Parallel Interpretation of Logic Programs

by

John S. Conery *
Dennis F. Kibler

Technical Report 173

April 1981

Department of Information and Computer Science
University of California
Irvine, Ca. 92717

* The authors' names appear alphabetically

Portions of this research were supported by a grant from the
Naval Ocean Systems Center (N66001-80-C-0377)

Table of Contents

1. INTRODUCTION 2
2. COMMON FEATURES 4
   2.1. Limitations 6
   2.2. Processing Elements and Messages 7
   2.3. Processes 8
3. AND/OR MODEL 10
   3.1. General Description 10
   3.2. OR Process 11
   3.3. AND Process 12
   3.4. Communication 12
   3.5. Summary 14
   3.6. Example 14
4. GOAL LIST MODEL 14
5. METRICS FOR COMPARING PROLOG INTERPRETERS 19
6. FUTURE RESEARCH 19

List of Figures

Figure 1-1: Example of a Prolog Program 5
Figure 3-1: Progress of Processes in the AND/OR Model 15
Figure 3-2: AND/OR Tree 15
Figure 4-1: Goal List Tree 17

ABSTRACT

Logic programs offer many opportunities for parallelism. We present two models of computation which allow parallel processing of Prolog programs. In the goal list model each processor is given a separate approach to the problem. In the AND/OR model each processor is given a separate subcomputation. Each model allows an arbitrary number of processors, and both perform the same sequence of unifications as would the standard depth first interpreter for Prolog if only one processor were available. In each model the parallelism is achieved as a result of the non-determinism in the program.

1. INTRODUCTION

Prolog is a high level applicative language in which programs are sets of Horn clauses. The language has been implemented on a number of different computer systems (c.f. [8]), and has been found to be comparable to LISP in terms of execution speed [11]. The DECsystem-10 interpreter [8] uses a depth first search of an AND/OR tree defined by the program. There are a number of other, more "intelligent" interpreters. IC-Prolog [2] has control annotations to help guide the search by using certain runtime information. A selective backtracking interpreter [9, 10] keeps track of where values are created, so that if a value later causes a failure, the interpreter can backtrack directly to the source of the error.

In this paper we present two abstract models for parallel execution of Prolog programs. At present, these models represent a line of research that is independent of the intelligent interpreters; perhaps in the future parallel interpreters will incorporate some of the methods developed for intelligent interpreters.

Section 2 is a discussion of some items that are common to both of our parallel models; the models themselves are presented in sections 2 and 3. We conclude with a discussion of metrics that can be used to compare the performance of the various single- or multi-processor interpreters and plans for future research. The remainder of this section contains definitions of terms and a

brief introduction to Prolog (using the syntax of DECsystem-10 Prolog).

There are two kinds of clauses in Prolog programs: implications and assertions. Implications are of the form

```
w :- x, y, z.
```

where w, x, y, and z are terms. w is the head of the clause, and those terms to the right of the :- symbol form the body of the clause. Assertions are clauses that have an empty body. Terms can be atoms, variables, or structured objects. The names of variables (which take values of either atoms or structures) begin with upper case letters, and all other names must begin with a lower case letter. The "scope" of a variable is a single clause; if X appears in two different clauses, it will not necessarily refer to the same object.

A Prolog program is activated by giving it a goal list, which has the syntax of a clause with no head. The interpreter tries to solve the goals in order, from left to right. To solve a goal, the interpreter looks for a clause whose head unifies with the goal (we sometimes say that a head matches a goal if they can be unified). Two terms can be unified if there is a substitution for variables that makes the terms identical. For example, the two terms

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

can be unified by the substitution {X/b,Y/a}, meaning substitute "b" for X and "a" for Y, to give the single term p(b,a). After the goal is unified with the head of some clause, the unifying substitution is applied to the entire goal list and the body of the clause, and this new body now replaces the original goal at the front of the goal list. Note that under this interpretation, terms in the body of a clause can be considered subgoals, i.e. one can read the statement

```
w :- x, y, z.
```

as "in order to solve the goal w, solve the goals x, y, and z in that order." Assertions (goals with no subgoals) always succeed.

If the current goal does not unify with the head of any clause, it fails, and the interpreter backtracks by undoing the latest unifying substitution, and trying another clause for the most recently matched goal. When the goal list is empty, the interpreter stops and prints the values obtained for any variables that were in the original goal.

Figure 1-1 contains an example of a Prolog program, and shows the series of steps carried out when the program is given various goals to solve. References [2], [5], [7], and [8] are other descriptions of logic programming and Prolog.

2. COMMON FEATURES

Both of our models are distantly related to dataflow models of

Prolog program:

```
      clauses                                  comments

(1) gf(X,Z) :- p(X,Y), f(Y,Z).   /* The grandfather of X is Z if */
                                 /* a parent of X is Y and Z is  */
                                 /* the father of Y. */
(2)  p(X,Y) :- m(X,Y).           /* Y is a parent of X if Y is   */
                                 /* the mother of X. */
(3)  p(X,Y) :- f(X,Y).
(4)  m(doug,peg).                /* peg is the mother of doug    */
(5)  m(den,peg).                 /*       (an assertion)         */
(6)  f(doug,larry).
(7)  f(den,larry).
(8)  f(larry,sam).
```

Example: find G such that G is the grandfather of den.

step	goals to be solved	matching clause	unifying substitution
[1]	:- gf(den,G)	1	{X/den,Z/G}
[2]	:- p(den,Y), f(Y,G)	2	{X/den}
[3]	:- m(den,Y), f(Y,G)	4	{Y/peg}
[4]	:- f(peg,G)	none	

Retry step [3], i.e. does den have another mother? This also fails, so retry step [2], i.e. does den have another parent? This time the goal p(den,Y) is unified with clause 3, giving us

[5]	:- f(den,Y), f(Y,G)	7	{Y/larry}
[6]	:- f(larry,G)	8	{G/sam}
[7]	<empty>		

An empty goal list means success; the variable G in the original goal list was bound to sam, thus giving the answer: sam is the grandfather of den.

Note: it is also possible to use this program to solve the goal gf(G,sam), i.e. who is the grandson of sam? This is an example of a nondeterministic goal; Prolog will generate the answer G = doug first, and then (if told to backtrack), the answer G = den.

Figure 1-1: Example of a Prolog Program

computation (c.f. [1]). Research in dataflow architectures can be described at three distinct levels of abstraction: there is a high level language, an intermediate level graph language called the base language, and, at the lowest level, a multiprocessor machine architecture.

Our high level language is Prolog. Both of our models for parallel execution of Prolog programs are explained in terms of dynamic tree structures that are created as the program is interpreted; these trees are the base language. Our models differ in the information stored at each node and in the rules for creating descendants of nodes. No attempt has yet been made to define the physical architecture; we anticipate that much of the work that has been already been done on dataflow architectures will be applicable.

Each model will perform with any number of processors. If there is only one processor, both models will behave like the standard Prolog interpreter. If there are more processors then required, both models will evaluate each non-deterministic choice asynchronously. With fewer processors only some of the non-deterministic choices will be searched in parallel.

2.1. Limitations

In order to keep the models as simple as possible, we have made the following assumptions for the initial versions of the interpreters:

- Processors will have some amount of local memory.
- Programs will not modify themselves (i.e. no "assert" or "retract" procedures will be used).
- Programs will be small enough so that each processor can store a copy of all the clauses.
- The programmer is responsible for coding in a style which will allow parallel processing. Opportunities for parallel processing arise from either non-deterministic choices or from independent subcomputations.
- The programmer is responsible for coding in a style which does not lead to inordinate data tranfers.
- The programmer can no longer control his program execution by "falling through". Since the interpreters execute clauses asynchronously, each clause must stand on its own as a true statement. We regard this limitation as a demand for good coding style.

2.2. Processing Elements and Messages

Processors (PEs) in our models are capable of sending messages to one another, performing the primitive operations of Prolog, e.g. unification and goal list construction, and storing information in a local memory.

Processors are either active or idle. At times a PE will divide the problem it is currently working on into two or more subproblems which can be solved in parallel. When this occurs, the PE can send any of the subproblems to a dispatcher, which in turn sends the subproblems to PEs that it knows are idle. This notion of a dispatcher is simply a convenient means for modeling the fact that idle processors will begin to work on subproblems set up by other processors; the actual mechanism for performing this distribution of subproblems will not necessarily have to be

a special PE in the system, and in fact will probably be similar to the mechanisms used in dataflow systems whereby a PE decides to work on an activity when all of the input tokens have been computed for that activity (c.f. [4]). If the dispatcher receives a subproblem from a PE, and there are no more idle PEs in the system, the problem is saved and assigned when a PE becomes idle. A second function of the dispatcher is to open a "communication channel" between a PE that originates a process and the PE that is eventually assigned to work on it. Thus, aside from the original assignment of a PE to a problem, the dispatcher is not involved in any interprocessor communication; rather, all messages are sent directly from one PE to another.

When a PE cannot proceed any further on its current problem (this state is defined later, in the context of a particular model), it notifies the dispatcher that it is idle. Note that this implies that a PE can possibly work on subproblems that it creates.

2.3. Processes

A problem being worked on by a PE is modeled as a process which can be in one of three states: active, ready, or suspended. There is only one active process on a particular PE at any one time. The PE maintains lists of other processes that it has worked on and that are currently suspended or ready. The PE works on the active task until such time as subproblems are identified; at this time, the process is put in the suspended

state; all but one of the subproblems are sent to the dispatcher to become processes on other PEs; the remaining subproblem is used to create a new process which is put on the list of ready processes (it is ready since the PE can immediatley begin to work on it); finally the PE selects one of the ready processes and makes it the active process.

From time to time, messages will arrive from other PEs, containing the results of subproblems. There are two possible results: failure or success. In the latter case, the message also contains a (possibly empty) list of values that were bound to variables during the solution of the subproblem. This result is recorded with the appropriate parent problem (the process for which is in the list of suspended processes). When enough information about its subproblems has been received, the process for a problem will be changed from suspended to ready. When the PE selects this problem for activation, it computes a result, and reports the result to the originator of the problem.

Our two models differ in what constitutes a subproblem, and what is meant by "enough information" to compute a result.

To summarize, a process typically goes through the following sequence of states:

- active, until the PE detects possible parallel subproblems.
- suspended, until results of subproblems are computed.
- ready.

- active, when results of subproblems are used to calculate the result of the orignial main goal.

Another type of message that can be sent from one PE to another is a "kill" message, where a PE that created a subproblem tells another PE to abort that subproblem. A process receiving this message will pass it on to all of its descendants, and then terminate itself (i.e. it will no longer be in either the ready list or the suspended list for the PE that was working on it). Situations where these messages originate are described below in association with a particular model.

3. AND/OR MODEL

3.1. General Description

There are many choices for a parallel interpreter for logic programs which are based on an AND/OR tree. We plan to experiment with several, but in this discussion, only one model, the simplest, will be explained. To understand the AND/OR interpreter, imagine the complete finite AND/OR tree associated with the genealogy problem (see figure 3-2). The top node of the tree is the AND node which represents the user's request. The sons of each AND node are OR nodes and the sons of each OR node are AND nodes. At any point in the parallel computation, processes will be associated with nodes in a subtree of the AND/OR tree which includes the root. A dispatcher will assign processors to these processes. Arcs between nodes in the tree correspond to a message passing capability between processes.

When a process finishes a task, it tells its father. If a processor has an empty ready list, it informs the dispatcher that it is free. This allows it to enter into the job stream again.

To understand this AND/OR interpreter it suffices to understand the AND process, the OR process, the dispatcher, and the communication among them.

3.2. OR Process

An OR process has a single goal to solve. When it finds a solution it passes this to its father, suspends itself, and waits for a message from its father to either send another solution or to kill itself. A solution is defined by a (possibly empty) substitution list. The OR process collects and saves other solutions to the goal that it may receive from its sons; these solutions will be sent to the parent upon request.

The OR process begins to solve a goal by unifying the goal with each head term in the program, which is a collection of clauses. Each successful unification generates a substitution list which is applied to the body of the corresponding clause. Each instantiated body will be the goal list for an AND process. These AND processes will become sons of the OR process. If the body is empty, the OR node has a trivial success.

In brief, OR processes allow for parallelism when the program has non-deterministic choices.

3.3. AND Process

An AND process has a list of goals to satisfy. Possible ways of defining an AND process vary greatly. The one we give is similar to the Prolog interpreter. The AND process sends the first goal in its goal list to an OR process and suspends itself. When this goal is satisfied, it instantiates the next goal and sends this off to an OR process. If a son fails its goal, the AND process asks the previously suspended process for another solution. If an AND process reaches the cut symbol "!", then it kills the sons prior to the cut. The AND process treats the goal "fail" exactly as if this goal were assigned to an OR and this goal failed. When an AND process constructs a solution to the entire goal list, it suspends itself and passes the answer to its father.

A more efficient AND process would adopt the intelligent backtracking ideas of [9, 10]. By severely complicating the tasks of the AND process, using ideas suggested by [6] and [9], one can achieve parallelism for independent subcomputations.

3.4. Communication

Each message in the parallel interpreter is tagged with the source and destination. A background job for every PE is to route messages to their destination process. The various forms of messages, and their interpretations are:

1. Kill message: A father sends this message to a son. The son process kills itself as well as sending kill messages to all of its sons.

2. Redo message: Sent by a father to a son, this message tells the son to compute a different solution than the any that have been sent previously. It causes the son to change from the suspended to the ready state.

3. Failure message: A fail message is sent from a son to his father when his goal or goal list cannot be satisfied. After a process sends a fail message, it kills itself. Fail messages are handled differently by OR and AND processes.

    a. OR process: Upon receiving a failure message, an OR node merely disowns his son. If he has no more sons, he too fails.

    b. AND process: The AND process awakens the process responsible for the previous goal. If the first goal sends a failure message, the AND process fails.

4. Success messages: A success message is a (possibly empty) substitution list. It is sent from son to father. AND and OR processes handle success messages differently.

    a. OR process: The OR process compares the solution with previous solutions, and if it is new, sends it to its father when the father sends a redo message. If the answer is not new, the son is sent a redo message.

    b. AND process: A success message to an AND process causes the process to be moved from the suspended state to the ready state. If the answer was from the last goal of the goal list, the answer is sent to the father of the AND process. Otherwise the next goal in the goal list is further instantiated and passed to an OR process and the AND process is again suspended.

5. Goal List Request: Sent from an OR process to the dispatcher, it sets up a son. If the list has more than one element, the son is an AND process. If the list has only one element, then the son can be an OR process, rather than setting up a trivial AND process.

6. Goal Request: Sent from an AND node to the dispatcher, it sets up a son, which is an OR process. If the OR

process unifies the goal with only one clause, then the son can be regarded as an AND process rather than setting up a trivial OR process.

7. Idle message: sent by a PE to the dispatcher when the PE has no processes in the ready state.

3.5. Summary

The AND/OR model given here allows for parallelism only at the OR nodes of the AND/OR tree. This model is adopted because it clearly separates the processes at AND and OR nodes. If the computer system had only one processor, then this model gives the same results, with degraded performance, as would the standard Prolog interpreter. The programmer cannot rely on the order of the clauses he defines to dictate the order that the clauses will be satisfied. The programmer can rely on the fact the body of the clauses will be executed in the order that he has specified.

3.6. Example

Because of the inherent asynchrony in the AND/OR model, a particular answer to a goal cannot be guaranteed. However the first three time slices of a computation that an AND/OR interpreter might obtain are illustrated in the following figure. The notation S stands for suspended, -- stands for not applicable, and PE1 and PE2 are different processors.

4. GOAL LIST MODEL

The goal list model was first presented in an earlier paper [3]. In this interpreter, every node of the tree contains a goal list which represents a complete <u>continuation</u> of the problem,

goal	process	father	time --> 1	2	3
gf(X,Z)	1	user	PE1	S	S
p(X,Y)	2	1	--	PE1	S
m(X,Y)	3	2	--	--	PE1
f(X,Y)	4	2	--	--	PE2

Figure 3-1: Progress of Processes in the AND/OR Model

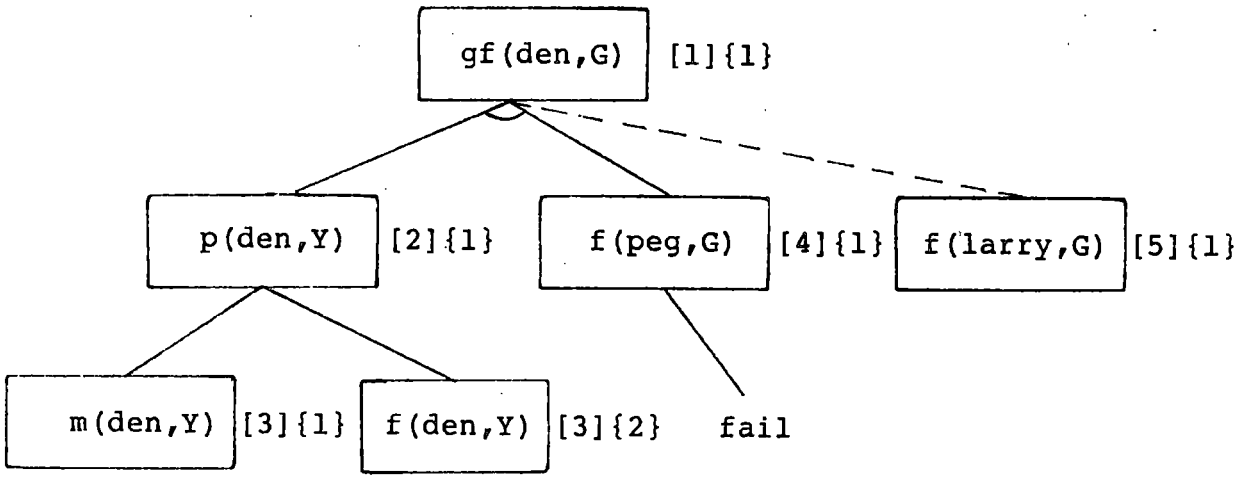

Figure 3-2: AND/OR Tree

The number in brackets next to a goal indicates the order in which a process was created to solve the goal i.e. gf(den,G) was the first process created; p(den,Y) was the second process created, and so on.
The number in braces indicates which PE is assigned to solve the goal.

i.e. if every goal in the list is solved, then the top node problem will be solved. In contrast, the goal lists at AND nodes of the AND/OR model represent independent parts of the top node problem.

Descendants of a node are created by selecting a goal from the list and unifying it with as many heads of clauses as possible. Each successful match creates a descendant, where the goal list in a descendant consists of both the body of the matched clause and the remaining goals from the parent's list. The unifying substitution is applied to the entire list in the descendant. Figure 4-1 shows a goal list tree for our example program.

The creation of descendants in this model essentially combines the effects of AND node processing and OR node processing of the AND/OR interpreter into one step, with the additional effect of having OR nodes pass along the remainder of their parents' goal lists with the new body. As the figure shows, each path from the root to a leaf represents an independent approach to the solution of the entire problem.

A branch node in a goal list tree is defined to be a node that has more than one descendant. A process is then defined to be a path that starts with an immediate descendant of a branch node and terminates at either a leaf or a branch node. The three processes of our example are labeled in figure 4-1.

A computation in the goal list interpreter is similar to a

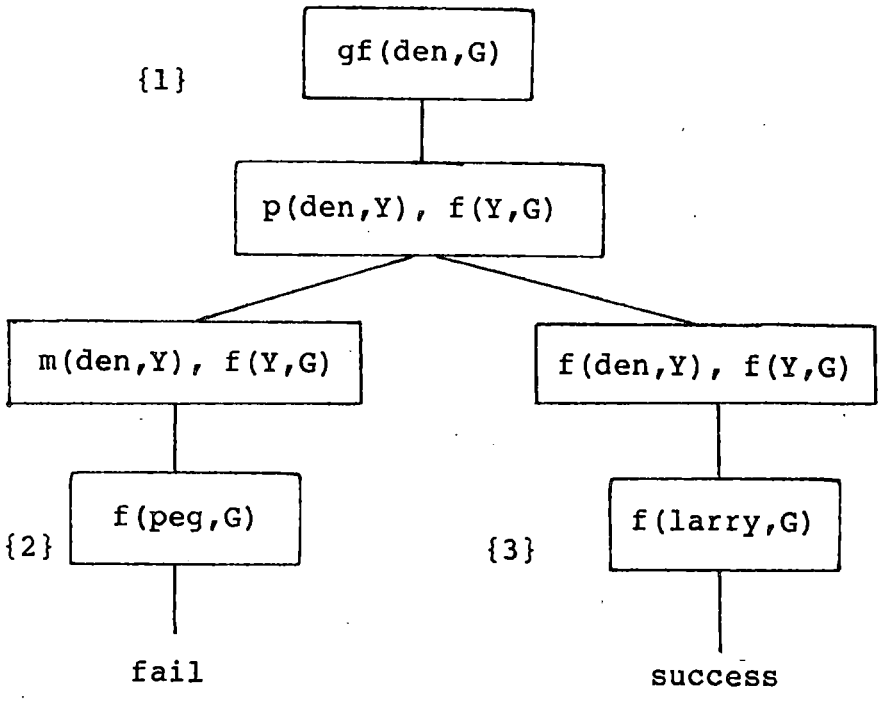

Figure 4-1: Goal List Tree

The numbers in braces refer to process numbers; processes {2} and {3} are descendants of {1}; PE1 interprets {1} and {2}, and PE2 works on {3}

computation using the AND/OR interpreter. A list of goals is sent to a PE, and the PE creates a process from the list. A goal is selected from the list and is used to create descendant lists. If there are no descendants, the process terminates in failure and sends a fail message to its parent. If there is exactly one descendant, the process can continue, and the PE puts it on its ready list. If there are two or more descendants, the current list corresponds to a branch node, and therefore the process is suspended pending results of each of its descendant processes. One descendant process is put on the ready list of the current PE, and the others are sent to the dispatcher. Any time the empty list is generated as a descendant list, it represents a successful computation; a success message is sent to the parent, and the process terminates.

Whenever a process generates a branch node, and suspends itself, it behaves like an OR node in the AND/OR model:

1. the first success message from a son is passed on to the parent process; all subsequent successes are stored.

2. a redo message from the parent causes the next succcessful son's result to be sent to the parent (assuming that the new result is different from any previous result).

3. when all sons have reported, with either success or failure, the next redo message from the parent causes the process to send the parent a fail message and then kill itself.

5. METRICS FOR COMPARING PROLOG INTERPRETERS

Some of the metrics that we plan to gather statistics on are:

1. Size and number of messages sent during the solution of a problem.
2. For each PE, the number of ready processes and the number of suspended processes generated during the solution of a problem.
3. Measure the ratio of idle time to processing time for each PE.
4. For each PE, the costs for preparing, routing, and receiving messages.
5. Cost for data base search for each processor.

It will also be possible to measure the number of unifications attempted, and the number of successful unifications, during the solution of a problem by one of the single processor interpreters, and compare them with the same statistics for the parallel interpreters.

6. FUTURE RESEARCH

The next step in our research is to build a simulator for the models of parallel computation that we have defined. We expect that the simulator will allow us to experiment with a variety of other models. In particular we are eager to try to incorporate various "intelligent" single processor interpreters into a parallel processing framework. We also want to test various models which allow for parallel execution of independent subcomputations, as might arise in a program whose high level plan was divide and conquer. A simulator is necessary to gain

some insight into the cost tradeoffs between different parallel models of computation.

It should also be possible to perform a "data dependency analysis" among goals in a goal list, in order to gain some insight into which of the goals should be solved before others, or which goals could be solved in parallel. This analysis could be performed by AND processes in the AND/OR model, and at every step in the Goal List interpreter. We have done some preliminary work in this area (this is the source of the "severe complications" alluded to earlier).

REFERENCES

1. Arvind, Gostelow, K. P., and Plouffe, W. P. An Asynchronous Programming Language and Computing Machine. Technical Report 114a, Dept. of Information and Computer Science, University of California, Irvine, December, 1978.

2. Clark, K. L., and G. McCabe. The Control Facilities of IC-Prolog. In D. Michie, Ed., Expert Systems in the Micro Electronic Age, Edinburgh University Press, 1979.

3. Conery, J. S., P. H. Morris, and D. F. Kibler. Efficient Logic Programs: A Research Proposal. Technical Report 166, Dept. of Information and Computer Science, University of California, Irvine, april, 1981.

4. Gostelow, K. P. and R. Thomas. Performance of a Simulated Dataflow Computer. IEEE Transactions on Computers C-29, 10 (October 1980), 905-919.

5. Kowalski, R. A. Predicate Logic as a Programming Language. Proc. IFIPS 74, 1974.

6. Kowalski, R. A. Logic for Problem Solving. Elsevier - North Holland, New York, 1979.

7. Kowalski, R. A. Logic as a Computer Language. Proc. Infotech State of the Art Conference "Software Development: Management", June, 1980.

8. Pereira, L. M., F. C. N. Pereira, and D. H. D. Warren. User's Guide to DECsystem-10 Prolog. Dept. of Artificial Intelligence, Univ. of Edinburgh, September, 1978. version 1.32

9. Pereira, L. P. and A. Porto. Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Theory. Report 2/79, Departamento de Informatica, Universidade Nova de Lisboa, October, 1979.

10. Pereira, L. P. and A. Porto. An Interpreter of Logic Programs Using Selective Backtracking. Report 3/80, Departamento de Informatica, Universidade Nova de Lisboa, July, 1980.

11. Warren, D. H. D., L. M. Pereira, and F. C. N. Pereira. Prolog - The Language and its Implementation Compared with LISP. ACM SIGPLAN Notices 12, 8 (1977), 109-115.