Proc. COMPCON, San Francisco, CA, IEEE, March 1988

PROLOG AT THE UNIVERSITY OF ILLINOIS*

M. M. Gooley L. V. Kalé D. A. Padua B. Ramkumar U. S. Reddy D. C. Sehr W. W. Shu B. W. Wah

University of Illinois at Urbana–Champaign

Abstract. This paper presents a brief description of four logic programming research projects under way at the University of Illinois at Urbana-Champaign. Three of these projects deal with the design of Prolog interpreters and compilers. The other project deals with the design of languages that combine then functional programming and the logic programming paradigms and with the transformation of programs written in these languages.

Introduction

The logic programming language Prolog was developed about fourteen years ago by A. Colmerauer and P. Roussel [Rous75]. Prolog in particular, and logic programming languages in general are today the subject of increasing interest on the part of many research groups. This paper presents a brief description of four of the logic programming research projects under way at the University of Illinois at Urbana-Champaign. The first three section of the paper describe projects dealing with the design of Prolog interpreters and compilers emphasizing efficiency and parallelism. The first project deals with the use of heuristics to reduce the search space. The second project concentrates on efficient OR-parallel execution of standard sequential Prolog programs. The third project studies the use of both AND and OR parallelism. The last project presented here deals with the design of languages that combine the functional programming and the logic programming paradigms and with the transformation of programs written in these languages.

Heuristic guiding and pruning

We have performed an extensive survey of computers, from both the hardware and software perspectives, to support artificial intelligence processing [WaLi86a, WaLi86b, Wah87, WaLi88]. We have found that the execution of Prolog programs is inefficient due to redundant searches. Our research on efficient execution of Prolog programs is centered around two themes: reducing the search space through heuristic guiding and pruning, and efficient execution of Prolog programs by parallel processing [LiWa86e].

In heuristic guiding, we have studied two related problems: the identification of attributes that can be used to guide the evaluation of Prolog programs, and the static reordering of programs at compile time to reduce redundant searches.

In identifying attributes to guide the search, we model the evaluation of a Prolog program as the search of an AND-OR tree. An attribute that has been found to be useful to characterize the merit of evaluating a subtree is the ratio of success probability of the subtree to the corresponding overhead (cost) of evaluation [LiWa85]. If a depth-first search is used and all nodes in the search tree are independent, then an optimal search strategy to minimize the expected total cost is to reorder the descendents of AND nodes by increasing the ratios of success probability to cost and to reorder OR nodes by decreasing the ratios of success probability to cost [LiWa86d]. In a best-first search, the AND-OR tree has to be transformed so that all OR nodes are in the top part of the search tree and all AND nodes are in the lower part. An optimal strategy to search the transformed tree is to search all OR nodes by decreasing ratios of success probability to cost and all AND nodes by increasing ratios of success probability to cost [LiWa85]. It should be noted that when the search tree is transformed as described above, its size is increased by an exponential order.

In practice, nodes in the AND-OR search tree are dependent and may be traversed more than once due to backtracking. We study the effects of backtracking by modeling the evaluation of subgoals in a clause and clauses with the same head as an absorbing Markov Chain [LiWa85]. Such an approach is heuristic in nature because nodes in the search tree may be dependent, while nodes in the Markov Chain are independent and satisfy the memoryless property. The evaluation of success probabilities and costs is recursive. That is, the success probabilities and costs in the lower part of the search tree are evaluated first. This will provide success probabilities and costs to clauses that call these subgoals, and their success probabilities and costs can then be evaluated.

*The work of L. V. Kalé, B. Ramkumar, and W. W. Shu was supported in part by the the National Science Foundation under Grant NSF-CCR-8700988. The Work of D. A. Padua and D. C. Sehr was supported in part by the National Science Foundation under Grant NSF-MIP-8410110, the Department of Energy under Grant DOE DE-FG02-85ER25001, and a donation from IBM Corporation to the Center for Supercomputing Research and Development. D. C. Sehr holds a fellowship from the Office of Naval Research. The work of U. S. Reddy was supported in part by the National Aeronautics and Space Administration under Grant NAG 1-613. The work of B. W. Wah and M. M. Gooley was supported in part by the National Aeronautics and Space Administration under Grant NCC 2-481.

Based on the technique described above, to approximate the merit of evaluating a clause or subgoal, we have studied the static reordering of Prolog subgoals and clauses to minimize the expected total cost [GoWa88]. Two major problems have been addressed. First, it is necessary to define the equivalence of results when the execution order is changed. Equivalence can be thought of as what is acceptable to the users as an equivalent solution set. We have defined four classes of equivalence: reflexive, set, tree, and inequivalent executions. In reflexive equivalence, the results in the solution set are obtained in the same order; this can only be achieved through minimal or no reordering. In set equivalence, the same set of results are obtained but possibly in a different order. In tree equivalence, a superset of subset of solutions may be generated, while in inequivalence, any result can be generated. Reordering may result in set, tree, or inequivalent executions. Our objective is to maintain set equivalence in the presence of side effects and control predicates. The second problem addressed in this research is the characterization of restrictions on reordering and the development of methods to detect these restrictions. Restrictions on reordering are governed by fixity, semi-fixity, and control predicates. We have designed a new system of calling modes for Prolog, geared to reordering, and a system for inferring them automatically. We have developed an approach that used multiple versions of the same clause in different modes, each with different order of execution. We improved the Markov-Chain method for determining a good goal order for Prolog clauses, and used it as the basis for a reordering system, showing how it could be guided by information about modes and restrictions to generate reordered Prolog programs that behaved correctly.

In heuristic guiding, we have studied the detection of redundant evaluations in a Prolog program, the abstraction of previous evaluations by explanation-based learning, and the storage of previous solution sets by caching [WaLi88].

In parallel processing of logic programs, we are currently simulating a parallel processing system that evaluates Prolog programs based on the Markov-Chain method described above, the maintenance of equivalence of solution sets, and the automatic detection of precedence order of execution. The architectural model assumed is a multiprocessing system with a hybrid of tightly coupled multiprocessors and loosely coupled distributed computers [Li85,LiWa85]. The objective here is to develop methods to coordinate the search in a distributed fashion, without violating precedence while maintaining equivalence of results, and investigate strategies to avoid detrimental anomalies in parallelism [LiWa86c].

OR-parallelism in the presence of side-effects

The goal of this project is the design and implementation of a Prolog compiler that accepts standard sequential Prolog as input language and produces efficient code for a parallel processor. The translation of sequential programs into equivalent parallel versions has several advantages. The programmer's task is simplified since there is no longer the need to deal with parallelism explicitly, and sequential programs can be easily ported between different classes of machines. Research on this same compiling strategy for FORTRAN and other languages was pioneered at the University of Illinois many years ago. Techniques developed at Illinois are used today in the FORTRAN compilers of many supercomputers and minisupercomputers.

In the early stages of this project we decided to take advantage of OR-parallelism only, and leave to the project described in the next section the study of the interaction between AND and OR-parallelism. OR-parallelism was chosen because it seemed more likely to achieve good speedups on conventional multiprocessors. The reason for this is that, under OR-parallelism, the target parallel program can be organized in such a way that there is little interaction between the different processes cooperating in the search for solutions. Also, restricting the work to only one form of parallelism will allow us to concentrate more on the quality of the analysis and code generation phases. We hope that the outcome of this work will be a compiler capable of generating very efficient parallel code in a reasonable time.

The project is organized into three phases. During the first phase, an interpreter was written. During the second phase we will explore different strategies and assess their performance potential, and during the third phase we will write the compiler based on the techniques and measurements obtained during the first two phases.

The central structure in the interpreter or target parallel program is a representation of an *evolving OR-tree* whose root corresponds to the input query. In this tree, each node represents a set of bindings and a conjunction of goals. Each descendant of a node corresponds to a successful unification of its leftmost goal with a different clause. The clause used for generating a node will be called its *originating clause*. The bindings produced by the unification become the bindings of the new node. Also, the query of a node is the query of its parent with the leftmost goal replaced by the right-hand side of the originating clause. A node containing an empty query is called a *success*, and a node whose leftmost goal cannot be unified with any clause is called a *failure*.

The target parallel program consists of several cooperating processes. Assume first that there are no goals with side-effects in the Prolog program. Under this assumption, execution proceeds as follows. When execution starts, the OR-tree is just the root node containing the user input query. One of the cooperating processes grabs this node while the others remain idle. The process then tries to unify the leftmost goal with every clause in its procedure. A child node is created for every successful unification. The binding set and the query of each child node are also computed as described above, and both items are stored in the child node. Execution terminates if all the children of the root node are successes, or if no unification is possible, i.e. the root is a failure. Otherwise there will be some children that are *ready nodes*, i.e. nodes that are neither successes nor failures and whose children have not been generated.

After processing the root, the search of the OR-tree proceeds in a similar way. At any given time, there may be several ready nodes in the evolving OR-tree which gives the opportunity for parallelism by having each of several processes work on a different ready node. The work on each of these nodes is the same as the one described for the root node. Execution terminates when no ready nodes remain.

When predicates with side-effects such as `assert`, `retract`, and `write` are present, the processes have to perform the action corresponding to these predicates before generating the child node. In our system, predicates with side-effects other than cut are evaluated one at a time and in the order specified by the Prolog program. This is achieved by linking the nodes of the evolving OR-tree in post-order to form the *scheduling queue*. Processes look for work in this queue, and predicates with side-effects other than cut are evaluated only when they are the leftmost goal of the leftmost ready node in the scheduling queue.

The three most important issues with respect to this work are scheduling, how to handle different simultaneous bindings to the same variable, and how to correctly process the side-effect predicates. We decided to concentrate, at least at the beginning on the scheduling and side-effect issues which have not been studied as extensively as the multiple bindings issue. In the current version of the interpreter, we have implemented some of the models described in the literature [HaCH87,Warr87] with the purpose of studying their behavior before selecting one or trying to design a new scheme.

Next we discuss briefly the algorithms we use to handle the predicates with side-effects. The main idea of these algorithms is easy to describe. The details, however, are lengthy, and cannot be presented here due to space limitations. They can be found in [KaPS88].

Executing a cut is done by deleting from the scheduling queue the nodes between the cut node and the parent of its frame. This is a constant time operation in our system. When the children of a ready node, say *N*, are generated, the identification of *N* is inserted in a list associated with the procedure of the leftmost goal of *N*. When a clause is asserted into a procedure, this list is used to insert an additional child for all the appropriate nodes in the list. Similarly, when a new node is added to the tree, its identification is inserted into a list associated to its originating clause. This list is used to handle `retract`.

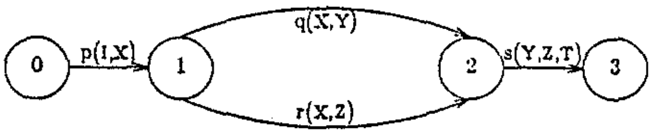
An important issue is the interaction between scheduling, garbage collection and the processing of `assert`, `retract`, and cut. The major difficulty is that care has to be given to the scheduling and the garbage collection algorithms to avoid redundant and useless work. For example, when a subtree is cut from the OR-tree, a process working in that subtree should be redirected towards another part of the tree to avoid wasting computation. Also, it may not be beneficial to garbage collect subtrees removed by cut, since the cut itself could later be removed by another cut.

The REDUCE-OR process model

The main motivation behind this research is a premise that massively parallel machines will be available in near future. We then need to extract maximal parallelism from given logic programs. This is particularly important for many combinatorially explosive AI computations. As a consequence, we cannot rely solely on AND or OR parallelism, but must pursue both sources of parallelism *in concert.*

We found that the AND-OR trees, which are a usual representation of logic computations, are not adequate for representing parallel computations. Firstly, as the binding information is spread through the tree, the nodes don't represent independent subproblems. Secondly, the tree hides an important form of parallelism, as there is only one OR child-node for every literal of an AND node [KaWa84]. The REDUCE-OR process model [Kale87a] is based on the REDUCE-OR trees which we developed as a representation of logic computations suitable for parallel interpretation.

We will use a simple example to illustrate the basic operation of the model. Figure 1 depicts a data join graph (DJG: a form of data dependence graph), for a clause. I is the input value given to this clause. Assume that p,q and r are non-deterministic, and return multiple solutions. As the graph indicates, p computes values for X that q and r can use in parallel. The ROPM starts a new instance of q and r as soon as a X value is returned from p. Every Y value returned from q is paired with each Z value that has been returned by r and that shares the same X value as an ancestor. (Similar, symmetric, pairing happens when a Z value arrives from r) For every new pair so formed, a new instance of s process is started to compute a T value. Thus, if p returns two X values, and q and r each give 3 values for every X value, the ROPM will form 9 Y-Z pairs for each X value, and may have 18 s processes running in parallel at a time. In contrast, the AND-OR process model (AOPM) [Cone83] can have only one instance of s running at a time. A pure OR process model will of course miss the parallelism between q and r. The AOPM does execute alternate clauses for p in parallel. Similarly, for each X value, it will have 3 processes each for q and r running in parallel. But as it deals with one binding for each variable at a time, it misses the parallelism between multiple instances of a consumer literals. We call this *consumer instance parallelism.* We find that it is an important source of parallelism in AI computations where the generate-and-test paradigm prevails. In an 8-queens program, admittedly a toy example, the ROPM obtains about 26,400 parallel actions in contrast to 4,544 obtained by a pure OR model, and just 7 obtained by AOPM. (The AOPM can get the same degree



$$\begin{tikzcd} 0 \arrow[r, "p(I,X)"] & 1 \arrow[r, bend left, "q(X,Y)"] \arrow[r, bend right, "r(X,Z)"'] & 2 \arrow[r, "s(Y,Z,T)"] & 3 \end{tikzcd}$$

Figure 1: The Operation of ROPM

of parallelism by rewriting the Logic program).

We have shown in [Kale87b] that ROPM is *complete* and that it produces more parallelism than most proposed models. This 'maximal parallelism' comes at a price. The ROPM has to do much more book-keeping than the AOPM, say. As a simple example of the complexities involved, notice that the Y values and Z values coming from q and r (in Figure 1) have to appropriately distinguished to ensure that a solution for q(x1,Y) is not paired with a solution for r(x2,Z). For a model such as the AOPM, this is no problem because it allows only one X value to be alive at a time. The challenge is then to develop techniques and algorithms that control the overhead in ROPM. In fact, we aim at reducing the overhead to the extent that even on computations where ROPM generates the same parallelism as AOPM, it should be as efficient as AOPM (and within a small factor of sequential compiled execution). We intend to achieve that by following the venerable dictum: pay the overhead for a feature only when you use it (which is, of course, easier said than done).

Consequently, we are focusing on optimizations to our basic algorithm. One optimization deals with the consistency problem mentioned above. Instead of requiring a full relational join of the XYZ values kept in relations at the p and q arcs, it allows us to access only the consistent values, by maintaining a few additional pointers. Many such optimizations are under investigation. The other techniques involve dynamically trading a part of our asset, the parallelism, for a reduction in overhead. These may lead to a tunable process model, which generates more or less parallelism depending on the runtime conditions in the parallel system. Throttling techniques which take over when the system memory starts to overflow are also being developed.

A major source of improvement will come from static analysis of given programs. Logic Programming is a very high level language, and our process model is also a more general one in that it attempts to handle different kinds of parallelism at once. In any particular clause the full generality of the basic algorithms is not usually needed. Compilation gives us an opportunity to handle each clause and each call differently, and static analysis should provide information useful to simplifying code for individual clauses. The kind of information we have found useful includes: a bound on the number of 'embedded' variables returned by a call, the variables that a given variable 'depends on' at a given point in the program, etc.

An interpreter for the ROPM has been implemented. It runs on an ALLIANT FX/8 (a shared memory machine), and on an Intel hypercube with 32 nodes. (The ROPM is implemented as a message passing system. A shared memory system provides additional opportunities for optimizations, and simplifies load distribution). It also runs on ORACLE, a multiprocessor simulation system that can simulate a variety of architectures. Related ongoing research includes
(1) Message-selection strategies, which significantly affect many performance metrics including the memory usage;
(2) load-balancing strategies for message-passing systems
(3) An 'operating system' (called the *chare kernel*) that can support many types of parallel symbolic computations including Logic programming.

Functional logic programming

The functional logic programming project at University of Illinois is involved in the investigation of languages that combine functional programming and logic programming paradigms into a unified framework. Our position, explained in [Redd85], is that *logic programming* means performing computations based on the notion of *solving* for variables. The notion of solving is not limited to predicates, it can be applied to functional expressions as well. The operational mechanism used for solving for variables in functional expressions is called *narrowing* [Fay79, Hull80, Redd85]. Narrowing essentially involves performing function application using unification for parameter passing, rather than pattern matching.

For example, consider a function for appending lists, defined by the equations

append(nil, Y) -> Y
append(A.X, Y) -> A.append(X,Y)

In a conventional implementation of functional programming, such a definition can only be used for evaluating applications of append to *ground* (variable-free) terms. But, in an implementation using the narrowing mechanism, append can also be applied to *nonground* terms. The variables in the argument terms are instantiated by unifying the arguments with the formal parameter terms on the left hand sides of the definition equations. For example, the evaluation of the function application append(L, M) can be unified with the left hand sides of both the equations. Each such evaluation yields a pair of results <s, e> where s is a substitution on the initial expression, and e the result of evaluation using the substitution. For the expression append(L, M), the first step of evaluation produces two such pairs:

1. substitution: L → nil, result: M
2. substitution: L → A.X, result: A.append(X, M)

Further evaluation of the second result term again yields two pairs:

2.1. substitution: L → A.nil, result: A.M
2.2. substitution: L → A.A'.X, result: A.A'.M

This then makes up the narrowing search space, and traversing it in a complete manner produces all possible solutions. Using backtracking to traverse the search space, though incomplete, provides an operationally viable method to enumerate the solutions, one by one. Each "solution" answers the question "for what instantiation of the variables does the initial expression reduce to a value, and what is its value under that instantiation?"

A functional logic language based on these ideas, called Scope, is being used as the *specification language* in the FOCUS program derivation system discussed later. In addition to the usual functional programming constructs, it contains 4 constructs which facilitate the introduction of logical variables in expressions and solving for them. These are

1. if p then e
2. e ; e'
3. forsome(X) e
4. unionover(X) s

where p is boolean valued expression, e, e' are any expressions, and s is a set-valued expression. (The terminology "set" is historical [Turn81], but these are in reality multisets, i.e., with possible duplicates). The first expression denotes the value of e if the expression p is true, and is undefined otherwise. The second construct denotes the choice of e and e'. If any one of them is defined, then it denotes the value of the defined expression. If both of them are defined, but have different values, then the expression is erroneous. The third construct, similar to existential quantification in predicate logic, means the value of e for some instantiation of X that makes e defined. Again, if e has different values for different instantiations of X, then the expression is erroneous. The fourth construct yields a set of values rather than a single value. It yields the union of all set values of s for all instantiations of X. The formal semantics of this language together with a set of examples illustrating its use may be found in [Redd87a].

The second aspect of our research is in using the narrowing mechanism for *transforming specifications into efficient programs.* We are constructing an interactive program transformation system called FOCUS [Redd87b] for this purpose. Most programs that can be expressed abstractly and concisely using the logical features of Scope have poor performance. (For example, maximum of a list can be expressed as its largest element, and sort of a list can be expressed as an ordered permutation of it). The FOCUS system allows the user to use such programs as specifications and transform them step by step into efficient programs.

The FOCUS system structures program derivations as trees, and uses a tree editor called XTED for its user interface. The program specification is entered in the root node of the derivation tree. Then nodes are created for each function that the user wishes to transform. Selecting a function to transform (together with its specification) is called *focusing.* A tree structure follows from the fact that in transforming a function, subsidiary foci may be needed for transforming its subexpressions. Focusing is also performed on *properties* of functions which need to be proved and then used in transformations. A number of transformation and deduction operations are provided to manipulate expressions and to prove theorems. The most often used transformation operations are *simplification* (evaluation without instantiation of any variables), *expansion* (evaluation with instantiation), and *recursing* or *application-introduction* (replacing instances of function specifications by function applications, or invoking inductive hypotheses). The deduction operations closely correspond to natural deduction inference rules.

The use of the FOCUS system operations ensures that the derived programs are equivalent to the original specifications. Moreover, the system also documents all the derivation activity performed in each focus node. Such a documentation is called a *derivation script.* When a specification is altered, the scripts off all the nodes are *replayed* to redo the derivation automatically. In this use, the FOCUS system acts much like a compiler, except that the optimizations performed are not preprogrammed by the compiler-writer, but programmed by the user himself in earlier derivation effort. The user is alerted to the situations where the replay *breaks* (i.e., where the old script is not applicable to the new specification or where it produces significantly different results from the earlier derivation). These situations are then handled by rederivation. Replaying is also used to modify a derivation. The system keeps track of the dependencies between focus nodes. When the derivation of a node is changed, the scripts of the other dependent nodes are replayed. Using the replay facility, changes can be propagated through entire programs semi-automatically. Thus, the FOCUS system supports the *automation-based software development* paradigm widely recognized to be the future software technology [Balz83].

References

[Balz83] R. Balzer, E. Cheatham, and C. Green. " Software technology in the 1990's: Using a new paradigm," *Computer*, Vol. 16, No. 11, Nov. 1983, pp. 39–45.

[Cone83] J. S. Conery. "The And/OR Process Model for Parallel Interpretation of Logic Problems", Ph.D. Thesis, University of California, June 1983.

[Fay79] M. Fay. "First order unification in an equational theory," *Fourth Workshop on Automated Deduction*, pp. 161–167, 1979.

[GoWa88] M. M. Gooley and B. W. Wah. "Efficient Reordering of Prolog Programs", to appear in *Proc. of 4th International Conference on Data Engineering*, Los Angeles, California, Feb. 1988.

[HaCH87] B. Hausman, A. Ciepielewski, and S. Haridi. "OR-parallel Prolog made efficient on shared memory multiprocessors," *Proc. of the 1987 International Symposium on Logic Programming*, San Francisco, California, 1987

[Hull80] J.-M. Hullot. "Canonical forms and unification," *Conference on Automated Deduction*,1980, pp. 318–334.

[Kale87a] L. V. Kalé. "Parallel execution of Logic Programs: the REDUCE-OR process model", *Proc. of Fourth International Conference on Logic Programming*, May 1987.

[Kale87b] L. V. Kalé. "'Completeness' and 'Full Parallelism' of Parallel Logic Programming Schemes", *Proc. of the 1987 Symposium on Logic Programming*, San Francisco, California.

[KaPS88] L. V. Kalé, D. A. Padua, and D. C. Sehr. "OR-parallelism in standard sequential Prolog," forthcoming.

[KaWa84] L. V. Kalé and D. S. Warren. "A Class of Architectures for Prolog Machine", *Proc. of the Conference on Logic Programming*, Uppsala, Sweden, July 1984, pp. 171–182.

[Li85] G. J. Li. "Parallel Processing of Combinatorial Search Problems", Ph.D. Thesis, School of Electrical Engineering, Purdue University, Dec. 1985.

[LiWa85] G.-J. Li and B. W. Wah. "MANIP-2: A Multicomputer Architecture for Evaluating Logic Programs", *Proc. of International Conference on Parallel Processing*, Aug. 1985, pp. 123–130.

[LiWa86a] G.-J. Li and B. W. Wah. "Coping with Anomalies in Parallel Branch-and-Bound Algorithms", *IEEE Transactions on Computers*, Vol. C-34, No. 6, June 1986, pp. 568–573.

[LiWa86b] G.-J. Li and B. W. Wah. "How Good are Parallel and Ordered Depth-First Searches?", *Proc. of International Conference on Parallel Processing*, Aug. 1986, pp. 992–999.

[LiWa86c] G.-J. Li and B. W. Wah. "Multiprocessing of Logic Programs", *Proc. of International Conference on Systems, Man and Cybernetics*, Oct. 1986, pp. 563–567.

[Redd85] U. S. Reddy. "Narrowing as the operational semantics of functional programs," *Symposium on Logic Programming*, Boston, Mass., 1985.

[Redd87a] U. S. Reddy. "Functional logic languages, Part I," in *Graph Reduction*, J. H. Fasel and R. M. Keller (eds.), Springer-Verlag Lecture Notes in Computer Science, Vol. 279, pp. 401–425, 1987.

[Redd87b] U. S. Reddy. *The FOCUS program derivation system: project report*, University of Illinois, 1987.

[Rous75] P. Roussel. *Prolog, manuel de reference et d'Utilisation*, Univ. of Marseilles, France, 1975.

[Turn81] D. A. Turner, *KRC Language Manual*, University of Kent, UK, 1981.

[Wah87] B. W. Wah. "Guest Editor's Introduction: New Computers for Artificial Intelligence Processing", *Computer*, Vol. 20, No. 1, Jan. 1987, pp. 10–15.

[WaLi86a] B. W. Wah and G.-J. Li. "A Survey of Special Purpose Computer Architectures for AI", *SIGART Newsletter*, April 1986, pp. 28–46.

[WaLi86b] B. W. Wah and G.-J. Li. "Tutorial in Computers for Artificial Intelligence Applications", *IEEE Computer Society Press*, May 1986.

[WaLi88] B. W. Wah and G.-J. Li. "Design of Multiprocessing Systems for Artificial Intelligence Applications", to appear in *Transactions on System, Man, and Cybernetics*, 1988.

[Warr87] D. H. D. Warren. "OR-parallel execution models of Prolog," *Proc. of the 1987 International Joint Conference on Theory and Practice of Software Development*, Pisa, Italy, Springer-Verlag, 1987, pp. 243–259.