need to be supported. The only clean way to support these in the logic-programming framework is via predicates over abstract data-types.

Improving Prolog's speed by incorporating high-level parallelism (in contrast to the low-level parallelism available in the unification algorithm), is discussed in the next section.

Statement: ***Prolog needs a better programming environment. Prolog programming environments are best implemented in Prolog.***

Given the short time they exist, and the number of man-years devoted to their development, some of the current Prolog programming environments are quite impressive. The main problem that prevents further, or faster, development, is that every new Prolog implementor reinvents the wheel. This phenomenon is most evident in the development of the Edinburgh Prolog family, in which every new implementator has implemented the programming environment from scratch.

One of the most important properties of Prolog is that it is an excellent language for developing its own environment. By defining a small core-Prolog, which is expressive enough to implement a full environment, a new Prolog implementor can simply implement this core, and port the environment from a previous implementation. The availability of such a core will also support distributed implementation efforts, in which different tools are implemented in different locations. This is in contrast to the current situation, in which the burdon of implementing a reasonable environment for Prolog falls solely on the implementor of the core Prolog and its close associates.

My experience suggests that a subset of the system predicates of Waterloo Prolog, or those of Edinburgh Prolog augmented with the 'retry' and 'ancestor-cut' predicates, are expressive enough to implement almost any tool desired. Many system predicates in Edinburgh Prolog are better viewed as utilities that are, in principle, implementable in core-Prolog, but are provided for the sake of convenience or efficiency.

Statement: ***Prolog should be kept a small.***

There are two good reasons for keeping the core of Prolog small. One is intellectual economy. I think we are still learning how to program in Prolog. A baroque set of features (cf. IC-Prolog[4]) will prevent us from identifying what is essential and what is superfluous, and will not encourage the development of innovative programming techniques that squeeze every ounce of expressiveness from a small set of constructs.

Another good reason to keep Prolog small is related to the discussion of programming environments and bootstrapping. If there is a small Prolog core, in which everything else is implemented, then:

1. Developing a new or better Prolog requires less effort.
2. A new Prolog implementation does not need a new environment.
3. Sophisticated Prolog programming tools, that know about all core-Prolog system predicates, are easier to develop.

An example of an enhacement to Prolog that can be implemented in core-Prolog is a module and type system. Most current Prolog implementations resemble an assembly language, rather than a high-level programming language, in their flat name space of procedures, and in their lack of support of any type system. It is clear that a facility for modular programming is necessary for substantial systems to be developed in Prolog by many programmers. MProlog [14] supports a notion of modules. However, its implementation is in a low-level language, and cannot be ported to other Prologs. On the other hand, [5] showed that modules can be implemented easily in Prolog, by preprocessing, without affecting the Prolog core. Since the preprocessor is written in Prolog, it can be ported, in principle, to any compatible Prolog implementation.

Another example is the Prolog-10 debugger [1]. It is implemented almost solely in Prolog, but pieces of it which are implemented in a lower level language (pseudo-Prolog), prevent it from being easily ported to a new, compatible, Prolog implementation, such as CProlog. On the other hand, the debugging algorithms in [12] are implemented solely in Prolog.

Inspite of what is said, investigating extensions to Prolog is still a useful activity. I beleive that any extension to Prolog's core should satisfy at least the following three criteria:

1. It can be demonstrated, with non-toy examples, that the extension is useful.
2. The extension cannot be implemented in Prolog (e.g by preprocessing).
3. The extension can be implemented efficiently, and does not incure runtime overhead when not used.

Examples of extensions that can be implemented in Prolog are second order predicates (setof, bagof) [15] and modules [5]. Examples of extensions whose implementation seem to induce run-time overhead even when not used are selective backtracking [10] and several other forms of more sophisticated control [4], [11].

In addition to supporting its own environment, I beleive that Prolog is, in principle, an ideal language for implementing a VLSI CAD system. The main requirements of such a system are the ability to integrate a large database with algorithms that manipulate it. No other programming language supports both database and algorithmic functions in the way Prolog does. The main obstacle to realize such a pratical system today seems to be Prolog's inefficiency.

3 Concurrency

Statement: ***Prolog is not suitable for expressing concurrency.***

Inspite of its expressivness in general, Prolog has a major blind-spot: it is not suitable for expressing concurrency. This means that in order to build a Prolog machine we must either extend Prolog substantially, or use a lower programming language to implement multi-tasking. Needless to say, multi-tasking is an essential feature even in sequential computers.

There have been many proposals to incincorpoe more sophisticated control-constructs to Prolog, to support coroutinning and concurrency, e.g. in IC-Prolog, Prolog-II, MU-Prolog, and Epilog, among others. It seems that none of those is both expressive and efficient enough to implement a multi-tasking operating system.

Statement: ***Concurrency and don't-know nondeterminism (deep backtracking) do not mix well, but can be interfaced.***

The Relational Language of Clark and Gregory [2] and Concurrent Prolog [13] take a different approach. They give up Prolog's non-determinism (implemented by deep backtracking) for the sake of expressing concurrency. The memory management of these languages is very different from that of Prolog, therefore integrating the two efficiently on a von Neumann machine is a non-trivial problem. Also, my experience with programming in Concurrent Prolog suggests that applications that require concurrency do not require non-determinism, and vice versa. A good interface between Prolog (or a logic programming-based database machine) and concurrent logic-programming languages are set expressions, or Prolog's setof predicate, as suggested by Clark and Gregory [3]. The availability and sufficiency of such an interface reduces the need for an immediate integration of the two languages.

4 Parallelism

Statement: ***Parallel execution of Prolog is difficult.***

Logic programs offer two kinds of parallelism: Or-parallelism and And-parallelism. Or-paralelism means trying several candidate clauses in parallel. And-parallelism means trying to solve several goals in a conjunction in parallel.

One approach to designing a logic programming language for parallel computers is to patch Prolog. However, since Prolog was designed specifically

for efficient execution on a von Neumann machine, it is not clear that it is a good starting point. Adding Or-parallelism to Prolog is not so difficult concenptually. One problem is the cut. If cut is used to implement implicit negation (a substitute for if-then-else) and defaults, then Prolog programs may behave incorrectly when executed in Or-parallel mode. This is a difficult problem, due to the pervasiveness of this use of cut. Another problem is memory management. The overhead of maintaining seperate environments for the Or-parallel subcomputations may eliminate the benefits gained from their parallel execution.

Incoprorating And-parallelism into Prolog is far more difficult.

Statement: ***And-parallelism and Or-parallelism have different applications, and are best explored independently.***

Since the problems of parallel computers are so difficult, and there is so little positive experience with them in other branches of computer science, I think it is much more sensible to start small.

The first step is to examine the uses of the two kinds of parallelism. Or-parallelism is useful for speeding the solution of problems that require search. One significant class of search problems are database queries. In many applications, however, good algorithms can often provide a substitute for simple brute-force search. And-parallelism is useful for implementing parallel algorithms. The class of problems for which efficient parallel algorithms have been designed is increasing rapidly. The existence of computers that can actually run them will no doubt increase the pace in which they are produced. These observations suggest that And- and Or-parallelism have different, disjoint applications, which, at least initially, are best studied seperately.

The design of an Or-parallel database machine is an important and challenging problem. The close relationship between logic programs and relational databases suggests that ideas and concepts from relational databases can readily be put into use within the logic programming framework.

Concerning And-parallel machines, my view is that simple languages such as the Relational Language and Concurent Prolog are a good starting point. These languages are expressive enough in their current form for a parallel implementation of them to be useful and interesting. Hence their incporporation with the more powerful features of sequential Prolog may be postponed until the problems of building a parallel machine for these simpler languages are better understood.

5 Logic vs. control

91

Statement: *Efficient algorithms cannot always be obtained by twiddling with the control of logic programs.*

Some of the research on logic programming was guided by the desire to find some 'philosopher's stone': a notation that eliminate the need to think algorithmically. Kowalski's celebrated equation [6]:

$$\text{Algorithm} = \text{logic} + \text{control}$$

have suggested to many [4],[7],[8],[10], that if only we could find the 'right' control regime, we could factor the task of devising and implementing efficient algorithms into two, independent subtasks: defining the logic of a solution to a problem, and converting it into an efficient algorithm by imposing control on it. I beleive that this interpretation of the equation is too strict. It is impossible, in general, to specify sophisticated algorithms just by modifying the control component of a logic program. No massaging will make a logic program that specifies the exponential generate-and-test permutation sort into quicksort. The same statement is certainly true for less basic algorithms.

Statement: *Sophisticated control has large runtime overhead, hence it is best implemented in an embedded language.*

The sophisticated control regimes developed in response to Kowalski's equation usually have an unacceptable runtime overhead. Hence they cannot be incorporated in a base language. An alternative way to achieve sophisticated control is to implement embedded languages in Prolog. Implementing interpreters for embedded languages in Prolog is by now a well understood technique [9].

Statement: *Compile-time optimizations are superior to runtime optimizations.*

One of the goals of sophisticated control is to make certain logic programs run faster. This approach may be called "runtime optimization".

Whenever a runtime optimization of an inefficient logic program represents a tracktable algorithm, it is usually possible to implemente the algorithm directly in ordinary Prolog. The transformation of inefficient logic programs to efficient Prolog programs may be called "compile-time optimization".

It is my (unsupported) beleif that compile-time optimizations represent a more promising approach than runtime optimizations.

6 The Fifth Generation Project

Statement: *Logic-programming machines will require new solutions to old problems.*

One goal of the Fifth Generation project is to construct computers with a new machine language, based on logic. To realize such machines we will have to address many questions which are already solved for von Neumann computers. There is a lot to learn from the old solutions, but one measure for the viability of logic-programming is the quality of the new solutions it will provide to these old problems.

7 References

[1] Lawrence Byrd, *Prolog-10 Debugging Facilities,* Technical Note, Department of Artificial Intelligence, Edinburgh University, 1980.

[2] Keith Clark and Steven Gregory, A Relational Language for Parallel Programming, In *Proceeedings of the ACM Conference on Fucntional Programming Languages and Computer Architechture,* pp. 171–178, 1982.

[3] Keith Clark and Steven Gregory, *PARLOG: A Parallel Logic Programming Language* (Draft), Technical Report DOC 83/5, March 1983.

[4] Keith Clark and F. McCabe, The Control Facilities of IC-PROLOG, In:*Expert Systems in the Micro Electronic Age,* D. Mitchie (ed.), Edinburgh University Press, pp. 122–149, 1981.

[5] Paul Egghart, Logic enhancement. In *Proceedings of the ACM Conference on Lisp and Functional Programming Languages,* August, 1982.

[6] Robert A. Kowalski, Algorithm = Logic + Control, *CACM* **22(7)**:426–346, July 1979.

[7] John McCarthy, *Coloring Maps and the Kowalski Doctrine,* Technical Report STAN-CS-82-903, Stanford University, April 1982.

[8] Lee Naish, *An Introduction to MU-Prolog,* Technical Report 82/2, Department of Computer Science, University of Melbourne, 1982.

[9] Luis M. Pereira, Logic Control with Logic, In *Proceedings of the First International Logic Programming Conference,* pp. 9-18, ADDP, Marseille, September 1982.

[10] Luis M. Pereira and Antonio Porto, Selective Backtracking, In *Logic Programming,* K.Clark and S.-A. Tarnlund (Eds.), Academic Press, 1982.

[11] Antonio Porto, Epilog: a language for extended programming in logic, In *Proceedings of the First International Logic Programming Conference* pp. 31-37, ADDP, Marseille, September 1982.

[12] Ehud Y. Shapiro, *Algorithmic Program Debugging,* ACM Distinguished Dissertation Series, MIT Press, 1983.

[13] Ehud Y. Shapiro, *A Subset of Concurrent Prolog and its Interpreter,* Technical Report TR-003, ICOT — Institute for New Generation Compputer Technologu, 1983.

[14] SZKI. *MProlog Language reference Manual,* SZKI, Budapest, Hungary, November 1982.

[15] David H. D. Warren, Higher order extensions to Prolog — are they needed? In *Machine Intelligence* **10,** D. Michie, J. Hayes, and Y. H. Pao (eds.), Ellis-Horwood, 1982.