LOGIC PROGRAMMING NEWSLETTER

5

WINTER 83/84

CONTENTS

Short Communications by: Alan M. Frisch, B. Canet, Earl Fogel, F. Kluźniak, Ian Watson, James F. Allen, Jan Sebelik, Kenneth M. Kahn, Leon Sterling, L. Ungaro, Mark Giuliano, Maarten van Emden, Mohd. Zahran Halim, O. Ridoux, R. A. Corlett, S. J. Todd, S. Szpakowicz, Y. Bekkers.

Community News & Events

New Journals & Books

Abstracts

EDITOR

Professor LUÍS MONIZ PEREIRA
Departamento de Informática
Universidade Nova de Lisboa

PUBLISHED

UNIVERSIDADE NOVA DE LISBOA
Centro de Informática
Faculdade de Ciências e Tecnologia
2825 Monte da Caparica — Portugal
☎ 245 4214 · 245 4987 · 245 4464 · 245 5299
245 5642 · 245 5643 · 245 5644 · 245 5645
Telex: 14542

ACKNOWLEDGEMENT

This publication has been partially subsidized by Junta Nacional de Investigação Científica e Tecnológica (JNICT), Av. D. Carlos I, 126, Lisboa, Portugal.

COLLABORATION

Thanks are due to ALEXANDRE SILVA for help with the address list.

SUBMISSIONS

Send to the editor, typed or printed, if possible double spaced, one side of the paper only.

CONTRIBUTIONS

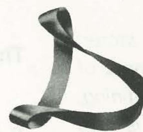
All those who can contribute please do so, by sending at least US$6.00 or equivalent, per number received, to the editor, payable to the Centro de Informática da Universidade Nova de Lisboa.

DESIGN, TYPESETTING AND PRINTING

Serviços Gráficos da Universidade
Nova de Lisboa
Av. Miguel Bombarda, 20-1.º
1000 Lisboa, Portugal

EDITOR'S FOREWORD

NEW GENERATION COMPUTING
An International Journal on Fifth Generation Computers
Vol. 1 No. 1 1983
Editor-in-Chief T. Moto-oka
Associate Editor K. Fuchi



OHMSHA, LTD. Tokyo Osaka Kyoto

Springer-Verlag Tokyo Berlin Heidelberg New York

354 New Gener Comp ISSN 0288-3635

• *Logic Programming is not merely picking up speed. It is spreading like a phase change to encompass the whole of computer science.*

A number of small and large companies and corporations are actively promoting research and development projects in the field.

Two new journals have been launched: "The Journal of Logic Programming" and "New Generation Computing", whose policy is expected to be one of cooperation. At the recent workshop in Portugal seventeen countries were represented, 80 people attended, and forty five papers were delivered.

A symposium is already scheduled for February, in Atlantic City, New Jersey, USA, and the next international conference is due in July, Uppsala, Sweden.

At least two collective books comprising a collection of papers are forthcoming: "Issues in Prolog Implementation", edited by John Campbell for Ellis Horwood Ltd. in England and "Logic Programming and its Applications", edited by David Warren and Michel van Caneghem for Ablex Publishing Corporation; both are expected at the beginning of the year. Ehud Shapiro is preparing a book with a selection of Prolog programs.

• *This newsletter has received another Portuguese government grant, about half the previous ones, since these grants are intended for scientific publications in the launching stage. Thus your financial support is increasingly more important (see note about contributions on this page).*

The newsletter also needs more collaboration, specially in the way of community news. New numbers will come out as soon as there is enough material to be published. Be sure to help it come out sooner. Act now.

• *Below we intended a photograph from the enjoyable "Logic Programming Workshop'83" which took place last June near Albufeira, Portugal. However, the photos we have are not good enough. Can you send us any for the next issue?*

short communications

AN OVERVIEW OF THE HORNE LOGIC PROGRAMMING SYSTEM

Alan M. Frisch, James F. Allen, Mark Giuliano

Computer Science Department
The University of Rochester
Rochester, New York 14627, USA

Introduction

HORNE is a PROLOG-based logic-programming system embedded in LISP. Programming in HORNE involves a careful mixture of logic programming and LISP programming.

Since the summer of 1981, HORNE has been continually evolving from its origin — HCPRVR (Chester, 1979). The main implementation is in FRANZ LISP on a VAX; a scaled-down implementation also exists in UCI LISP on a PDP-10. Today, HORNE bears little resemblance to HCPRVR; the primary similarity being the manner of embedding logic in LISP and the LISP-logic interface.

This paper conveys the flavor of the current state of the system — sacrificing detail and completeness for succinctness and simplicity. After a brief overview of the basic systems, this paper highlights those aspects of HORNE that differ from more conventional PROLOG systems. We assume that the reader is familiar with the rudiments of PROLOG and LISP. A more thorough account of the system may be found in the "Horne User's Manual" (Allen and Frisch, 1982).

The Basic System

Each clause in the HORNE data base is represented by a list stored on the property list of its procedure name, i.e., the predicate name of its sole positive literal. HORNE has a large set of facilities for defining, editing, deleting, and examining the data base of clauses. It is also capable of saving a subset of the database in a file and adding clauses to the data base from a file. Each clause may be labelled, enabling data base manipulations to be specified either by procedure names or by labels. All of these facilities are implemented as LISP functions.

The HORNE interpreter is a traditional PROLOG-style LUSH resolution theorem prover, also implemented as a LISP function. This function enables the user to specify that a certain number of proofs of a theorem are desired or that all proofs are desired. The prove function also provides a battery of tracing and debugging facilities.

HORNE has several built-in predicates that are typically found in PROLOG — among them a cut predicate, a predicate to test if a variable is bound, and one to test if a variable is bound to an atom. Many of the usual built-in predicates, such as those for arithmetic and input/output, are not needed since they are provided as built-in functions in LISP and can be accessed through the LISP-logic interface.

Embedding Logic in LISP

Those features of HORNE discussed so far are a straighforward LISP implementation of a logic-programming system. This section, and the next, point out that HORNE is a much tighter combination of logic and LISP.

A clause is encoded as a list of atomic formulae, the first of which is interpreted as the sole positive literal of the clause. An atomic formula is represented as a list whose first element, a LISP atom, is interpreted as the predicate name. The remaining elements of the list, each of which may be any S-expression, are the terms of the atomic formula. Through a mechanism not described here, certain S-expressions are interpreted as logical variables. For purposes of this paper, we will show variables as atoms whose print names begin with "?".

The use of LISP S-expressions as HORNE terms is a crucial design point. It is as if HORNE has only one function symbol, cons. Thus, the S-expression (f a) represents the term cons(f,cons(a,nil)), not the term f(a). From this viewpoint it is easy to see that the following terms unify with the most general unifier (m.g.u.) shown:

a	?x	with m.g.u. {?x/a}
(a)	?x	with m.g.u. {?x/(a)}
(a)	(?x)	with m.g.u. {?x/a}
(a b c)	(?x . ?y)	with m.g.u. {?x/a, ?y/(b c)}
(a b)	(a ?x . ?y)	with m.g.u. {?x/b, ?y/nil}
(a)	(a ?x . ?y)	does not unify
(a b)	(?x)	does not unify

This approach eliminates the need for special notation and mechanisms to handle lists and allows for a notation that appears to handle functions of a variable number of arguments. LOGLISP (Robinson and Sibert, 1981) takes the same approach to embedding logic in LISP; QLOG (Komorowski, 1982) does not.

The LISP-Logic Interface

We have already mentioned that HORNE'S logical terms are LISP S-expressions. There are two other mechanisms through which logic and LISP interact in HORNE.

A predicate name can be declared to be a LISP-predicate. Whenever the prover tries to prove a goal whose predicate is a LISP-predicate the LISP evaluator is used to test the goal. The goal fails if it evaluates to nil and succeeds otherwise. Before the LISP evaluation takes place, all logic variables in the expression are replaced with their current bindings.

HORNE has a built-in predicate, setv, that takes two terms. A goal with the setv* predicate succeeds if and only if the first term is equal to (i.e. unifies with) the result of the LISP-evaluation of the second term. All logic variables in the second term are replaced with their current bindings before the LISP-evaluation takes place.*

Typed Variables

Logical variables of HORNE may be restricted to range over a subset of the domain. Such a subset is called type. A variable, ?x, that ranges only over the type A is written as ?x:A. The advantage of adding types to the languages is that the theorem prover can reason about a set of individuals rather than backtracking over all individuals in the set. This is a sort of minimum commitment strategy.

The user may add clauses to the HORNE data base to specify a type theory that is used to derive sentences about relationships between types. Relationships of interest include an element of the domain belonging to a type, one type being disjoint from another, and one type being a subset of another. The theorem prover deals with typed variables solely during unification. Two variables, ?a:A and ?b:B unify to ?c:A B if and only if A intersects B. A variable, ?a:A, and a constant, b, unify to b if and only if b is an element of A. The unifier determines whether A intersects B or whether b is an element of A, by recursively calling the theorem prover to derive the desired fact from the type theory.

The idea of handling types totally during unification has been used by Reiter (1977). Under a strong restriction, called τ-completeness, he has proved his method sound and complete. Frisch (1983) has relaxed this restriction as far as possible with the following result: The method is complete if, and only if, the type theory can be rewritten as a set of logically equivalent Horn clauses. Hence, HORNE'S method of handling typed variables is sound and complete. A current concern of ours is with placing restrictions on the use of function signs in the type theory — if no such restrictions are made there may exist an infinite number of most general unifiers of two terms.

A great deal of efficiency in handling types is gained by precomputing the relationships among them. Rather than derive facts from the type theory when needed, much of the computation is done when the type theory is specified or updated and the results are stored.

Hashing

HORNE has a facility for specifying parameters to a hashing technique used to find quickly database clauses that match a goal literal with a certain predicate. This allows the hashing technique to be tailored to the characteristics of each predicate. A user is able to specify what terms, or substructures of terms, of the predicate are useful keys for indexing the clauses. In addition to the primary key, a sequence of secondary keys to be hashed in succession can also be specified. A user can also declare the size of the hashtable for each key. The HORNE hash function operates on ground atomic keys. Two special locations are reserved in each hashtable — one for keys that are non-atomic and one for keys that are variables. Hashing has no effect on the ordering of axioms; it simply eliminates futile attempts at unification.

The HORNE Compiler

The HORNE compiler translates HORNE clauses to LISP functions which are in turn compiled with the LISP compiler. Experiments have shown that the resulting code runs 2 to 4 times faster than the interpreted code. The primary factors accounting for this speed-up are an improved method of variable allocation and binding and a deeper level of LISP embedding. The HORNE compiler currently performs no optimizations of the type found in the Edinburgh PROLOG compiler (Warren, 1977).

Acknowledgements

We thank Dan Chester for starting this work off on the right track and supplying us with his HCPRVR, Rich Pelavin for his aggressive work on the compiler, and Rose Peet for her expert editorial assistance. The work reported in this paper was supported by National Science Foundation grants IST-8012418 and IST-8210564.

REFERENCES

ALLEN, J. F. and FRISCH, A. M. — "HORNE user's manual". Internal Report, Computer Science Dept., Univ. of Rochester, September, 1982.

CHESTER, D. L. — "Using HCPRVR". Internal Report, Dept. of Computer Science, Univ. of Texas at Austin, August, 1979.

FRISCH, A. M. — "Knowledge retrieval as specialized inference". Ph.D. thesis, Computer Science Dept., Univ. of Rochester, 1983.

KOMOROWSKI, H. J. — "QLOG - The software for PROLOG and logic programming". In K. Clark and S. A. Tarnlund (Eds.), *Logic Programming*. New York: Academic Press, 1982.

REITER, R. — "An approach to deductive question-answering". Report No. 3649, Bolt Beranek and Newman, September, 1977.

ROBINSON, J. A. and SIBERT, E. E. — "The LOGLISP's users manual". Technical Report, School of Computer and Information Science, Syracuse Univ., December, 1981.

WARREN, D. H. D. — "Implementing PROLOG - compiling predicate logic programs". Research Reports 39 and 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1977.

PURE PROLOG IN PURE LISP

Kenneth M. Kahn

UPMAIL, Department of Computing Science, Box 2059
Uppsala University, S-750 02 Uppsala, Sweden

In the Summer 1982 issue of the Logic Programming Newsletter, one finds a Pure Lisp in Pure Prolog by Pereira and Porto. The following is an interpreter for Pure Prolog written in Pure Lisp. It uses the primitive "Let" which is simply syntactic sugar. (Let ((x v-1) (y v-2)) body) *is equivalent to* ((lambda (x y) body) v-1 v-2).

```
(defun prolog (database) ;;a top-level loop for Prolog
  ;;reads a form, proves it, and then iterates
  (prove (list (rename-variables (read) '(O)))
         '((bottom-of-environment)) database 1)
  (prolog database))

(defun prove (list-of-goals environment database level)
  ;;proves the conjunction of the list-of-goals
  ;;in the current environment
  (cond ((null list-of-goals)
         ;;succeeded since there are no goals
         (print-bindings environment environment)
          ;;ask user if another possibility is wanted
         (not (y-or-n-p "More? (y or n) ")))
        (t (try-each database database
                     (rest list-of-goals)
                     (first list-of-goals)
                     environment level))))
```

```
(defun try-each (database-left database goals-left goal environment level)
 (cond ((null database-left)
        ()) ;;fail if nothing left in database
       (t (let ((assertion
                 (rename-variables (first database-left)
                                   (list level))))
            (let ((new-environment
                   (unify goal (first assertion) environment)))
              (cond ((null new-environment) ;;failed to unify
                     (try-each (rest database-left) database
                               goals-left goal
                               environment level))
                    ((prove (append (rest assertion) goals-left)
                            new-environment database
                            (addl level)))
                    (t (try-each (rest database-left) database
                                 goals-left goal
                                 environment level))))))))

(defun unify (x y environment)
  (let ((x (value x environment))
        (y (value y environment)))
    (cond ((variable-p x) (const (list x y) environment))
          ((variable-p y) (const (list y x) environment))
          ((or (atom x) (atom y)) (and (equal x y) environment))
          (t (let ((new-environment
                    (unify (first x) (first y) environment)))
               (and new-environment
                    (unify (rest x) (rest y) new-environment)))))))

(defun value (x environment)
  (con ((variable-p x)
        (let ((binding (assoc x environment)))
          (cond ((null binding) x)
                (t (value (second binding) environment)))))
       (t x)))

(defun variable-p (x) ;;a variable is a list beginning with "?"
  (and (list p x) (eq (first x) '?)))

(defun rename-variables (term level)
  (cond ((variable-p term) (append term level))
        ((atom term) term)
        (t (cons (rename-variables (first term) level)
                 (rename-variables (rest term) level)))))

(defun print-bindings (environment-left environment)
  (cond ((rest environment-left)
         (let ((variable (first (first environment-left))))
           (cond ((zerop (third variable)) ;;variable level
                  (print (second variable)) ;;variable name
                  (princ " = ")
                  (prinl (value variable environment)))))
         (print-bindings (rest environment-left)
                         environment))))

;;a sample database:
(setq db '(((father jack ken))
           ((father jack karen))
           ((grandparent (? grandparent) (? grandchild))
            (parent (? grandparent) (? parent))
            (parent (? parent) (? grandchild)))
           ((mother el ken))
           ((mother cele jack))
           ((parent  (? parent) (? child))
            (mother (? parent) (? child)))
           ((parent  (? parent) (? child))
            (father  (? parent) (? child)))))

;;the following are utilities

(defun first (x) (car x))
(defun rest (x) (cdr x))
(defun second (x) (cadr x))
(defun third (x) (caddr x))

(defun assoc (x list)
  (cond ((null list) list)
        ((equal x (first (first list))) (first list))
        (t (assoc x (rest x)))))

(defun y-or-n-p (mesage)
  (print message)
  (let ((response (read)))
    (cond ((eq response 'y) t)
          ((eq response 'n) nil)
          (t (y-or-n-p message)))))
```

Perhaps someone would like to try running the Lisp written in Prolog in this Prolog written in Lisp. I wish to acknowledge Par Emanuelson and Martin Nilsson for showing me inspiringly small Prolog interpreters in Lisp.

A SHORT NOTE ON GARBAGE COLLECTION IN PROLOG INTERPRETERS

Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro
Irisa / Inria, Rennes
FRANCE

Abstract

Any good marking algorithm for garbage collection in PROLOG starts the marking from all active goal statements. Our point is that when marking information which is accessed via an active goal statement kept in a backtrack point, it is incoherent to consider bindings which have been established later than the creation of this backtrack point.

Background

It is assumed that the reader is familiar with PROLOG interpreters. This short note follows Bruynooghe's articles on memory management [1] and garbage collection [2] in PROLOG interpreters.

Using the trail for marking

The state of a sequencial backtracting PROLOG interpreter represents a sequence of active goal statements by means of objects such as goal skeletons and bindings. The current bindings are those of the current goal statement, every earlier active goal statement is defined by considering its goal skeletons under a subset of the current bindings: namely the bindings which where current at the creation of the goal statement. Considering the goal skeletons of an earlier goal statement under the current bindings defines a "phantom" entity, hence useless accesses.

Therefore, to know the information which is accessed from a goal statement kept in a backtrack point, one should "undo" the bindings which are irrelevant to this goal. This is similar to what is done with the trail before resuming a backtrack point, but in a marking algorithm for garbage collection, the undoing should only be simulated to allow resolution to proceed. A special tag associated whith every binding can accomplish this.

Example

The program is

C1 : a(Y) <– c(Y,f(Z)), d(Z).
C2 : c(q(a),V).
C3 : d(f(a,T)) <– e(g(T)), f(W,U).
C4 : e(g(b)).

The question is

<– a(q(X)), b(X).

After four steps of execution, the state of the interpreter is as in figure 1. The active goal statements are in node III which is supposed to be a backtrack node, and in node V which is the current node. Using Bruynooghe's algorithm to mark information will result in marking garbage as useful memory cells. On the contrary if the trail is used to tag irrelevant bindings, the term f(a,T) will not be marked. The binding of variable Z to this term is irrelevant for the goal statement where variable Z occurs because this goal statement is older than the binding of Z to term f(a,T).

In this example Z and T are global variables in Warren's sense [3]. However, with our method, the binding of Z is proved to be a garbage, therefore it is destroyed. As a consequence, the global variable T is no longer accessible from any goal statement, and the binding of T and T itself are proved to be garbage.

Figure 2 shows what is remaining after collecting garbage.

Figure 1

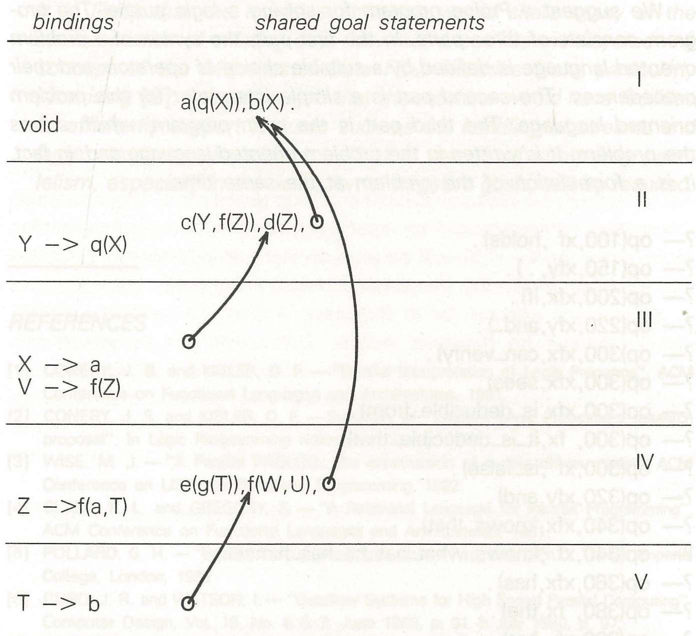
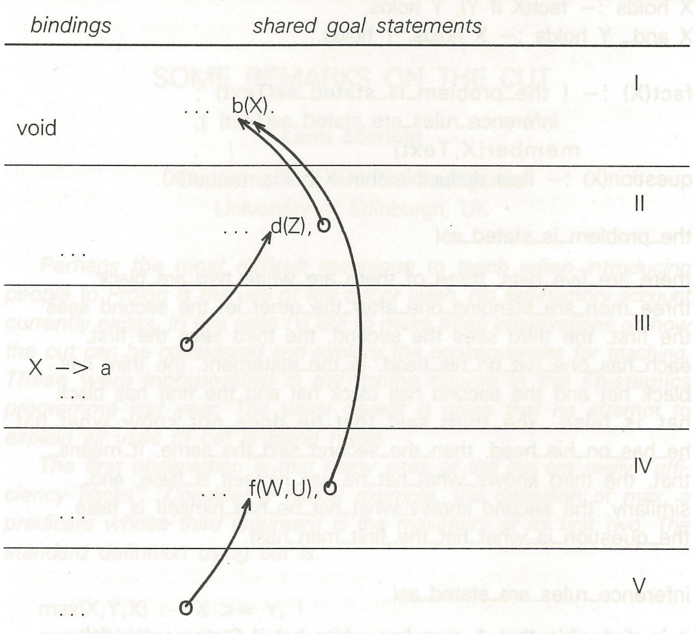


Figure 2



REFERENCES

[1] "The memory management of PROLOG implementations", M. Bruynooghe, in Logic Programming, eds. Tarnlund and Clark, Academic Press 1982.

[2] "A note on garbage collection in PROLOG interpreters", M. Bruynooghe, Proceedings of the First International Logic Programming Conference, Sept 14-17th 1982, Marseille, France.

[3] "Implementing Prolog-compiling logic programs. Vol. 1 and 2". D.H.D. Warren. D.A.I. Research Report, No. 39, 40. University of Edinburgh, 1977.

short communications

A PROBLEM DESCRIPTION IN PROLOG

Jan Šebelík

Institute for Application of Computing Technique in Control
Revoluční 24, Prague 1, Czechoslovakia

We suggest a Prolog program for solving a logic puzzle. The program consists of three parts. In the first part, the syntax of a problem oriented language is defined by a suitable choice of operators and their precedences. The second part is a simple interpreter for this problem oriented language. The third part is the main program, which solves the problem. It is written in the problem oriented language and, in fact, it is a formulation of the problem at the same time.

```
?— op(100,xf ,holds).
?— op(150,xfy, . ).
?— op(200,xfx,if).
?— op(220,xfy,and_).
?— op)300,xfx,can_verify).
?— op(300,xfx,sees).
?— op(300,xfx,is_deducible_from).
?— op(300, fx,it_is_deducible_that).
?— op(300,xf ,is_false).
?— op(320,xfy,and).
?— op(340,xfx,knows_that).
?— op(340,xf ,knows_what_hat_he_has_himself).
?— op(360,xfx,has).
?— op(380, fx,the).
?— op(380,xf ,hat).

X holds :— fact(X).
X holds :— fact(X if Y), Y holds.
X and_ Y holds :— X holds, Y holds.

fact(X) :— ( the_problem_is_stated_as(Text) ;
             inference_rules_are_stated_as(Text) ),
           member(X,Text)                         .
question(X) :— it_is_deducible_that X holds, output(X).

the_problem_is_stated_as(

there_are_five_hats. three_of_them_are_white_two_are_black.
three_men_are_standing_one_after_the_other_ie. the second sees
the first. the third sees the second. the third sees the first.
each_has_one_hat_on_his_head. ie_the_statement. the third has
black hat and the second has black hat and the first has black
hat is_false. the_third_said_that_he_does_not_know_what_hat_
he_has_on_his_head. then_the_second_said_the_same. it_means_
that. the third knows_what_hat_he_has_himself is_false. and_
similarly. the second knows_what_hat_he_has_himself is false.
the_question_is_what_hat_the_first_man_has).

inference_rules_are_stated_as(

it_is_deducible_that A_man has white hat if Statement is_false
and_ Statement is_deducible_from A_an has black hat. similarly.
A_man has white hat is_deducible_from An_assumption if Statement
is_false and_ Statement is_deducible_from A_man has black hat
and An_assumption.
A_man knows_what_hat_he_has_himself is_deducible_from An_assump
tion if A_man can_verify An_assumption and_ A_man has certain
hat is_deducible_from An_assumption.
Man1 can_verify Man2 has Some hat if Man1 sees Man2. A_man
can_verify Fact1 and Fact2 if A_man can_verify Fact1 and_ A_man
can_verify Fact2.
Everything is_deducible_from An_assumption if An_assumption
is_false.
and_it_is_all_for_solving_the_problem).
```

After starting the program with the goal

```
?— question( the first has What hat ).
```

the program will answer: the first has white hat .

DATA DRIVEN LOGIC PROGRAMMING: A RESEARCH PROPOSAL

Mohd. Zahran Halim [1], Ian Watson

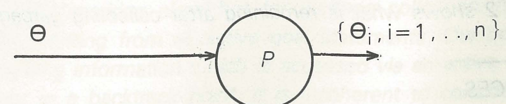
Department of Computer Science
University of Manchester, UK

There are already a number of approaches put forward for the parallel execution of logic programs [1-5] All the schemes proposed involve the creation of processes that may proceed concurrently and communicate with one another via messages. A major problem encountered when trying to realise such message-based schemes is that of simultaneous access to common memory by processes executing in parallel. This leads to a degradation in performance and results in an inefficient use of resources.

The data flow approach to parallel computer architectures is an attempt to resolve the problems of multiprocessor machines by introducing a known communication requirement between processes and memory and to make use of processing resources only to perform useful computation [6, 7]. We propose to investigate execution models which are effectively data driven and to base the design of a parallel logic programming engine on such a model. An outline of a preliminary model which exploits OR-parallelism follows.

Model Outline

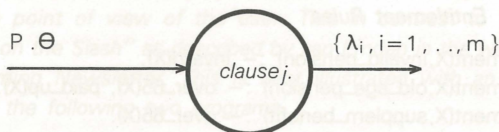
A goal <—P is activated by the arrival of a binding environment Θ. *The result of activating a goal is a stream of bindings* $\{\Theta_i, i=1,\ldots,n\}$ *representing new binding environments resulting from alternative ways of solving P. Diagrammatically:*



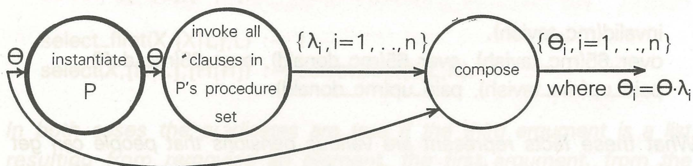
[1] On study leave from the University Sains Malaysia, Penang, Malaysia.

short communications

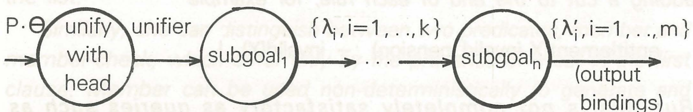
Or-parallelism is exploited by sending P.Θ *to all clauses whose head predicate matches that of* P. *The result of invoking a clause is a stream of output bindings* $\{\lambda_i, i=1,\ldots,m\}$ *each representing an alternative way of solving* P *using the clause:*



Thus, a more detailed view of goal activation is



In the above diagram, $\{\lambda_i, i=1,\ldots,n\}$*, represents the merged stream of alternative bindings returned by each clause invoked. The "compose" function adds the bindings returned to* Θ *(whether explicit composition of substitutions is performed is a question to be resolved later).*



Notice that each subgoal is activated in turn i.e. no attempt is made in this model to exploit AND-parallelism. However, it should be noted that activation of subgoal_i *does not have to wait for the completion of* subgoal_{i-1} *i.e. it can be activated as soon as a stream element arrives. In the case where the clause is an assertion, either the empty stream or the output bindings of the unifier is returned, depending on whether unification is successful.*

The essence of the model is that it treats a logic program as defining a program graph where nodes are activated upon arrival of their arguments. Viewed this way, the creation of child processes for searching independent branches of the search tree and the subsequent communication of results back to the parent reduces to a function call to each alternative subprogram (clause) with appropriate call/return mechanisms.

Further research will look into

— extensions of the model

(i) to include negation by failure to prove

(ii) to explore restricted forms of AND-parallelism (eg. detection and parallel activation of independent goals)

(iii) to consider lower level functions (say by decomposing the functions outlined above) — this approach would imply that logic programs would be compiled into a program graph of lower level functions. However, more potential parallelism may be discovered eg. unification could possibly be compiled to match some arguments in parallel.

— implications in the design of machine architecture

It is likely that the machine structure would share many of the characteristics of existing data flow machines. The multilayered ring concept of the Manchester Data Flow Machine looks particularly promising [6]. *It is envisaged that the machine would have facilities to efficiently support features such as the handling of streams and large structures. In addition, effective means of controlling parallelism, especially in the activation of negative goals, will be required.*

REFERENCES

[1] CONERY, J. S. and KIBLER, D. F. — "Parallel Interpretation of Logic Programs". ACM Conference on Functional Languages and Architectures, 1981.

[2] CONERY, J. S. and KIBLER, D. F. — Summary of "Efficient Logic Programs:a research proposal". In Logic Programming Newsletter, Spring, 1981.

[3] WISE, M. J. — "A Parallel PROLOG: the construction of a data driven model". ACM Conference on LISP and Functional Programming, 1982.

[4] CLARK, K. L. and GREGORY, S. — "A Relational Language for Parallel Programming". ACM Conference on Functional Languages and Architectures, 1981.

[5] POLLARD, G. H. — "Parallel Execution of Horn Clause Programs". Ph.D. Thesis. Imperial College, London, 1982

[6] GURD, J. R. and WATSON, I. — "Dataflow Systems for High Speed Parallel Computing". Computer Design, Vol. 19, No. 6 & 7. June 1980, p. 91 & July 1980, p. 97.

[7] WATSON, I. and GURD, J. R. — "A Practical Dataflow Computer". IEEE Computer, Vol. 15, No. 2, feb 1982.

SOME REMARKS ON THE CUT

Leon Sterling

Department of Artificial Intelligence
University of Edinburgh, UK

Perhaps the most difficult technique to teach when introducing people to Prolog is the use of the cut or slash. No satisfactory account currently exists. In this note I'd like to make three observations on how the cut can be considered and explore the consequences for teaching. These were incorporated in my Prolog course in the Epistemics programme last year. The usual caveat is given that no attempt to explain all uses of cut is being made.

The first observation is that many uses of the cut are really "efficiency hacks". Consider a typical example, the definition of max, a predicate whose third argument is the maximum of its first two. The standard definition using cut is

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,X).
```

Here the equivalent program without the cut is

```
max(X,Y,X) :- X >= Y.
max(X,Y,X) :- X < Y.
```

The purpose of the cut in the first program is to obviate the need of the second comparison, X =< Y. *There are also important considerations of saving space by removing choice points, but that won't be considered in this note. But look at the cost in order to gain a little efficiency. The first program has to have its 2 clauses in the order stated. It is dependent on the Prolog order of reading clauses, which is not true of the second program.*

Further, the second program is much clearer. The logic is in a form that would enable reasoning about the program in some larger context. Come the millenium, and an intelligent compiler which knows that either X > Y *or* X =< Y, *will be able to make an appropriate optimization for itself producing the second program from the first.*

What I am arguing for in this case is that the second program should always be the one taught. The conditions by which the predicate is chosen should be made explicit. At a later stage when one's program is working and super efficiency is desired, then cuts can be introduced to make the program go faster, but that is only at the last stage, and the cut in the first program should be viewed in that light.

Let us consider another classic example, computing factorials. The totally naive program is

```
fact(0,1).
fact(N,Fact) :- M is N - 1, fact(M,P), Fact is N * P.
```

This program is "wrong". It will correctly anwer the query ?- fact(4,X). *with* X=24, *but asking for another solution would involve the program becoming lost down an infinite branch of the search tree. One solution is to change the first clause to*

```
fact(0,1) :- !.
```

This removes the problem when you ask for alternatives to the above query, but still gets lost on a query such as ?- fact(-1,X). *A correct program is*

```
fact(0,1).
fact(N,Fact) :- N > 0, M is N -1, fact(M,P), Fact is N * P.
```

The difference is outting in the explicit condition. This addition makes the code correct, more robust and easier to teach. It is true that the code might be made more efficient by adding a cut. But in this case the cut is a minor consideration. The program would be made much more efficient by adding an extra argument as an accumulator and making fact tail-recursive. Thus this is not a good example for explaining the use of cut.

Making conditions explicit corresponds well with Shapiro's suggestion of guarded clauses in concurrent Prolog [Shapiro 83]. Here the explicit conditions would naturally form the guard.

This leads to the second observation about the use of cut. A common need when writing logic programs is to specify a set of mutually exclusive cases. Consider the following small database in a social security office in outback Scotland.*

```
/*      Entitlement Rules       */

entitlement(X,invalid_pension) :- invalid(X).
entitlement(X,old_age_pension) :- over_65(X), paid_up(X).
entitlement(X,supplem_benefit) :- over_65(X).
entitlement(X,nothing).

/*      Facts      */

invalid(mc_tavish).
over_65(mc_tavish). over_65(mc_donald). over_65(mc_duff).
paid_up(mc_tavish). paid_up(mc_donald).
```

What these facts represent are various pensions that people can get under particular conditions, such as being an invalid or over 65 years old. The conditions are mutually exclusive and if none apply the person receives nothing by the last entitlement rule. The problem with the rules as they stand are that they don't express that the conditions are mutually exclusive. So the query ?- entitlement(mc_tavish,X). *will give 3 alternatives for X, namely* X=invalid_pension, X=old_age_pension *and* X=supplem_benefit. *Stopping alternatives can be achieved by adding a cut to the end of each rule, for example*

```
entitlement(X,invalid_pension) :- invalid(X), !.
```

But this is not completely satisfactory as queries such as ?-entitlement(mc_tavish,nothing). *succeed. To avoid that particular problem, one can make the conditions explicit as advocated above. The rules would then be*

```
entitlement(X,invalid_pension) :- invalid(X).
entitlement(X,old_age_pension) :-
        over_65(X), paid_up(X), not(invalid(X)).
entitlement(X,supplem_benefit) :-
        over_65(X), not(paid_up(X)), not(invalid(X)).
entitlement(X,nothing) :-
        not(invalid(X)), not(over_65(X)).
```

The not contitions seem a little unnatural, however. There are also the usual difficulty of using negation in Prolog with non-ground terms. There ought to be some higher-level predicate which can express that certain conditions are mutually exclusive, and it be left to a compiler to incorporate the consequences efficiently.

There is another way of avoiding the success of ?- entitlement(mc_tavish, nothing). *That is, by writing each rule as follows.*

```
entitlement(X,Y) :- invalid(X), !, Y = invalid_pension.
```

* This is a modified example originally from Sam Steel.

But again a higher-level predicate would be much preferable, and the use of cut here should be taught as an implementation trick, not a feature.

The final observation is that it can be useful to view the addition of a cut to a program as changing the intention of the program, at least from the point of view of the user. This in contrast to "Warren's Doctrine on the Slash" as described by van Emden in the winter Logic Programming Newsletter. This is best illustrated with an example. Consider the following two programs.

```
select(X,[X|L],L).
select(X,[H|L],[H|R]) :- select(X,L,R).

select_first(X,[X|L],L) :- !.
select(X,[H|L],[H|R]) :- select_first(X,L,R).
```

In both cases the predicates are true if the third argument is a list resulting from removing an element, the first argument, from the second argument, a list. The difference comes when you use the predicates in a larger context. The select predicate we standardly use non-deterministically to find an element of a list with a particular property and keep the rest of the list for further computation. The predicate select-first cannot be used in the same way. It can however be used apparently non-deterministically to choose the first element of the list.

Similarly, one can distinguish between two predicates, member and member-check, which differ only by the presence of a cut in the first clause. Member can be used non-deterministically to generate and test, and further will succeed more than once if there are multiple copies of an element in a list. Member-check only succeeds once in the above case, and cannot be used non-deterministically. Each variant of the predicate has its use. For example in the standard utilities we use, both predicates are available.

Whether the program variants, i.e. with or without cuts, can be said to have different meanings is beyond the scope of this note. It relates however to a question whether two versions of a compiled predicate with different mode declarations can be said to be different. The point is, however, that when using the programs they are considered as two different programs.

The consequences for teaching cut related to this last observation is that cuts should be considered in the context of a particular use of a program. Again this is analogous to setting mode declarations for compiling programs. Once that program is used for any different purpose, all cuts present should be reconsidered as being appropriate for the new use.

REFERENCE

[SHAPIRO, 83]
Shapiro, E. "A Subset of Concurrent Prolog and its Interpreter". Report TR-003, ICOT, Japan, 1983.

A BASIC INTERPRETER FOR COROUTINING

R. A. Corlett, S. J. Todd

Marconi Research Centre
West Hanningfield Road
Great Baddow
Chelmsford, Essex CM2 8HN
United Kingdom

1. The interpreter

The advantages of coroutining processes have been well documented elsewhere [1, 2, 3, 4]. This note describes a simple interpreter written in Prolog to enable selective coroutining of goals within a Prolog program. In the program below the goal 'system(P)' succeeds if P is a call to a built-in predicate and is used to avoid errors from accessing the body of system predicates.

```
suspend.
system(//(X,Y)).

?- op(251,xfy,//).

true//P :- !, call(P).
(true,P)//Q :- !,(P//Q).
(suspend,P)//Q :- !, (Q//P).
(P;Q)//R :- !,(P//R ; Q//R).
((P;Q),R)//S :- !,(P,R//S ; Q,R//S).
((P;Q),R)//S :- !,(P,Q,R//S).
((P//Q)//R) :- !,(P//Q//R).
(P,Q)//R :- !,(system(P) -> call(P),(Q//R);
                          clause(P,S),(S,Q//R)).
P//Q :- !,(system(P) -> call((P,Q));
                        clause(P,R),(R//Q)).
```

The infix operator, '//', has a declarative reading of 'and', and coroutines the processes it separates. Transfer of control out of the active process is effected by the explicit inclusion of suspension points defined by the presence of a 'suspend' goal. This basic approach to coroutining is useful in problems that are amenable to the type of analysis carried out in [1].

2. An example

An example of a problem for which the explicit insertion of suspension points is appropriate is the classic Eight Queens problem for which a simple sequential solution, from [2], is given below:

```
solution(Perm) :-
    permutation([1,2,3,4,5,6,7,8],Perm),safe(Perm).

permutation(L,[Q|M]) :-
    remove(Q,L,L1),permutation(L1,M).
permutation([],[]).
```

```
remove(X,[X|L],L).
remove(X,[Y|L],[Y|M]) :- remove(X,L,M).

safe([Queen|List]) :-
    nodiagonal(Queen,List,1),safe(List).
safe([]).

nodiagonal(Q1,[Q2|List],N) :-
    noattack(Q1,Q2,N),N1 is N + 1,
    nodiagonal(Q1,List,N1).
nodiagonal(Q1,[],N).

noattack(Q1,Q2,N) :-
    Q1 > Q2, Diff is Q1 - Q2, Diff = N.
noattack(Q1,Q2,N) :-
    Q2 > Q1, Diff is Q2 - Q1, Diff = N.
```

Gallagher [1] analyzes the inefficiencies in this solution and suggests a "transformation" of the program to simulate a coroutining solution. An implementation of his solution using our interpreter is given below:

```
solution(Perm) :-
    permutation([1,2,3,4,5,6,7,8],Perm)//safe(Perm).

permutation(L,[Q|M]) :-
    remove(Q,L,L1),suspend,permutation(L1,M).
permutation([],[]).

remove(X,[X|L],L).
remove(X,[Y|L],[Y|M]) :- remove(X,L,M).

safe([Queen|List]) :-
    suspend,(nodiagonal(Queen,List,1)//safe(List)).
safe([]).

nodiagonal(Q1,[Q2|List],N) :-
    noattack(Q1,Q2,N),N1 is N + 1,
    suspend,nodiagonal(Q1,List,N1).
nodiagonal(Q1,[],N).

noattack(Q1,Q2,N) :-
    Q1 > Q2, Diff is Q1 - Q2, Diff = N.
noattack(Q1,Q2,N) :-
    Q2 > Q1, Diff is Q2 - Q1, Diff = N.
```

Advantages of this approach are:

(i) Our interpreter provides a more general implementation of coroutining for this class of problem.

(ii) Our program has a declarative reading that is identical to the original program.

(iii) The interpreter need only be invoked where the additional control is required.

(iv) The interpreter can be tuned to increase efficiency in particular problems.

3. Tuning the interpreter

There are several ways in which the interpreter can be tuned:

(i) By removing unused clauses from the interpreter, for example in the Eight Queens problem those that handle disjunctions are not required.

(ii) By declaring certain predicates to be system predicates, we can limit the depth to which the interpreter operates and executes subgoals of coroutining processes, that do not include suspends, in native mode. By declaring 'remove' and 'noattack' as 'system' we halved the execution time in the above program.

(iii) By including application specific instantiations of interpreter clauses: this becomes equivalent to Gallagher's approach.

```
suspend.
?- op(251,xfy,//).
((P//Q)//R) :- !,(P//Q//R).
solution(Perm) :-
    permutation([1,2,3,4,5,6,7,8],Perm)//safe(Perm).

permutation(L,[Q|M])//R :-
    remove(Q,L,L1),(R//permutation(L1,M)).
permutation([],[])//Q :- !,call(Q).
remove(X,[X|L],L).
remove(X,[Y|L],[Y|M]) :- remove(X,L,M).
safe([Queen|List])//P :-
    P//(nodiagonal(Queen,List,1)//safe(List)).
    safe([]).

nodiagonal(Q1,[Q2|List],N)//P :-
    noattack(Q1,Q2,N),N1 is N + 1,
    (P//nodiagonal(Q1,List,N1)).
nodiagonal(Q1,[],N)//P :-!,call(P).

noattack(Q1,Q2,N) :- Q1 > Q2, Diff is Q1 - Q2, Diff = N.
noattack(Q1,Q2,N) :- Q2 > Q1, Diff is Q2 - Q1, Diff = N.
```

4. Limitations

The interpreter as given makes no provision for 'if then-else' constructs or cuts. The interpreter might also be extended to handle multiple levels of active coroutined processes, but this requires a more detailed analysis of the flow of control that is in fact required.

REFERENCES

[1] GALLAGHER, J. — "Simulating Coroputining For The 8 Queens Problem", Logic Programming Newsletter, No. 3.

[2] CLARK, K. and McCABE, F. — "The control Facilities of IC-Prolog", in Expert Systems in the Microelectronic Age, Edinburgh Univ. Press, 1979.

[3] PEREIRA, L. M. — "Logic Control With Logic", Proceedings of the First International Logic Programming Conference, 1982.

[4] PEREIRA, L. M. and MONTEIRO, L. F. — "The Semantics of Parallelism and Co-routining in Logic Programming", in Mathematical Logic in Computer Science, North-Holland, 1981.

short communications

A NOTE ON DEFINITE CLAUSES

Maarten van Emden
University of Waterloo
Ontario, Canada

Several recent publications have used "definite clause" as a synonym for "Horn clause". This was certainly not the intention when the former concept was introduced.

A Horn clause is a clause with at most one positive literal. A definite clause is a Horn clause with at least one positive literal.

It seems to me that no synonym is needed for "Horn clause", whereas "definite clause", as defined here, can be very useful.

EDISON IN PROLOG

Ron Hayter
Department of Computer Science
University of British Columbia
Canada

Edison [1] [2] is a new multiprocessor language designed by Brinch Hansen. It is descended from Pascal, Concurrent Pascal, and Modula, and it was designed to be suitable both for teaching concurrent programming and for writing real-time programs. As an exercise in the use of Prolog, I developed an experimental Edison system for a course in language implementation taught by Professor Harvey Abramson. The system consists of a compiler producing intermediate machine code, an interpreter, and a translator of that code into BCPL. This system was developed in less than one month.

The compiler is the largest component of the system. It is divided into three conventional passes: lexical, syntactic, and semantic. The lexical pass organizes the characters of the Edison program into tokens. The syntactic pass builds a parse tree from the list of tokens. Finally, the semantic pass transforms the parse tree into code for a hypothetical intermediate machine.

Each of these passes was written using definite clause grammars. The use of DCGs (together with the exceptionally clean design of Edison) made the writing of the compiler quite straightforward. The compiler is concise and, I hope, quite readable. However, it is also very poor at error recovery: it stops as soon as an error is detected. Since the purpose of the project was not to produce a production-quality compiler, this strategy was acceptable. It seems that to add reasonable error recovery would add considerably to the size of the compiler and would seriously reduce its readability.

Next, an interpreter for this intermediate machine was written, also in Prolog. DCGs were again used, this time to describe the effects of the machine instructions on the state of the machine. Unfortunately, the interpreter had to be abandoned before it was completed because it was able to run only for a short while before our local implementation of Prolog ran out of stack space. This problem could be solved by adding a tail-recursion optimization to the Prolog interpreter. Although the details of how to interpret Edison's multiprocessing statements have not been worked out, a coroutining mechanism, such as that in Epilog [3] [4], should make implementation easy.

When the idea of implementing an interpreter in Prolog had to be abandoned, another approach was taken. A Prolog program was written to translate the intermediate machine code into BCPL. This translation was very straightforward. A BCPL routine is defined for each Edison procedure and function in the program. Each machine instruction is simply translated into a call to a BCPL global procedure which implements the instruction. Edison processes map quite conveniently onto BCPL coroutines [5].

This Edison system implements the full language, except for separate compilation, but it is rather limited in capability. In the course of this project, most of the limits of our Prolog interpreter were reached, particularly internal table sizes and the stack size. As a result, only small programs (no more than about 100 lines) can be compiled, and it was not possible to complete a Prolog definition of an interpreter for the compiled code. However, despite the limitations of our Prolog, I was able to produce an experimental implementation of a "real" programming language in a short time. Prolog is a very powerful tool, especially with the addition of DCGs and logic control, for the implementation of programming languages.

REFERENCES

[1] HANSEN, P. B. — "Edison-A Multiprocessor Language", Software-Practice and Experience, Vol. 11, pp. 325-361, 1981.

[2] HANSEN, P. B. — "The Design of Edison", Software-Practice and Experience, Vol. 11, pp. 363-396, 1981.

[3] PEREIRA, L. M. — "Logic Control With Logic", Proceedings of the First International Logic Programming Conference, pp. 9-18, Marseille, 1982.

[4] PORTO, A. — "Epilog: A Language for Extended Programming in Logic", Proceedings of the First International Logic Programming Conference, pp. 31-37, Marseille, 1982.

[5] MOODY, K. and RICHARDS, M. — "A Coroutine Mechanism for BCPL", Software-Practice and Experience, Vol. 10, pp. 765-771, 1980.

IMPLEMENTATION OF LISPKIT LISP IN PROLOG

Earl Fogel
Department of Computer Science
University of British Columbia
Canada

Introduction

The project was the implementation in PROLOG of the LISPKIT (Henderson, 1980) version of LISP.

LISPKIT is a purely functional language, unlike most LISPs. Variables are instantiated only through the association of parameters and arguments, and functions are evaluated in the environment in which they were created, not the environment at the time they are called.

General approach

An input s-expression passes through lexical and syntactic analysis based on a Definite Clause Grammar (DCG) description of LISPKIT.

If the input expression is a valid LISPKIT form, a parse tree is produced by the syntactic analysis. This parse tree is then evaluated, and the resulting value is printed.

Lexical analysis

The input expression is broken down into a string of lexemes. which are: identifiers; integers; and parentheses.

These are passed on to the syntactic analyser in the forms:

LEXEME	FORMAT
identifiers	id(*id)
integers	int(*int)
parentheses	(and).

Syntactic analysis

All the valid LISPPKIT s-expressions are recognized, and a parse tree is produced.

For example:

(CONS e1 e2)

would parse as:

CONS(e1,e2)

Evaluation

The value of a constant is itself. The same holds for integers and the special identifiers NIL and T.

All other identifiers are evaluated by looking up their values in the environment list.

The LISPKIT built-in functions are grouped into two classes: those that evaluate their arguments before executing (EVAL type functions); and those that do not evaluate arguments before execution (NOEVAL type).

All the built-in functions are evaluated by applying a PROLOG version of the function to its arguments (or to a list of their values in the case of EVAL functions).

User-defined functions are all EVAL type. They can be called either by name (where the name has previously been definecd in a LET or a LETREC), or by explicitly giving a LAMBDA expression defining the function whenever it is called.

User functions are handled in a somewhat more complex fashion. The function (name or LAMBDA expression) is evaluated, and the resulting function definition is then applied to a list of the argument values for that particular function application.

In order to evaluate functions in the environment in which they were created, the entire environment at the time of a function definition is stored as part of that definition, along with a parameter list, and the LAMBDA expression itself.

Differences from LISPKIT

LISPKIT uses the period '.' in the definition of LET and LETREC. I omit it.

In this version of LISPKIT, LET or LETREC expressions may be used to define directly recursive functions, but only LETREC will handle indirect recursion. In Henderson's LISPKIT, LETREC must be used for both forms of recursion.

Conclusions

LISPKIT LISP has been implemented successfully in PROLOG. As we have at this time only a PROLOG interpreter, this LISPKIT runs more slowly than compiled LISPs.

Input and Output are not yet ideal. The input s-expression must be enclosed in double quotes and be followed by a period. The value that is output is printed with a very simpleminded pretty print function.

REFERENCES

GOEBEL, R. — "PROLOG/MTS User's Manual", TM80-2, UBC Dept. of Computer Science, Dec. 1980.

HENDERSON — "Functional Programming Application and Implementation", Prentice-Hall, 1980.

A NOTE ON PROLOG SYNTAX

F. Kluźniak, S. Szpakowicz

Institute of Informatics
Warsaw University, POLAND

Prolog is now undergoing a period of rapid growth: new implementations are making the language accessible to an increasing amount of computer users. For a lot of people — implementors included — Prolog-10 is the standard Prolog as far as syntax and most built-in predicates are concernd. It might not be realistic to expect that a completely different standard would ever gain wide acceptance, but it might not yet be too late for some revisions. We would like to draw attention to several aspects of this syntax which make is unnecessarily unpalatable.

1. Variable names

It is not a good idea to start variable names with capital letters, and other names with small letters. It should be the other way round: constants should stand out clearly from the enclosing text and the best way to achieve that is to capitalise them; variables are usually more numerous and writing them in small letters would save keystrokes and make programs look cleaner, variables usually play the part of common names and constants that of proper names.

Of course, infix functors would have to be capitalised, too. (We use "infix" as a generic name for "infix", "prefix" and "postfix" — calling them "fix functors" would perhaps be carrying it too far, but the term "operators" is manifestly misleading). The choice between
inherits_after(John, Jack).
and
Inherits_after(John, Jack).
is a matter of taste, but
?– who Inherits_after John.
would have to be mandatory.
?– Who inherits_after john .
only serves to suggest another meaning.

2. List notation

Special list notation instead of Marseille Prolog's original dotted lists seems to have been introduced only to please LISP (or rather POP) adherents and to avoid the minor technical problem of recognising whether a dot terminates a term. Such considerations are certainly not worth the drawbacks:

— *the intended reading is* x cons y , *so* x.y *seems more natural than* [X|Y]; *it also saves keystrokes;*
— John.Jack.NIL *does not save keystrokes, but is still easier to write and tells us more than* [john, jack]: *the result of binding* x *to* NIL *in* John.Jack.x *is more obvious than that of binding* X *to* [] *in* [john, jack|X] *(quite different than in* [john, jack, X] *);*
— *commas are used to separate calls, and procedure parameters, and term arguments and list elements: given sufficient nesting, it is a conscious effort to unravel the real context of a comma;*
— *the apparent complexity of a program is increased by introducing an extra level of bracketing: we like infix functors because we are not tolerant to nesting;*
— *there is nothing special about lists as compared to other terms (except that they are used by some built-in predicates): the extra notation contributes to the fact that some people are more comfortable with lists than with other terms and use them eg. to represent binary trees.*

In short, we definitely prefer

Append(NIL, l, l).
Append(e1 . l, l2, e1 . ll2) :– Apped(l, l2, ll2).
to
append([], L, L).
append([E1 | L], L2, [E1 | LL2]) :– append(L, L2, LL2).
Try some more involved examples, using (a.b).c *for* [[A|B]|C]!

3. Characters

In Prolog-10, strings are not equivalent to lists of characters, but to lists of ASCII codes, which are the standard character representation. To declare # *as a special character, one writes*
special(35).
This is ridiculous: one doesn't need to know about character codes to program in most assemblers. To be sure, there is a special convention — apparently added as an afterthought — that "#" (which is really [35]*) is sometimes dereferenced to* #:
special(Hash) :– Hash is "#".
This is ad-hocery at its worst!

Of course, one does need non-printing characters — from time to time — but special effects should be achieved by a built-in predicate without making things inconvenient for the everyday user. Adopting the natural convention of representing small-letter constants in quotes, we would write

Ctrl_z(char) :– Ord_chr (26, char).
Letter(char) :– char >= 'a' , char =< 'z'.
Letter(char) :– char >= A, char =< Z.
rather than
ctrl_z(26).
letter(Char) :– Char >= 97, Char =< 122.
letter(Char) :– Char >= 65, Char =< 90 .
(or whatever the codes might be).

4. Priorities

This is a minor point, but it is a good example of the effects of premature de-facto standardisation. It is generally accepted in elementary arithmetics, programming languages etc. that the priority of multiplication in a + b × c *is higher than that of addition: Prolog-10 would call it lower. The source of this strange departure from custom is probably accidental: a mistake, or an artefact of the parsing method.*

We will stop here, although it would not be difficult to raise some other complaints. A good example is the interaction of the cut and the system predicate call. Apparently, in

a(X) :– b, call(X), fail.
a(_) :– c.
c is not executed if the goal is
:– a((d, !)).
This, again, seems to be an accidental artefact of a concrete implementation which allows horrible misuse of the language and sometimes makes other implementations more difficult. The rational solution is to make all manuals discourage taking advantage of this effect by calling it non-standard!

The four points we have raised here barely scratch the surface of Prolog syntax and the set of built-in predicates. Would it be too difficult to get rid of the old conventions by modifying existing implementations and passing existing programs through a preprocessor? If revisions of this sort are ever to be made, now is the time!

Acknowledgement

The question of syntax has been raised by others from time to time; authors of several implementations adopted more rational conventions. It would be difficult to draw up a complete list of references, so allow us to acknowledge our own lack of originality in this note.

SECOND INTERNATIONAL LOGIC PROGRAMMING CONFERENCE

Uppsala University, Uppsala, Sweden
2-7 July 1984

Call for Papers

The series of International Logic Programming Conferences is the main forum for papers describing original *Logic Programming Research*. The conference now arrives in Uppsala following a successful meeting in Marseille 1982.

PROGRAM AREAS

Applications of Logic Programming

Data bases — Natural language understanding
Education — Office systems
Expert systems — Speech understanding
Graphics — Vision
Knowledge theories

Architecture and Hardware for Logic Programming

Architecture models — Memories
Architecture assessment — Networks
Buses — Processors
Circuits — (V)LSI design methods

Foundations of Logic Programs

Computability — Semantics
Program analysis and complexity

Logic Programming Implications

Economical — International
Educational — Professional
Industrial — Social

Logic Programming Languages

Algorithms and methods — Implementations
Constructs and principles

Logic Programming Methodology

Formal development of programs (synthesis)
Program transformation
Control of program computations
Verification — Metalevel inference

PAPER SUBMITTAL

The conference will consider all aspects of *Logic Programming*. Authors should submit four complete copies of their papers concerning (but not limited to) the listed topics. The papers should arrive no later than January 15, 1984 to the *Program Chairman*:

Professor Sten-Ake Tärnlund
c/o Professor J. A. Robinson
School of Computer and Information Science
313 Link Hall, Syracuse University
Syracuse, New York 13210, USA

Author notification: March 15, 1984.
Camera-ready copy: April 15, 1984.

Papers will be reviewed for their clarity, originality and significance by three members of the program committee.

Papers must be written and presented in English and be typed double spaced on one side only of each sheet. They must not be longer than 7 proceedings pages, about 5 000 words.

Approvals for presentations and publications must be obtained from the authors when they submit their papers. A paper should contain the following items: Abstract and title of paper; name, country, affiliation, mailing address and telephone number; one program area; the following signed statement: "The paper will be presented at the conference by one of the authors."

PROGRAM COMMITTEE

K. A. Bowen, Syracuse University, USA
M. Bruynooghe, Leuven University, Belgium
K. Fuchi, ICOT, Japan
H. Gallaire, Laboratoires de Marcoussis, France
K. M. Kahn, Uppsala University, Sweden
P. Köves, SZKI, Hungary
F. G. McCabe, Imperial College, UK
F. Pereira, SRI, USA
L. M. Pereira, Universidade Nova de Lisboa, Portugal
J. A. Robinson, Syracuse University, USA
E. Shapiro, Weizmann Institute, Israel
S.-A. Tärnlund, Uppsala University, Sweden
M. van Caneghem, University of Marseille, France

LOGIC PROGRAMMING PROJECTS FUNDED THROUGH DEC'S EXTERNAL RESEARCH PROGRAM

Realizing the importance and potential of logic programming, and appreciating the dificulties associated with conducting research in this area without adequate computing resources, Michael Poe and Roger Nasr, from the 32-Bit Systems Advanced Development Department in Digital Equipment Corporation, have proposed and received the approval for funding a series of Logic Programming related projects through DEC'S external research program office. The funding provides for credit applicable towards the purchase of DEC equipment, mostly VAX11-730'S and VAX11-750'S, and is in exchange for research results in the form of literature, technical assistance in specific Logic Programing topics, or software for research and prototyping purposes. The five projects are the following:

— David Warren, SRI International, USA, was funded to work on a Prolog engine design, applicable to VAX architecture, and with potential for implementation in microcode.
— Ehud Shapiro, Weizmann Institute, Israel, was funded to work on prolog programming environments aimed at increasing programmer productivity and convenience (E. G. Built-in Editor and Intelligent Debugger).
— Luís Moniz Pereira, New University of Lisbon, Portugal, was funded to work on knowledge engineering related Prolog extensions as well as expert system implementation methodology. A second project, with Luís Monteiro, is for multiprocessor Prolog implementation.
— Alain Colmerauer, Groupe Intelligence Artificielle, Luminy, France, was funded to work on Prolog extensions for natural language processing, infinite trees support, alternative unification techniques, and enhanced rule database access techniques.

LOGIC PROGRAMMING WORKSHOP'83
Albufeira, Algarve, Portugal

Maurice Bruynooghe

A very nice place at the borders of the Atlantic, a program spread over five days, this was the setting for the Workshop. As a conse-

quence, the daily program was not overloaded and a two-hour lunch-time break allowed to absorb delays on the timetable. This created a jovial informal atmosphere, with plenty of time as well for informal meetings and exploration of the nice surroundings.

The presentations gave a good impression of the ongoing research in the fast growing field of Logic Programming. The panels gave a broader perspective on the individual research efforts, sketched the most important areas and created some deeper understanding of the basic research needs in this field.

LOGIC PROGRAMMING WORKSHOP'83

Paul F. Wilk
Department of Artificial Intelligence,
University of Edinburgh, UK

Over 80 delegates from 17 different countries attended the Logic Programming Workshop'83 in the Algarve, Portugal, from the 26th June to the 1st July. The workshop was organized by the Núcleo de Inteligência Artificial, of Departamento de Informática, Universidade Nova de Lisboa. The program chairman was Luís Moniz Pereira.

The workshop consisted of 43 formal presentations (35 of which appear in the proceedings) and six panel sessions. The sessions were divided into topics on: Natural Language; Knowledge Base Systems; Logic Programming Theory; Prolog Implementation; Data Bases and Logic Programming Methodology.

Jan Chomicki (Warsaw University, Poland) discussed the problems related to using Prolog as an implementation language for data bases. In particular the paper discusses how to organize and access large Prolog data bases (based on extendible hashing and partial match retrieval).

Wlodzimierz Grudzinski (Warsaw University, Poland) described SPOQUEL — a query language for relational data bases (written in Prolog).

Tomasz Pietrzykowski (Acadia University, Wolfville, Canada) presented a data base model of a functional programming language, called PROGRAPH, which uses a graphical display for the user interface.

Luís Pereira (Universidade Nova de Lisboa, Portugal) presented a relational data base modeller for generating data bases. The program uses information gathered interactively, from the user, to generate specific menu-based consultation programs.

Jan Komorowski (Harvard University, USA) presented a universal display editor (not in the proceedings) as a software prototype language tool. An example application of a Pascal syntax-directed editor was explained.

Patrick Saint-Dizier (IRISA, Campus Universitaire de Beaulieu, France) described a way of building an intelligent interface between a human and a computer.

António Porto (Universidade Nova de Lisboa, Portugal) gave a talk about the natural language interface to a garden store assistant (written in Prolog).

Miguel Filgueiras (Universidade Nova de Lisboa, Portugal) described the design of a kernel for a knowledge directed parser of natural languages. To check consistency of syntax analysis with respect to meaning, non-application dependent semantic tests are performed during syntax analysis. The application dependent parts of the semantic analysis are specified in a separate module. Therefore, it is claimed that it is easier to adapt the interface to new applications.

Paul Sabatier (University of Paris, France) presented a formalism and implementation technique by which left and/or right contextual constraints (used in contextual grammars to specify rule ordering) can be easily expressed and efficiently computed in Prolog II. The implementation technique builds a graph containing contextual information (built during parsing) which may be used to recover the context when a contextual constraint has to be satisfied.

Veronica Dahl (Simon Fraser University, Canada) chaired the panel on Natural Language the theme of which was current trends in logic grammars. A case was made for context-sensivity in grammars contrary to the current trend of augmenting context-free grammars with new rules during the parse.

Martin Williams (Heriot-Watt University, UK) described the implementation of an approach to security and integrity in Query-by-Example based on the idea of maintaining the consistency of data in the data base. The approach extends the conventional types of integrity constraint to include functional, multivalued and embedded multivalued dependencies.

José Neves (Heriot-Watt University, UK) presented an extension to Query-by-Example (written in Prolog) that enables a user of a data base to obtain positive and negative feedback information from queries or updates that are incomplete or incorrect.

Igor Mozetic (Jozef Stefan Institute, Ljubdljana, Yugoslavia) described the development of an expert system that models the electrical behaviour of the heart. The model is used to automatically generate a knowledge base of all physiologically possible combinations of cardiac arrhythmias and their corresponding ECG descriptions.

Ferenc Darvas (Szki, Budapest, Hungary) described a logic-based expert system for model building in regression analysis. The system (written in MProlog and FORTRAN; which communicate by files) has been used to test drug design performance.

Eugénio de Oliveira (Universidade Nova de Lisboa, Portugal) described a proposal to develop expert building tools in Prolog. The system will combine a knowledge base acquisition subsystem (gathering semantic nets, metaknowledge and production rules) with a consultation subsystem (which uses metaknowledge to guide the developer through the presentation of explanations, reasoning and deductions).

Ed Stabler (University of Western Ontario, Canada) discussed the problem of achieving optimally efficient response to queries addressed to a large deductive data base. Moreover, where the user wishes to interactively optimize the query for subsequent use. The system permits the interactive addition of general rules (expressions containing logical variables) as well as particular facts (expressions containing no variables) to the data base.

Jack Minker (University of Maryland, USA) chaired the panel on Knowledge Based Systems. Current issues in developing expert systems were discussed, in particular: what distinguishes an expert system from an application pogram; how is the expert's knowledge acquired and represented; how is temporal data handled; what toolkit should be provided for the expert system developer; what morals should be applied to expert systems; how is search controlled in an expert system and what are the criteria for user acceptability (particularly when different

classes of user perceive different system models).

Pierre Deransart (INRIA, France) proposed an operational algebraic semantics for Prolog programs that follows resolution.

Andrzej Lingas (Linkoping University, Sweden) proposed that in order to fully understand the behaviour of parallel goal execution of logic programs it was necessary to apply the ideas of Turing-machine complexity theory to the complexity measures of logic programs i.e. goal size, goal length, goal depth and conjunctive goal size.

Dan Sahlin (The Royal Institute of Technology, Sweden) described an abstract machine called "gepr" (goal, environment, program and resumption register). The gepr machine is a state transition system. The paper contains a definition the transition rules that convert one state to the next. Each rule corresponds to a rule in a natural deduction system.

Patrizia Asirelli (Istituto Elaborazione Informazione, Pisa, Italy) described a fixed point semantics of Horn clauses with infinite terms — infinite terms are often used to define parallel communicating processes in Prolog but current semantic definitions are incomplete and apply to unwanted infinite terms. Two semantic definitions are proposed based on least fixed point construction. A proposed operational semantic definition generates a unit clauses — representing a terminal clause (if non has been defined). A fixed-point semantics is defined which reflects the idea that non-terminal symbols are partial approximations of infinite terms.

Patrizia Asirelli then commented on some aspects of the first order semantics of a connective suitable for expressing concurrency. This is achieved by introducing the concept of class; a cluster of concurrent atoms; where an atom has a predicate and "N" terms. Processes communicate by semaphores manifested as shared logical variables.

Maarten van Emden (Imperial College, London, UK) chaired the panel on Logic Programming Theory. The panel commented on the growth in theory papers. In particular many ideas had been transferred from other fields such as complexity theory. The topics discussed included complexity, concurrency, infinite terms, operational and fixed-point semantics and negation by failure.

John Lloyd (Melbourne University) announced the release of MU-Prolog; which is written in "C", has a correct implementation of negation by failure, data base support, and a syntax similar to DEC-10 Prolog.

Gerard Ballieu (Katholieke Universiteit Leuven, Belgium) described a virtual machine to implement Prolog. The machine is based on David Warren's abstract Prolog machine but uses a structure copying techniques for working storage.

David Bowen (Edinburgh University, UK) described a Prolog implementation, similar to that of Ballieu, that aims to combine a high degree of portability with speed and an efficient utilization of memory. The virtual machine for the implementation is written in the programming language "C". The design is well suited to optimization for particular machines, because there is is a central core which can be translated into microcode or assembly language.

Paul Wilk (Edinburgh University, UK) described the production and evaluation of a set of Prolog benchmarks (not in the proceedings). The benchmarks consist of a number of large AI programs and over 100 small benchmarks. A methodology was described for producing the small benchmarks (each one of which is designed to benchmark a particular facet of an implementation) which can be applied to other AI languages. Benchmarking results were given for four Prolog implementations on six different computer systems.

Frank McCabe (Imperial College, London, UK) briefly described Lambda Prolog which is an attempt to unite Lambda Calculus and logic programming. He then described Abstract Prolog Machine (APM) which will be used as a target architecture for Lambda Prolog. However, APM is seen mainly as a Prolog architecture for single user machines.

Hiroshi Nishikawa (Institute for New Generation Computer Technology, Japan) gave an overview of the design of the Personal Sequential Inference Machine architecture. PSI has a 40 bit word format (8 bit tag and 32 bit data) It has a 32 bit real address space (no virtual addressing) with a non-structure sharing implementation of working storage. The language system consists of a subset of DEC-10 Prolog with extended abilities for hardware resource handling, interrupt handling and process control. Notedly, the machine architecture includes provision for garbage collection and process switching.

Marco Bellia (Universita di Pisa, Italy) proposed a compiler that maps Prolog to a demand driven architecture. This is done by automatically annotating clauses (similar to automatic generation of mode declarations) according to functional dependencies. An annotation distinguishes between an atomic formula that computes a value for a given variable and one which uses the value of the variable.

Stanislaw Matwin (University of Ottawa, Canada) described an intelligent backtracking algorithm, applicable to first order logic. Essentially, a depth first search of the proof tree is directed by information kept in a separate graph structure; which represents the unifications generated during the proof.

Jack Minker outlined PRISM — A Parallel Inference System for Problem Solving. PRISM is based on logic programming and is implemented on SMOB; a parallel multi-microprocessor system. The system is designed to provide a general experimental tool for the construction of large artificial intelligence problem solvers.

Seif Haridi (The Royal Institute of Technology, Sweden) described a mechanism that would control the traversal of the search tree in an Or-Parallel token machine (where a token pool contains processes that are ready to execute but have not yet been allocated a processor). This is complemented by a mechanism that prunes the search tree; removing branches that are obsolete computations.

Subsequently Haridi explained a machine architecture (similar to that of ALICE by John Darlington of Imperial College) for the Or-Parallel token machine.

Maurice Bruynooghe (Katholieke Universiteit Leuven, Belgium) chaired the panel on Prolog implementation. The talk focussed on the effect of a Prolog implementation on programming style — noting that programming elegance was often sacrificed for time and space efficiency. The desire for a Prolog standard was mooted because of Prolog portability problems. But generally, delegates thought that the subject area was at too early a stage in its evolutionary development to merit this.

E. Elcock (University of Western Ontario, Canada) gave reasons why Prolog should not be thought of as a specification language. In particular, the procedural semantics of a

Prolog program are incomplte; often it is not clear that a program will terminate so it is necessary to consider proof of termination of a program.

Pavel Brazdil (Faculdade de Economia, Porto, Portugal) discussed the problems associated with the development of Prolog programs (not in the proceedings). Suggestions were made for a Prolog toolkit similar to that of Interlisp.

Richard O'Keefe described a polymorphic type system for Prolog (obtainable from DAI, Hope Park Square, Edinburgh University, UK) and how it integrated with other Prolog development tools written at Edinburgh University. One advantage of a good type system is that it provides a static tool for determining whether all cases in a Prolog predicate have been considered. Moreover this type system can be used as a basis for encapsulation; providing an abstract data type facility.

The Panel on Data Bases was chaired by Jack Minker. The panel discussed Hervé Gallaire's paper (Laboratoires de Marcoussis, France) "Logic Data Bases vs Deductive Data Bases". Gallaire's important paper presents a taxonomy of data bases formulated by decomposing logic programming and data bases into their component parts and studying their interconnections. Detailed descriptions of two important contrasting implementation approaches are described; logic data bases and deductive data bases. Logic data bases are built above or aside Prolog and have their own description and manipulation languages. The deductive data base approach uses logic to provide extensions to conventional data base systems; if only to remove the problem of implementing query languages with procedural languages. Gallaire expects that the Japanese Fifth Generation Project will reveal a more precise taxonomy than his, based on axioms rather than relationships.

Madhur Kohli (University of Maryland, USA) presented a theory for the intelligent control and execution of function free logic programs based on integrity constraints. The integrity constraints may be user supplied or automatically generated at run-time by analysis of goal failure.

Chris Moss (University of Pennsylvania, USA) defined a predicate "seqof" that enables the total set of solutions to a problem to be returned individually, and processed individually, rather than returned collectively and processed as required.

Harvey Abramson (University of British Columbia, Canada) gave a definition of HASL (contained in the proceedings; written in Prolog). HASL is a purely application language which uses SASL's combinator reduction machine in conjunction with unification based conditional binding expressions.

Ed Babb (ICL, Stevenage, UK) put forth an alternative philosophy for adapting resolution for logic programming termed Finite Computation Principle. The principle maintains the power of symbolic substitution but seeks to manage infinite processes by combining order independence of predicate execution with infinite process detection. Management is performed by knowing and detecting the conditions that lead to infinite processes and then applying axioms of logic to determine a predicate order that makes the computation finite.

Keith Clark (Imperial College, London, UK) chronicled the historical development of Logic Programming in relation to Computer Science (not in the proceedings). From this chronology the reasons for the features incorporated into PARLOG (A parallel logic programming language) were rationalized. The main features of the language are: Modules; And-parallelism; Or-parallelism; Eager functions; Lazy functions; Set expressions and Prolog as a subset (unlike Concurrent Prolog).

Ehud Shapiro presented a rationale for the design of Concurrent Prolog. Of particular note was his reluctance to move away from simplicity until experimentation had confirmed him of the features that should be added to the language (not in the proceedings; a copy of the interpreter, written in Prolog, can be obtained from Department of Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel).

Akikazu Takeuchi (Institute for New Generation Computer Technology, Japan) described interprocess communication in Concurrent Prolog. This is realized by sharing variables amongst processes. Therefore when a shared variable is instantiated to a message all processes sharing the variable receive the message. However, in Prolog, destructive assignment of shared variables is not permitted so every time a message is sent a new shared variable must be generated for the next communication. This form of communication is known as streaming.

Luís Monteiro (Universidade Nova de Lisboa, Portugal) described work, similar to that described by Asirelli, for concurrent programs. However, in this work Prolog is extended with the concept of an event, which gives a temporal logic programming language.

António Porto (Universidade Nova de Lisboa, Portugal) described a concurrent language design that combines action reduction with logic programming. The main features of the language are: synchronization; concurrency; action rules and abstract data types.

The Panel on Logic Programming Methodology was chaired by Ehud Shapiro. The session was oriented to research methodology rather than programming methodology. (Delegates noted that there were no good texts available that described the programming methodology and techniques applicable to Logic Programming. However Ehud Shapiro announced that shortly he would have a book published suitable for advanced logic programmer's.

It should be noted that in some cases presentation of work was not given by the author of the paper that appears in the proceedings. Furthermore, some papers that appeared in the proceedings were not presented at the conference and so are not covered here.

The following list represents the open research areas covered by the conference: (logic programming is defined here to be synonymous with Prolog):

1. data base implementation,
2. natural language processing,
3. query languages,
4. intelligent user interface,
5. expert systems,
6. expert system building,
7. logic programming semantics,
8. parallel logic programming machines,
9. sequential logic programming machines,
10. abstract logic programming machines,
11. intelligent proof-tree search,
12. logic programming toolkits,
13. sequential logic programming languages,
14. parallel logic programming languages.

PROLOG — WHAT IT IS AND WHERE TO GET IT

Fernando Pereira

SRI International

Why should I want Prolog?

Prolog is a simple but powerful programming language for symbolic computation based on a computationally treatable subset of logic. It can be seen as a clean combination of the concepts of symbolic programming languages such as Lisp and those of relational databases.

Prolog was born at the University of Marseille in the early 70's. After 8 years of comparative obscurity un a small community of dedicated implementors and users, Prolog has been brought to the attention of the wider world by its surprising adoption as the starting point for the Japanese 5th generation computer research effort.

Prolog has been used for (order of items doesn't imply any form of ranking:

- natural language interaction with computer systems
- architectural design
- drug design (very successful commercial application in Hungary)
- VLSI circuit analysis
- artificial intelligence research
- compiler writing (Prolog itself, APL)
- algebraic computation
- database access and data description languages
- descrete event simulation
- program development systems
- expert systems

and certainly more thant I can remember or know about.

I have a Unix system; how do I get Prolog?

Please note that Prolog, like other interactive symbolic languages requires more space to run than lower-level languages. Don't expect miracles on a PDP-11.

For the PDP-11 UNIX V6 or V7:

- Chris Mellish's system, obtainable from the Dept. of AI. Edinburgh University, Forest Hill, Edinburgh, Scotland.
- Very compact and reasonably fast, will run substantial programs even without separate I/D space.
- As far a I know, its development has been frozen.
- Written in PDP-11 assembly code.

For the VAX UNIX 4.1 BSD or Eunice under VMS:

- CProlog, obtainable from EdCAAD, 20 Chambers Street, Edinburgh EH1 1JZ, Scotland.
- Designed for machines with 32 bit addresses; requires at least 750K of virtual memory to run comfortably.
- it has an extensive set of system predicates.
- still being developed and improved: if you get the initial licence from EdCAAD, you may ask me for bug fixes and improvements.
- Written in C and Prolog.

Forthcoming:

For the VAX AND Z8000:

- POPLOG, combined POP-11 and Prolog from the University of Sussex (available now only for VMS).
- Reported to be somewhat faster than CProlog.
- Written in POP-11 and VAX assembly code.

For 68000:

- EdCAAD's CProlog is being ported.

All these systems comply broadly with the syntax and repertoire of system predicates described in "Programming in Prolog" by Bill Clocksin and Chris Mellish, Springer Verlag 1981. This is the book to get if you want to get into Prolog.

Others:

There are several other more or less portable Prolog systems written in C or Pascal, but they are rather experimental and I cannot recommend them for general use.

If you have information about other Prolog systems, want to know more about Prolog or have bug reports on CProlog, write to:

Fernando Pereira
Artificial Intelligence Center
SRI International
333 Ravenswood Ave.
Menlo Park, california 94025
USA
Phone: (415)859-5494
ARPANET: PEREIRA SRI-AI

The 1984 IEEE International Symposium on LOGIC PROGRAMMING

Atlantic City, New Jersey, February 6-9, 1984

Sponsored by the IEEE Computer Society and its Technical Committee on Computer Languages

The symposium will consider fundamental principles and important innovations in the design, definition, and implementation of logic programming sustems and applications. Of special interest are papers related to parallel processing. Other topics of interest include (but are not limited to): distributed control schemes, FGCs, novel implementation techniques. performance issues, expert systems, natural language processing and systems programming.

Additional information:

Doug DeGroot
Program Chairman
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598, USA

Technical Committee:

Jacques Cohen (Brandeis)
Doug DeGroot (IBM Yorktown)
Don Dwiggins (Logicon)
Bob Keller (University of Utah)
Jan Komorowski (Harvard)
Michael McCord (IBM Yorktown)
Fernando Pereira (SRI International)
Alan Robinson (Syracuse University)
Joe Urban (Univ. Southwestern Louisiana)
Adrian Walker (IBM San Jose)
David Warren (SRI International)
Jim Weiner (Univ. New Hampshire)
Walter Wilson (IBM DSD Poughkeepsie)

Proceedings will be distributed at the symposium and will be subsequently available for purchase from IEEE Computer Society.

CALL FOR PROGRAMS

I am looking for interesting Prolog programming examples, to be included in a book I am writing on advanced Prolog programming techniques. Interesting programs that satisfy 2 or more of the following requirements are solicited:

(1) The program solves an interesting programming problem, or implements a non-trivial algorithm, for which previous reference in the literature is available.

(2) The program demonstrates an interesting Prolog programming technique, not necessarily available in other programming languages.

(3) The program is written in Pure Prolog. Negation and cut included. I/O excluded unless it is an inherent part of the problem, such as data-entry and display, or tokenization. Side-effects excluded unless the problem explicity requires the manipulation of the clauses in the internal data-base.

(4) The program is short.

(5) The program is brilliant dirty Prolog trick.

Please include references to published statements of the problem and/or solutions in Prolog or other programming languages.

Authors of the programs published will be appropriately acknowledged.

Please send them to:

Ehud Shapiro
Dept. of Appl. Mathematics
The Weizmann Institute of Science
Rehovot, 76100 ISRAEL

AI AND DATABASE RESEARCH LABORATORY AT THE UNIVERSITY OF MARYLAND

by Jack Minker

The AI and Database Research Laboratory at the University of Maryland consists of Jack Minker, the director of the laboratory, and a number of faculty and students engaged in diverse activities. Other faculty connected with the group are Donald Perlis and John Grant. Perlis is concerned with theoretical issues associated with logic programming, non-monotonic, reasoning, and cognitive problems. Grant, of Towson State University, is primarily concerned with theoretical issues in databases. It is anticipated that other faculty will become part of the laboratory.

There are several graduate students and one visiting scholar in the laboratory. The graduate students of the laboratory are: U.S. Chakravarthy, A. Csoeke-Poeckh, S. Kasif, M. Kohli, r. Piazza, and D. Wang. The visiting scholar is: S. Desai (India) — United Nations Fellowship (until September'83).

Individuals who plan to visit the AI and Database Research Laboratory for short periods of time include: I. Futo of Hungary, S. Gregory of the Imperial College of London, T. Imielienski of McGill university, R. Reiter of the University of British Columbia, J. A. Robinson of Syracuse University and F. Pereira of SRI. Visitors during the past year were: K. Bowen, Syracuse, R. Reiter, U. British Columbia and S. Haridi, Sweden. Individuals interested in activities in which our laboratory is engaged are encouraged to visit the AIDBRL at Maryland.

2. Computers and Programs

In addition to conventional computer systems, we have available a parallel computer, ZMOB. This computer was designed by Chuck Rieger while he was a member of our faculty, is being implemented, and should be available by the summer 1983. ZMOB consists of 256 Z80A microprocessors interconnected on a high speed conveyor belt with the VAX 11/780 as the host processor. Moblets (individual Z80A machines) can be accessed by physical address and by pattern matching. A message can be sent to a specific machine, to all machines, or to a set of machines. Point-to-point high speed transmission between processors exists.

The AIDBRL members are developing a logic programming system for the ZMOB, termed PRISM (parallel inference system).

3. Logic Programming Activities

a. *PRISM*

The major activity of the group is the development of PRISM. The work started in earnest in aproximately January 1982. PRISM has been implemented and is being tested on a simulated system since the ZMOB hardware is not available. Is is expected that ZMOB will be available during the summer of 1983.

The underlying ideas behind PRISM appearedin Eisinger, Kasif, and Minker [1981, 1982]. A detailed functional report on the system appears in Minker et al. [1982]. PRISM is a first-generation system to permit parallelism in problem solving. As such, many features that are required for an ultimate parallel problem solving system have been omitted in the initial system. (For example, we have ignored operating systems and recovery issues in the inital development). There are, however, a large number of capabilities built into the initial system.

PRISM consists of several parts: a user host interface that exists on the VAX 11/780; a set of Z80A machines (moblets) designated as problem solvers (PSMs); a set of machines designated as Extensional Database Machines (EDB) that store ground atomic formulae (relational database tables); a set of machines designated as Intensional Database Machines (IDB) that store procedures (the general rules in the system); and a IDB monitor machine which determines if the IDB machines are overloated.

The user can specify control in terms of the sequence of atoms to be executed in a set of problems to be solved. Atoms can be executed in parallel,sequentially, or as specified by a partial ordering. Similarly procedures can be specified as being executed sequentially, in parallel, or as specified by a partial order. The PSM has been written in a modular fashion to permit alternative control structure programs to be incorporated in the system. Alternative node and literal selection algorithm may be incorporated as part of the control structure. The user may specify the configuration (i.e., the number of moblets required as a minimum) in which a problem is to be run. If additional moblets are available, the PRISM will automatically take advantage of them. In addition to the papers cited above, the following papers descrive various portions of PRISM: Minker et al. [1983], Chakravarthy et al. [1982], Kasif et al. [1983a, 1983b].

b. *Control Structure*

The control of searches in a logic programs is important for developing efficient expert systems and problem solvers. In Kohli and Minker [1983a, 1983b, 1983c] we discuss how failures to find solutions along paths, together with integrity constraints may be used o provide more intelligent search than is achievable with backtracking. The control structure of PRISM is presented in Kasif et al. [1983].

A major objective of our work in this area is the investigation of control issues with respect to distributed problem solving. Preliminary studies have been directed towards developing a control specification language for logic programming systems.

c. *Interfacing Predicate Logic Programs and Large Databases*

Artificial Intelligence programs will have to interface effectively with large database. The AIDBRL has addressed problems involved in interfacing predicate logice programs with relational databases. In Chakravathy et al. [1981, 1982] alternative methods of interfacing a predicate logic program with relational database in a system such as PROLOG are described. Approaches discussed are modifications to PROLOG interpreters and meta-level constructs.

In Grant and Minker [1981, 1982, 1983] the optimization of compiled deductive searches that do not include recursion is considered. A branch-and-bound type algorithm is used that takes advantage of a database that is indexed, the number of responses anticipated, and performs the union of a set of conjuncts in an optimal fashion.

4. **Deductive (Logic) Databases**

It is our belief that logic provides the appropriate context in which to investigate databases. The following summarizes some of the activities in the AI and Database Research Laboratory.

a. *Survey of Logic and Databases*

In collaboration with Herve Gallaire of Laboratories de Marcoussis, France, and Jean Marie Nicolas of ONERA-CERT, Toulouse, France, this writer has written a comprehensive survey describing the various ways in which logic has been used in databases (Gallaire et al. [1983]). The paper On deductive Relational databases, (Minker [1983]) also provides a survey of deductive databases.

b. *Generalized Closed World Assumption*

The Closed World Assumption, as named by Reiter, and discussed by Nicolas and by Clarke for Horn clauses, has been generalized in "On Indefinite databases and the Closed World Assumption for Non-Horn Clauses" (minker [1982]). A proof theoretic and a model theoretic definition is provided of the generalized closed world assumption, and it is shown that the definitions are equivalent. The generalized closed world assumption leads to a non-monotonic logic. A paper in preparation by Grant and Miker will address how one can compute answers to queries using the generalized closed world assumption.

c. *Recursive Axioms*

When recursive axioms are part of a deductive database, problems may arise with respect to termination. A number of researchers have investigated how to handle recursion including Chang, Reiter, Shapiro, Henschen, and Naqvi. In "On recursive Axioms is Deductive Databases" (Minker and Nicolas [1981]) and "On recursive Axioms in Relational Databases" (Minker and Nicolas [1983]) it is shown how one can obtain termination conditions for classes of clauses that contain recursive axioms.

d. *Theories of Databases*

Depending upon the types of clauses in a database one obtains different theories which handle different phenomena. In "On Theories of Definite and Indefinite Databases", Minker [1983] precise theories are given that describe different phenomena in a database. An open world definite database is a first-order theory where negative data must be specified explicitly. A closed world definite database is characterized by another set of assumptions, and, in particular, by Horn clauses and the use of negation by lailure. Indefinite databases lead to another theory which, itself, has to modified to handle the "null value" problem in databases. Using logic captures the theories precisely.

The notion of numerical dependencies is introduced in Grant and Minker [1983]. It formalizes the constraint that with an element of an attribute, or set of attributes, at most k elements of another attribute can be associated. The uses of numerical dependencies in database design are considered via generalized normal forms.

5. **Research Directions**

The major research directions in the laboratory over the coming year will be devoted to the following areas:

(1) Implementation of PRISM on ZMOB,
(2) Experimentation using PRISM,
(3) Control structure investigations,
(4) Expert systems and PRISM,
(5) Theories of databases,
(6) Non-monotonic logic, and
(7) Parallel problem solving and architecture issues.

6. **References**

1. Chakravarthy, U., Minker, J. and Tran, D. [1981] — "Interfacing Predicate Logic Languages and Relational Databases", Technical Report 1019, Computer Science Center, University of Maryland, February 1981.
2. Chakravarthy, U., Minker, J. and Tran, D. [1982] — "Interfacing Predicate Logic Languages and Relational Databases", *Proceedings of Logic Programming Conferences,* Marseilles, France, September 1982.
3. Chakravarthy, U., Kasif, S., Minker, J., Kohli, M. and Cao, D. [1982] — "Parallel Logic Processing and ZMOB", *1982 International Conference on Parallel Processing,* August 1982.
4. Eisinger, N., Minker, J. and Kasif, S. [1981] — "Logic Programming: A Parallel Approach", Technical Report 1124, Computer Science Center, University of Maryland, December 1981.
5. Eisinger, N., Minker, J. and Kasif, S. [1982] — "Logic Programming: A Parallel Approach", *Proceedings of the First International Logic Programming Conference,* September 14-17, 1982, Faculte des Sciences de Luminy Marseille, France, 71-77.
6. Gallaire, H., Minker, J. and Nicolas, J. M. [1983] — "An Overview and Survey of Logic and Databases", Computer Science Department, University of Maryland, January 1983.
7. Grant, J. and Minker, J. [1981] — "Optimization in Deductive and Conventional Relational Data Base Systems", In: *Advances in Data Base Theory,* Volume 1, Plenum Publishing Co., New York, 1981, 195-234.
8. Grant, J. and Minker, J. [1983] — "On Optimizing the Evaluation of a Set of Expressions", to appear *International Journal of Computer and Information Systems* (1983).
9. Grant. J. and Minker, J. [1982] — "A Set Optimizing Algorithm", *Proceedings of the CISS,* Princeton, New Jersey, May 1982.
10. Grant, J. and Minker, J. [1982] — "Numerical Dependenices", *Proceedings Workshop in Logical Bases for Databases,* ONERA-CERT, Toulouse, France, 1982, pp. 13-1 to 13-48.
11. Kasif, S., Kohli, M. and Minker, J. [1983] — "Control Structure of PRISM: A Parallel Inference System for Problem Solving", to appear in *Proceedings of the Logic Programming Workshop,* Albufeira, Portugal, June 1983.
12. Kasif, S., Kohli, M. and Minker, J. [1983] — "A Parallel Inference System for Problem Solving", to appear in *Proceedings of IJCAI,* Karlsruhe, West Germany, August 1983.
13. Kasif, S., Minker, J. and Kohli, M. [1983] — "PRISM — A Parallel Inference System Based on Logic", Technical Report, Computer Science Department, University of Maryland, February 1983.
14. Kohli, M. and Minker, J. [1983] — "Intelligent Control using Integrity Constraints", to appear in *Proceedings of the National Conference on Artificial Intelligence,* Washington, D. C., August 1983.
15. Kohli, M. and Minker, J. [1983] — "A Theory of Intelligent Forward and Backward Tracking", Technical Report, Computer Science Department, University of Maryland, March 1983.
16. Kohli, M. and Minker, J. [1983] — "Control of Logic Programs Using Integry Constraints", to appear in *Proceedings of the Logic Programming Worshop,* Albufeira, Portugal, June 1983.
17. Minker, J., Cao, D., Chakravarthy, U., Csoeke-Poeckh, A., Kasif, S. and Kohli, M., Piazza, R. and Wang, d. [1983] —

"Parallel Problem Solving on ZMOB", *Proceedings, Trends and Applications 83.*

18. Minker, J. [1982] — "On Deductive Databases", to appear in *Journal New York Academy of Sciences,* Summer 1983, also Technical Report 1184, Computer Science Center, University of Maryland, 1982.
19. Minker, J. [1982] — "On Indefinite Databases and the Closed World Assumption", *Proceedings of the CADE-6 Conference,* New York, 1982, also Technical Report 1076, Computer Science Center, University of Maryland, July 1981.
20. Minker, J. [1983] — "On Theories of Definite and Indefinite Databases", Technical Report, Computer Science Department, University of Maryland, February, 1983.
21. Minker, J. and Nicolas, J. M. [1981] — "On Recursive Axioms in Relational Databases", Technical Report 1119, Computer Science Center, University of Maryland, July 1981.
22. Minker, J. and Nicolas, J. M. [1983] — "On Recursive Axioms in Deductive Databases", in *Information Systems,* Vol. 7, No. 4 (1983).
23. Minker, J., Asper, C., Cao, D., Chakravarthy, U. S., Choeke-Poeckh, A., Kasif, S., Kohli, M. and Piazza, R. [1982] — "Functional Specification of the ZMOB Parallel Problem Solving System", technical Note Z-1, August 1982.

MU-PROLOG RELEASE

The MU-PROLOG system is now available, for educational and research purposes, at a cost of $100. It is written in C and is especially suitable for UNIX environments. It is currently running, under UNIX, on a VAX 11/780, a Perkin Elmer 3240 and an MC68000-based machine.

Features

(1) all UNIX PROLOG and most DEC-10 PROLOG facilities are provided.

(2) There are extra control facilities, which enable coroutining. User-defined procedures may have *wait declarations,* which can delay calls to these procedures. Calls to many system predicates may also be delayed.

(3) Three sound forms of negation are provided: not (~), not equals (~=) and if-then-else. Calls to these predicates are delayed if they are insufficiently instantiated.

(4) With Berkeley UNIX (at least), MU-Prolog *save files* are executable. They can also access the command line arguments, with the *argv* predicate.

For more information, contact:

The Secretary
Department of Computer Science
University of Melbourne
Parkville 3052
Australia

JOBS AT EDINBURGH IN PROLOG IMPLEMENTATION / DEVELOPMENT

At Edinburgh we hope shortly to have two research posts available for work in Prolog Development, one sponsored by International Computers Ltd., and one by the Science and Engineering Research Council.

If anyone has [studens about to finish courses or research degrees and who have] interests in this area.

Please contact:

Dr. David Bowen
Department of Artificial Intelligence
Forrest Hill
Edinburgh, UK

INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING

WG5.2 Working Conference on Knowledge Engineering in Computer-Aided Design

11-14 September 1984
Budapest, Hungary

CALL FOR PAPERS

The conference

IFIP's Working Group 5.2 on Computer-Aided Design is organizing a working conference on *Knowledge Engineering in Computer-Aided Design* to be held neas Budapest, Hungary, as part of its conference series. These working conference explore some CAD topics in detail. The language of the conference will be English.

Aims

The Working Conference aims to provide a forum for the exchange of ideas and experiences related to knowledge engineering in computer-aided design, to present and explore the state-of-the-art of knowledge engineering in computer-aided design, to promote further development and to delineate future directions.

Papers

The conference will have two primary themes:

(a) state-of-the-art research in knowledge engineering in CAD, and

(b) state-of-the-art practice of knowledge engineering in CAD.

The papers with the discussion will be published by the North-Holland Publishing Company under the title of the conference.

Call for papers

Intending authors are invited to submit papers within the themes of the conference — particularly within the following topic areas:

(i) Expert Systems in CAD
— as part of analysis
— as part of design
(ii) Encoding Design Knowledge for CAD
(iii) Knowledge Engineering Methodologies Used in CAD
— knowledge representation
— knowledge acquisition
— control methods to guide use
(iv) Knowledge Engineering and CAD Graphics
(v) Knowledge-Based Simulation
(vi) Knowledge-Based representation of Artifacts
(vii) Applicable Software
— knowledge engineering
— konwledge-based CAD systems
(viii) Implications of Knowledge Engineering for CAD Design Process
(ix) User Experience of Knowledge-Based CAD Systems

Timetable

Intending authors should submit their proposals as soon as practicable.

1. Full paper (four copies) submitted to the address below no later than *24 February, 1984*
2. Notification of authors of selected papers by *30 April, 1984*
3. Conference brochure available *1 May, 1984*
4. Final copy of selected papers in reproducible form from authors by *1 July, 1984*
5. Close of conference registration *1 August, 1984*
6. Preprints sent to registrants *August, 1984*
7. Conference *11-14 September, 1984*

Conference format

1. The conference is scheduled for four days with a restricted number of participants.

community news & events

2. About twenty to twenty-five papers will be selected for presentation. It is assumed that the selected authors will attend the conference.
3. The papers will form the conference pre-prints which will be mailed to all participants.
4. Papers will be presented with considerable time available for discussion which will be recorded to form the conference proceedings.
5. The official language of the conference is English.

Address for all correspondence

All papers, queries and correspondence should be addressed to:

Professor John S. Gero
Department of Architectural Science
University of Sydney
Sydney N.S.W. 2006
AUSTRALIA
Telex: AA20056 GERO-ARCHSCI
Phone: (61)-(2)-908 2942 or
(61)-(2)-692 2328

CHANGE OF ADDRESS

William F. Clocksin has moved to:

Computer Laboratory
University of Cambridge
Corn Exchange Street
Cambridge CB2 3QG
England

THE NEW SOUTH WALES
INSTITUTE OF TECHNOLOGY

Australia

SENIOR LECTURER

Faculty of Mathematical
& Computing Sciences
School of Computing Sciences

The School of Computing Sciences offers a Bachelor's Degree, a Postgraduate Diploma, a Master's Degree by research and thesis and a Master's Degree by coursework and report. The School has extensive computing facilities including a 3MB Prime 750 and a WICAT 200 as well as numerous microcomputers. It is also the major academic user of the Institute's Honeywell network based on duplexed Level 66/60 systems supporting 200 terminals. The school is active in research and consulting to industry.

The School is divided into two Departments, Computer Science and Information Sciences, with a total academic establishment of 40. Students of the highest calibre are attracted to the School's courses.

The successful candidate will be appointed to a Senior Lectureship in the Department of Computer Science. Applicants must have a higher degree and should have teaching competence and strong research records in one of more of the following areas: artificial intelligence, logic programming, and theoretical computer science. Other research interests of the Department include microprocessor arrays, systems architecture, languages and processors, performance evaluation, computer graphics, and programming techniques.

Further information concerning the above position may be obtained from Dr. J. R. Quinlan, Head of Computing Sciences, on (02) 218 9428, quoting reference No. 83/108.

Salary for this position will be in the range of $30,096 to $35,077.

Written applications should include: address, telephone number, personal particulars, evidence of qualifications, work experience, research work undertaken, publications, and the names and addresses of three references from whom confidential resports may be obtained.

Closing date for applications is October 14, 1983, and should be addressed to:

Appointments Officer
NSW Institute of Technology
PO Box 123 Broadway, NSW 2007
Australia

MARSEILLE CONFERENCE (September 82)

By courtesy of Martin Nilsson, from Uppsala, here is a photo of the friday evening dinner participants, immediately prior to the staging of Shakespeare's Julius Ceaser — A Meta-Play.

new journals & books

ANNOUNCING
THE JOURNAL OF LOGIC PROGRAMMING

AN INTERNATIONAL JOURNAL DEVOTED
TO THE SUBJET OF LOGIC PROGRAMMING

Published by the Logic Programming Research Center, Syracuse University

Logic programming *is one of the most active and rapidly growing areas of research in computer science today.* The Journal of Logic Programming *is intended to serve the international logic programming research community by publishing contributions on all aspects of logic programming. The Journal will publish original research papers together with survey articles, reviews and tutorial expositions aimed at a more general readership.*

The programming language PROLOG has demonstrated the elegance and power of logic programming and has stimulated a flood of investigations into the theory, application, implementation and scope of this novel deductive computational method. The recent adoption of logic programming as one of the central design ideas of Japan's Fifth Generation Computer Systems Project has called the world's attention to its potential technological importance.

The Fifth Generation Computer Systems Project has recently started its own journal, New Generation Computing. *It will cover all research areas related to the Project. The two journals together form a natural pair, one dealing with logic programming in general, the other particularly concerned with the role of logic programming within the Fifth Generation Computer Systems Project. The editorial organizations of the two journals have twelve members in common, thus ensuring a close, cooperative relationship.*

Editorial Organization

Editor-in-Chief

J. A. Robinson, Syracuse University, USA (a)

EDITORIAL ADVISORY BOARD

K. A. Bowen, Syracuse University, USA
K. L. Clark, Imperial College, UK
A. Colmerauer, University of Marseille, France (b)
K. Fuchi, ICOT, Japan (c)
H. Gallaire, Marcoussis, France
A. Hansson, Uppsala University, Sweden
K. M. Kahn, Uppsala University, Sweden
J. Komorowski, Harvard University, USA
R. A. Kowalski, Imperial College, UK (b)
J. W. Lloyd, Melbourne University, Australia
J. McCarthy, Stanford University, USA
C. S. Mellish, Sussex University, UK
U. Montanari, University of Pisa, Italy (b)
F. Pereira, SRI International, USA
P. Roussel, University of Marseille, France
E. E. Sibert, Syracuse University, USA
S. A. Tarnlund, Uppsala University, Sweden (b)
M. H. van Emden, University of Waterloo, Canada (b)
M. Bruynooghe, Leuven University, Belgium
W. F. Clocksin, Cambridge University, UK
V. Dahl, Simon Fraser University, Canada
K. Furukawa, ICOT, Japan (e)
C. C. Green, Kestrel Institute, USA
P. J. Hayes, University of Rochester, USA
H. Kanoui, University of Marseille, France
P. Koves, SZKI, Hungary
G. Levi, University of Pisa, Italy
F. G. McCabe, Imperial College, UK
M. McCord, IBM Yorktown Heights, USA
J. Minker, University of Maryland, USA
T. Moto-oka, Tokyo University, Japan (d)
L. M. Pereira, University de Lisboa, Portugal (b)
E. Y. Shapiro, Weizmann Institute, Israel (b)
P. Szeredi, SZKI, Hungary
M. van Caneghem, University of Marseille, France
D. H. D. Warren, SRI International, USA (b)

(a) Member, Overseas Advisory Board, *New Generation Computing*
(b) Member, Overseas Editorial Board, *New Generation Computing*
(c) Associate Editor, *New Generation Computing*
(d) Editor-in-Chief, *New Generation Computing*
(e) Member, Editorial Board, *New Generation Computing*

- The Journal of Logic Programming will be published quarterly.
- Approximately 100 pages per issue.
- First issue scheduled for January 1984.
- Price: $60 per year (individuals); $100 per year (institutions).
- Subscriptions are entered by prepayment only.
- All payments in US dollars. Checks or International Money Orders should be made payable to Syracuse University.

SUBSCRIPTION ORDER FORM

To: The Journal of Logic Programming
Syracuse University
Syracuse, New York 13210, USA

Please enter my subscription to The Journal of Logic Programming

Name: ______________________

Address: ______________________

____

____

____

Signature: ______________________

Date: ______________________

Logic Programming Worshop'83 Proceedings

The 630 page proceedings can be obtained Air Mail by sending a cheque (personal will do), money-order, etc., for 2,000 escudos or equivalent, in name of

LUÍS MONIZ PEREIRA
Dept. de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica
Portugal

Algorithmic Program Debugging

Ehud Y. Shapiro

This book formulates and explores a potentially productive new subarea of computer science that combines elements of programming languages and environments, logic, and inductive inference. It devises a theoretical framework for program debugging and develops techniques that will partly mechanize this activity. In particular, it formalizes and validates algorithmic solutions to finding and then fixing program bugs.

The author first develops interactive diagnostic algorithms that identify a bug in a program that behaves incorrectly, as determined by the execution of the program through a list of sample inputs that should produce known outputs. He then integrates these diagnostic algorithms with correction algorithms to form an interactive debugging system. Moreover, this system can also be used as an inductive inference algorithm for the synthesis of programs from examples of their input/output behavior, by specifying an empty an emply initial program. The book develops a number of incremental strategies for such inductive program synthesis.

The algorithms are written in the logic programming language Prolog. Because Prolog is a functional language with a simple and powerful semantics, Dr. Shapiro was able to discover elegant implementations of his debugging system, including algorithms that query the user about the intended meaning of the program, or that pinpoint the cause of an incorrect output, a missing output, or a stack overflow. This implementation results in a functioning system that is both useable and almost exactly parallel to the author's theoretical description.

Ehud Y. Shapiro is in the Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel. He took his doctorate at Yale University.

June 1983 — 208 pp. — 8 illus. — $30.00
ISBN 19218-7 SHAAH
MIT PRESS
The ACM Distinguished Dissertation Series

NEW GENERATION COMPUTING

An International Journal on Fifth Generation Computers

Editor-in-Chief **T. Moto-oka,** The University of Tokyo
Associate editor **K. Fuchi,** ICOT

Aims and Scope

Computer science is now at the threshold of a new stage of development. Conventional computer architectures established by Turing and von Neuman have begun to encounter numerous difficulties. Of these, low software productivity, insufficient processing power on non-numeric data, and poor man-machine interface are the most evident and thus require the most rapid solutions. However, promising results from fundamental studies in fields such as artificial intelligence, software engineering, computer architecture and data base, imply the possibility of creating a new generation computer which will overcome these difficulties using a new set of principles.

More specifically, this advance is being technically supported by the maturity of Lisp and Prolog and the development of new software technology such as data abstraction and object-oriented programming. Recent developments in highly parallel computer architecture, like data flow and reduction and advanced VLSI technology, are also making significant contributions towards this end.

Various research projects being conducted into new application fields such as natural language processing and knowledge engineering for Lisp or Prolog are also being aggressively pursued. These may be viewed as the final goals of the new generation computers.

Cognizant of the situation confronting computer science today, this journal, drawing support from various international groups involved in research on the fifth generation computer, as well as from the world's leading experts on new generation computer science, aims to encourage the activities and contribute to the exchange of the latest results among the people involved in creating new generation computers.

Major Fields

Basic Theory

Computation Model, Programming Language Semantics, Parallel Computation Model, Problem Solving, Theorem Proving.

Programming Language

Logic Programming, Functional Programming, Object-oriented Programming, Intelligent Programming Environment.

Computer Architecture

Parallel Machine (Data Flow Machine, Reduction Machine), High-level-language Machine, Inference Machine, Knowledge Base Machine.

VLSI Design Technology

VLSI Architcture, VLSI CAD.

Knowledge Information Processing

Knowledge Representation, Knowledge Base, Expert System, Natural Language Processing, Cognitive Science.

Authors Information

Prospective authors are encouraged to submit manuscripts within the scope of the Journal. To qualify for publication, papers must be written in English, previously unpublished and not be under consideration for publication elsewhere. All material should be sent in three copies to: Dr. T. Moto-oka, Editor-in-Chief, New Generation Computing, c/o Ohmsha, Ltd., 1-3 Kanda Nishiki-cho, Chiyoda-ku, Tokyo 101, Japan. All manuscripts will be assessed by anonymous referees.

When preparing the manuscript please follow the "Instructions to Authors" available on request from the publisher.

Fifty (50) offprints of each paper will be supplied free of charge. Up to one hundred (100) additional copies may be ordered at set prince at the time when proofs are returned.

Editorial Board

JAPAN

M. Amamiya, ECL
K. Furukawa, ICOT
Y. Futamura, Hitachi, Ltd.
K. Hirose, Waseda University
T. Itoh, Tohoku University
F. Mizoguchi, Science University of Tokyo
K. Murakami, ICOT
M. Nagao, Kyoto University
R. Nakajima, Kyoto University
S. Ohsuga, The University of Tokyo
N. Suzuki, The University of Tokyo
H. Tanaka, The University of Tokyo
H. Tanaka, ETL
Y. Tanaka, Hokkaido University
J. Tsujii, Kyoto University
T. Yokoi ICOT A. Yonezawa, Tokyo Institute of Technology

OVERSEAS

F. Bancilhon, Univ. of Paris-Sud/INRIA (France)
K. Berkling, Gesellschaft für Mathematik u. Datenverarbeitung. ISF (FRG)
W. Bibel, Technischen Universität München (FRG)
D. G. Bobrow, XEROX (USA)
A. Colmerauer, Group d'Intelligence Artificielle (France)
J. Darlington, Imperial College of Sci. & Tech. (UK)
R. Davis, MIT (USA)
P. Deutsch, XEROX (USA)
D. P. Friedman, Indiana University (USA)
G. Huet, INRIA (France)
R. Kowalski, Imperial College of Sci. & Tech. (UK)
U. Montanari, University of Pisa (Italy)
L. M. Pereira, Universidade Nova de Lisboa (Portugal)
E. Y. Shapiro, Weizmann Institute of Science (Israel)
P. D. Treleaven, University of Newcastle upon Tyne (UK)
D. A. Turner, University of Kent (UK)
Sten-Åke Tärnlund, Uppsala University (Sweden)
M. van Emden, University of Waterloo (Canada)
D. H. D. Warren, SRI (USA)
T. Winograd, Stanford University (USA)

Advisory Board

JAPAN

H. Aiso, Keio University
k. Amo, Toshiba Corp.
Y. Anraku, Oki Electric Ind. Co., Ltd.
H. Enomoto, Tokyo Institute of Technology
T. Fukumura, Nagoya University
E. Goto, The University of Tokyo
Y. Hatano, Hitachi, Ltd.
H. Inose, The University of Tokyo
Y. Mizuno, NEC Corp.
Y. Ohno, Kyoto University
H. Ohta, Mitsubishi Electric Corp.
T. Sakai, Kyoto University
K. Tanaka, Fujitsu, Ltd.
I. Toda, NTT
H. Yamada, Fujitsu Lab., Ltd.

OVERSEAS

J. Allen, MIT (USA)
J. Backus, IBM (USA)
F. Bauer, Technischen Universität München (FRG)
G. Bell, Digital Equipment Corp. (USA)
C. A. R. Hoare, Oxford University (UK)
J. L. Lions, INRIA (France)
J. A. Robinson, Syracuse University (USA)
D. Scott, Carnegie-Mellon University (USA)

abstracts

Definite Clause Translation Grammars

Harvey Abramson

Department of Computer Science
University of Bristh Columbia
Vancouver, B. C. Canada

In this paper we introduce Definite Clause Translation Grammars, a new class of logic grammars which generalizes Definite Clause Grammars and which may be thought of as a logical implementation of Attribute Grammars. Definite Clause Translation Grammars permit the specification of the syntax and semantics of a language: the syntax is specified as in Definite Clause Grammars; but the semantics is specified by one or more semantic rules in the form of Horn clauses attached to each node of the parse tree (automatically created during syntactic analysis), and which control traversal(s) of the parse tree and computation of attributes of each node. The semantic rules attached to a node constitue therefore, a local data base for that node. The separation of syntactic and semantic rules is intended to promote modularity, simplicity and clarity of definition, and ease of modification as compared to Definite Clause Grammars, Metamorphosis Grammars, and Restriction Grammars.

Algebraic Semantics of Logic Programming: the Reduction Method

G. Marque-Pucheu

École Normale Supérieure
45, rue d'Ulm 75005 Paris, France

This paper describes a new theoretical framework to deal with the semantics of pure logical programming: the algebraic semantics. This semantics is derived from fixpoint semantics by the definition of the least fixpoint as the solution of a system of equations in the set of tuples of ground terms. The careful analysis of the algebraic structure of the substitutions (considered as operators on the set of tuples of ground terms) enables the algorithmic definition (with the help of a Knuth-Bendix like completion algorithm) of a large class of syntactically simple programs. This class contains some classes of theoretical interest (clausal form of decidable subclasses of the first order predicate calculus) and some classical list manipulation programs. For these classes, the following three properties hold:

- The least model has a simple algebraic structure.
- The halting problem is decidable.
- Each logic program is equivalent to a logic program running in goal-size linear non-deterministic time.

Unfortunately, this optimisation in non-deterministic time cannot be used in practical implementation, the number of clauses in the improved program being exponentialy large. This fact can lead to an "improved" program with a worst deterministic running time.

Real-Time Functional Queue Operatings Using the Logical Variable

W. F. Clocksin

Programming Research Croup
University of Oxford
Keble Road, Oxford OX1 3QD, UK

Hood and Melville describe an efficient functional (no side-effects) implementation of FIFO queue operations, in which it is not necessary to copy the entire queue for each insertion operation. The implementation allows constant time access to both ends of the queue, but the withdrawal of an element from the queue requires the reversal of a list of lebgth k when k insertions have been performed since the previous withdrawal. Burton has independently described a similar functional implementation which uses essentially the same trick of deferring reversals. Hood and Melville also suggest but do not demonstrate the existence of a more complicated real-time version, which would require the list reversal to be distributed over a number of operations.

The logical variable, a device available to practitioners of logic programming, permis a very simple functional implementation of the queue operations: no reversals or copyings are required, and at most two conses are performed per operation. By criteria given by Hood and Melville this would constitute a real-time implementation.

The Personal Sequental Inference Machine (Sim-P or PSI) Outline of its Architecture and Hardware System

S. Uchida, M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, T. Chikayama, T. Hattori

ICOT, Japan

In the framework of the fifth generation computer project, the development of software and hardware tools are planned. One of the most important tools is the sequential inference machine (SIM). SIM is considered as a personal computer and several SIMs are planned to be developed in the initial stage of the project.

Since SIM is planned to be used for software research such as a natural language understanding system and an expert system, it is desirable that SIM can run very fast and process large programs. For this kind of purposes, super personal machine should be appropriate. On the other hand, interactive use of a computer is often important in the development of various experimental programs. For this purpose, a medium performance machine, which is relatively cheap and can be copied to share it by a few users, is more appropriate.

In the course of the development, two types of SIM are planned. One of them is the personal sequential inference machine which is called "PSI" or "SIM-P". another one is the super personal model of SIM which is called "Super PSI" or "SIM-C".

PSI is designed to be a medium performance personal machine which supports the logic programming language KL0 which is also called the version 0 of FGKL (Fifth Generation Kernel Language). And PSI is designed to attain about 20-30 K-LIPS (Logical Inference Per Second). On the other hand, Super PSI will be designed to be a high performance machine which is planned to attain 100K-1M LIPS.

In this document, functional specification of the machine architecture (PSI or SIM-P architecture) is described. As this document is a tentative report of the machine design, the content will be changed in the course of detail design and implementation processes.

abstracts

Toward a New Generation Computer Architecture

S. Uchida
ICOT, Japan

This paper outlines the research and development plan for the Fifth Generation Computer Project from the view point of computer architecture. The architectural goal of this project is to develop the basic technology to build a highly parallel processor which supports a logic programming language. As intermediate goals, a parallel inference machine, a knowledge base machine and a sequential inference machine are considered.

In the paper, an approach to the goal is introduced as well as motivations of the planning and its technological background.

Inference Machine

S. Uchida
ICOT, Japan

In this paper, the research and development plan for computer architecture in the fifth generation computer system project (FGCS Project) is described, focusing on the research on the inference machine.

Inference machines planned in this project can be classified into two types. One is a sequential inference machine and another is a parallel inference machine.

The sequential inference machine is to be developed to provide researchers with an efficient programming environment. It is designed as a personal computer which supports a logic programming language named KL0. This is an immediate development target and its design philosophy and machine characteristics are described.

The parallel inference machine which is one of the ultimate goals of this project is in a basic research stage and thus, an abstract discussion is made for introducing current research status and future research direction.

APES: A User Manual

Peter Hammond
Imperial College, London, UK

APES (A Prolog Expert System Shell) is a suite of modules which can be used to construct domain dependent expert systems.

This report describes the facilities provided in a micro-PROLOG implementation of APES and its compatibility with the Simple front-end to micro-PROLOG. The subcomponent in APES which handles uncertain information is not yet available but will be described in a future version of this report.

Learning Algebraic Methods from Examples — A Progress Report

Bernard Silver
Department of Artificial Intelligence
University of Edinburgh, UK

This paper describes LP, a program that learns new methods of solving equations from worked examples. The program is intended to be part of a self-improving algebra system.

The paper also indicates possible future directions for development of LP, and discusses its relationship to other systems.

LP is implemented in PROLOG.

The Role of Truth Maintenance in Expert Systems

Stephen J. Todd
Marconi Research Centre
Great Baddow
Essex CM2 8HN, England

An important feature of "real world" problems is that they often involve the use of knowledge which is either incomplete or likely to change. In many applications, a key performance criterion for an Expert System is its ability to produce "robust" plans or advice that can be quickly updated. These factors place great emphasis on designing flexible systems that can deal effectively with incremental changes in information. This paper compares work by Doyle and McAllester on Truth Maintenance Systems and considers their use as a basis for building flexible problem-solving systems.

Programming Meta-Logical Operations in Prolog

R. A. O'Keefe
Department of Artificial Intelligence
University of Edinburgh, UK

This paper presents some common meta-logical operations and shows how they may be coded in Prolog. It may be regarded as an appendix to "Programming in Prolog".

Protocol Verification via Executable Logic Specifications

Deepinder P. Sidhu
Research and Development Division
SDC 6 A Burroughs Company
Paoli, PA 19301, USA

This paper discusses the use of logic programming techniques in the specification and verification of communication protocols. The protocol specifications discussed are formal and directly executable. The advantages of executable specifications are: (1) the specification is itself a prototype of the specified system, (2) incremental development of specifications is possible, (3) behavior exhibited by the specification when executed can be used to check conformity of specification with requirements. We discuss Horn clause logic, which has a procedural interpretation, and the predicate logic programming language, PROLOG, to specify and verify the functional correctness of protocols. The PROLOG system possesses a powerful pattern-matching feature which is based on unification.

PARLOG: A Parallel Logic Programming Language

Keith L. Ckack & Steve Gregory
Imperial College
London, UK

PARLOG is a logic programming language in the sense that nearly every definition and query can be read as a sentence of predicate

logic. It differs from PROLOG in incorporating parallel modes of evaluation. For reasons of efficient implementation, it distinguishes and separates and-parallel and or-parallel evaluation.

PARLOG relations are divided into two types: and-relations and or-relations. A sequence of and-relation calls can be evaluated in parallel with shared variables acting as communication channels. Only one solution to each call is computed.

A sequence of or-relation calls is evaluated sequentially but all the solutions are found by a parallel exploration of the different evaluation paths. A set constructor provides the main interface between and-relations and or-relations. This wraps up all the solutions to a sequence of or-relation calls a list. The solution list can be concurrently consumed by an and-relation call.

The and-parallel definitions of relations that will only be used in a single functional mode can be given using conditional equations. This gives PARLOG the syntactic convenience of functional expressions when non-determinism is not required. Functions can be invoked eagerly or lazily; the eager evaluation of nested function calls corresponds to and-parallel evaluation of conjoined relation calls.

This paper is a tutorial introduction and semi-formal definition of PARLOG. It assumes familiarity with the general concepts of logic programming.

Algebraic Semantics of Logic Programming: The Reduction Method

G. Marque-Pucheu

École Normale Supérieure
45, rue d'Ulm 75005, Paris, France

In this paper, we introduce a formalism for explicit definition of least models of Horn theories. This formalism enables the characterisation of these models for a large (algorithmicaly defined) class of logic programs, including both theoretical examples (Horn formulas in solvable classes like extended Skolem class) and small practical ones.

A Query-The-User facility for Logic Programming

Marek Sergot

Department of Computing
Imperial College of Science and Technology
University of London
180 Queen's Gate
London SW7 2BZ England

With the aim of providing declarative input-output for logic programs, a facility is presented which allows a computer system to extract information from the user in the same way the user extracts information from the system. All man-machine communication is conducted in a single language. Sample dialogues illustrate the potential of this facility in a number of application areas, particularly for education and for the construction of expert systems. The approach is criticized to identify directions for future work.

A Simple Dialogue in Polish: Interactive Railway Guide

Stanislaw Szpakowicz

Institute of Informatics
Warsaw University
P.O.B. 1210, 00-90 Warszawa Polland

Marek Świdzinski

Institute of Polish Language
Warsaw University
Krakowskie Przedmiescie 26/28,
00-325 Warszawa, Poland

A simple train timetable information system has been implemented in the Prolog programming language. It is a case study in dialogue systems design, in Prolog programming and in the processing of Polish texts. In this paper, the latter aspect of the experiment is brought into focus. The programs is the first one to perform what can be called a full analysis of Polish sentences.

Object Oriented Programming in Concurrent Prolog

Ehud Shapiro

Department of Applied Mathematics
Weizmann Institute of Science
Rehovot 76100, Israel

Akikazu Takeuchi

Research Center
Institute for New Generation
Computer Technology
Mita-Kokusai buildinf, 21F.
4-28, Mita 1-chome, Minato-ku, Tokyo 108
Japan

It is shown that the basic operations of object-oriented programming languages — creating an object, sending and receiving messages, modifying an object's state, and forming class-superclass hierarchies — can be implemented naturally in Concurrent Prolog. In addition, a new object-oriented programming paradigm, called incomplete messages, is presented. This paradigm subsumes stream communication, and greatly simplifies the complexity of programs defining communication networks and protocols for managing shared resources. Several interesting programs are presented, including a multiple-window manager. All programs have been developed and tested using the Concurrent Prolog interpreter described.

Interprocess Communication in Concurrent Prolog

Akikazu Takeuchi, Kouichi Furukawa

Research Center
Institute for New Generation Computer Technology
Mita-Kokusai Building, 21F.
4-28, Mita 1-chome, Minato-ku, Tokyo 108
Japan

Concurrent Prolog is a logic-based concurrent programming language which was designed and implemented on DEC-10 Prolog by E. Shapiro. In this paper, we show that the parallel computation in Concurrent Prolog is expressed in terms of message passings among

distributed activities and that the language can describe parallel phenomena in the same way as actor-formalism does. Then we examine the expressive power of communication mechanism based on shared logical variables and show that the language can express both unbounded buffer and bounded buffer stream communication only by read-only annotation and shared logical variables. Finally the new feature on Concurrent Prolog is presented, which will be very useful in describing the dynamic formation and reformation of communication network.

ESP as a Preliminary Kernel Language of Fifth Generation Computers

Takashi Chikayama

Institute for New Generation Computer Technology Research Center
Mita Kokusai Building, 21F.
4-28, Mita 1-chome, Minato-ku, Tokyo 108
Japan

In the first three-year development stage of the fifth generation computer systems project, a series of high-performance personal computers called sequential inference machines are being developed at ICOT Research Center. The machines have a high-level machine language called KL0, which is a PROLOG-based logic language with various extensions. In the software development of the sequential inference machines, ESP, a software-supported yet higher level language copiled into KL0, is used instead of directly using KL0. This paper describes the design the language system of sequential inference machines. Description will be centralized on ESP with an overview of KL0.

Learning Equation Solving Methods from Examples

Bernard Silver

Department of Artificial Intelligence
University of Edinburgh, UK

This paper describes LP, a PROLOG program which learns new techniques for solving symbolic equations. LP learns the new techniques by examining worked examples. These techniques can then be tested on some new problems.

The equations used by LP are taken from A level mathematics papers. (A levels are exams taken at 18 and are used for university selection). Other work in this field has concentrated on much simpler equations.

LP uses techniques from the planning field, as well as more traditional learning methods.

The Personal Sequential Inference Machine (PSI): Its Design Philosophy and Machine Architecture

Hiroshi Nishikawa, Minoru Yokota, Akira Yamamoto, Kazuo Taki, Shunichi Uchida

Institute for New Generation Computer Technology
Mita-Kokusai Building, 21F.
4-28, Mita 1-chome, Minato-ku, Tokyo 108
Japan

As a software development tool of the Fifth Generation Computer Systems (FGCS) project, a personal sequential inference machine is now being developed. The machine is intended to be a workbench to produce a lot of software indispensable to our project. Its machine architecture is dedicated to effectively execute a logic programming language, named KL0, and is equipped with a large main memory, and devices for man-machine communication. We estimatc its execution speed is about 20K to 30K LIPS. This paper presents the design objectives and the architectural features of the personal sequential inference machine.

The Proposal of Prolog Machine Based on Reduction Mechanism

R. Onai, H. Shimizu, N. Ito, K. Masuda

First Research Laboratory
Research Center
ICOT, Japan

When we look at the executing process of a Prolog program, we find the close similarity between the process and graph reduction. Therefore, we design the Prolog machine based on graph reduction mechanism.

There are two kinds of parallel execution. One is And-parallel execution and the other is Or-parallel execution. In And-parallel execution, if there is a variable shared among And-literals, we have to check the consistency in solutions of the shared variable. This check is complicated.. On the other hand, there happens resource explosion in Or-parallel. Though we should consider the both parallel execution, in the beginning we realize Or-parallel execution by picking up control of the reducible packets and the swapping control.

This memo proposes the reduction machine which executes Prolog programs in Or-parallel. The machine features are as follows:

(1) Packet (with tag) communication method is adopted in order to realize highly distributed computation.

(2) Or-literals are executed in parallel and And-literals are executed sequentially.

(3) A processor and a memory are divided into some banks for highly parallel processing.

This memo describes

(1) four kinds of memories for packets, caluse information, and structure data
(2) two kinds of subunits for unification
(3) three kinds of networks
(4) one simple execution example.

Using Proof Plans to Control Deduction

Lincoln Wallen

Department of Artificial Intelligence
Edinburgh, UK

This paper describes aspects of MT; an experimental theorem proving system developed to investigate ways of controlling the process of deduction, Proof plans, constructed on the basis of properties of the conjecture to be proved, provide the system with guidance at two distinct levels. At the global level the proof plan indicates how the main proof may be decomposed into several component proofs, each carried out in a separate proof module. At the local level the proof plan introduces constraints on the form of proof tree generated within each proof module. This reduces the complexity of the proofs required and

allows or the application of special purpose methods to isolated areas of the main proof. An example proof plan for a simple proof by induction is developed to illustrate how useful patterns of control may be specified within the system.

A Subset of Concurrent Prolog and Its Interpreter

Ehud Y. Shapiro

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

Concurrent Prolog is a variant of the programming language Prolog, which is intended to support concurrent programming and parallel execution. The language incorporates guarded-command indeterminacy, dataflow-like synchronization, and a commitment mechanism similar to nested transactions.

This paper reports on a subset of Concurrent Prolog, for which we have developed a working interpreter. It demonstrates expressive power of the language via Concurrent Prolog programs that solve benchmark concurrent programming problems. It describes in full detail an interpreter for the language, written in Prolog, which can execute these programs.

Prolog Compared With LISP?

R. A. O'Keefe

DAI Edinburgh, UK

In the recent ACM Symposium on LISP and Functional Programming, there was a paper with the title "Prolog Compared With LISP". In it, Gutierrez presents a program in LISP, and a related program in Prolog, and uses the inferior performance of the latter to suggest in strong terms that advocates of Prolog may have over-stated its performance. However, his program makes very poor use of Prolog. In this article, I point out which features of Prolog have been misused and give guidelines for their proper use. I also compare a new Prolog and LISP program performing a similar task, and find that the execution times are comparable. This is in accord with earlier results.

Outline of PSI

S. Uchida, et al.

ICOT, Japan

PSI is a personal computer system being developed as a tool for providing researchers with an efficient programing environment. It directly supports a logic programming language, KL0 (Fifth Generation Kernel Language, Version 0), with firmware and hardware.

Its interpreter is implemented in the firmware and several hardware mechanisms are provided to attain almost the same levels of performance as the DEC-10 Prolog on DEC2060. It also provides the user with a large memory space, 40 bits X 2 to 16 MW, which is essential for developing actual application programs like as expert system.

To make efficient man-machine interaction possible, such input and output devices as bit-map display, pointing device are key-board are provided. A local area network is also beis developed to build a distributed system.

A relational Database Machine "Delta"

Shigeki Shibayama, Takeo Kakuta, Nobuyoshi Miyazaki, haruo Yokota, Kunio Murakami

Institute for New Generation
Computer Technology
Japan

Japan's Fifth Generation Computer System (FGCS) project is scheduled to be a ten-year-long activity. Knowledge Base Machine is one of its expected achievements in that research period. When completed, KBM (Knowledge Base Machine) will be seen as an Intelligent Knowledge store, which co-operates interactively with the inference-based new architecture computers, to provide users with natural and intelligent interfaces to the consolidated system.

In our first-stage three-year project, the primary object of KBM (Knowledge Base Machine) Group is to provide a working Relational Database Machine which serves multiple of Sequential Inference Machine (SIM) users via a local area network (LAN). Investigation for the methods to amalgamate database management system (DBMS) and logic programming language, represented now by Prolog, is also an important subject which we are now greatly interested.

Japan's Fifth Generation Computer Project — A trip report

Ehud Y. Shapiro

Department of Applied Mathematics
Weizmann Institute of Science
Rehovot 76100, Israel

Last April Japan's Ministry of International Trade and Administration (MITI), in cooperation with eight leading computer companies, launched a research project to develop computer systems for the 1990's. The project, called the Fifth Generation Computers Project, will span 10 years. Its ultimate goal is to develop integrated systems — both hardware and software — suitable for the major computer application for the next decade, identified by the Japaneses as "Knowledge Information Processing". Even though it may ultimately have applicable results, the current focus of the project is basic research, rather than the development of commercial products.

In addition to bringing Japan into a leading position in the computer industry, the project is expected to elevate Japan's prestige in the world. It will refute accusations that Japan is only exploiting knowledge imported from abroad, without contributing any of its own to benefit the rest of the world. Hence the project aims at original research, and plans to make its results available to the international research community.

I was the first non-Japanese researcher invited for a working visit to ICOT, the Institute for New Generation Computer Technology, which conducts the project. Due to the nature of the project I was given explicit permission, even encouragement, to report on everything I saw and heard during my visit; hence this report.

LOFE: A Language for Virtual Relational Data Base

J. K. Debenham and *G. M. McGrath*
School of Computing Sciences
NSW Institute of Technology
P.O. Box 123, Broadway
NSW 2007, Australia and Telecom. Australia

It is assumed that the reader will be familiar with logic as a data base language, as discussed for example by Gallaire, Minker and Nicholas (1978) of Dahl (1982).

In this paper we describe our front end language; LOFE, which is based on logic and designed for virtual relational data base. It is well-known that data base up-dates, deletes and integrity checks can be phrased in logic (Gallaire, Minker and Nicholas, 1978): we will not discuss them. We will concentrate solely on the query facilities in our language.

First, we discuss the role of front end languages in data base, and note criteria for their design. We then define LOFE, a powerful data base query language for the high level user. A declarative interpretation of LOFE is given; this is illustrated with examples. Then the imperative interpretation is discussed and comments are made on an experimental implementation. Finally we show that a natural restriction of LOFE provides an adequate and simple language for the low level user, and that this restriction will prevent inefficient usage.

Protocol Verification via Executable Logic Specifications

Deepinder P. Sidhu
Research and Development Division
SDC — A Burroughs Company
Paoli, PA 19301, USA

This paper discusses the use of logic programming techniques in the specification and verification of communication protocols. The protocol specifications discussed are formal and directly executable. The advantages of executable specifications are: (1) the specification is itself a prototype of the specified system, (2) incremental development of specifications is possible, (3) behavior exhibited by the specification when executed can be used to check conformity of specification with requirements. We discuss Horn clause logic, which has a procedural interpretation, and the predicate logic programming language, PROLOG, to specify and verify the functional correctness of protocols. The PROLOG system possesses a powerful pattern-matching feature which is based on unification.

Logic and Programming Methodology

E. W. Elcock
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A 5B9

The intent of the paper is to expose and compare different methodologies in the context of the weell-know and pedagogically attractive eight-queens problem. In particular, the paper attemps to exhibit the richness of the space of logical transforms of the problem specification and illustrates how suchrichness might be exploited to obtain different methodological objectives.

Although few of the threads are novel, the author hopes that the fabric will be found pleasing.

Goal Selection Strategies in Horn Clause Programming

E. W. Elcock
University of Western Ontario
Canada

It is arguable that knowledge representation and use should be founded on a complete system. For example, if the knowledge representation language is to be first order logic, then we would like to express the knowledge *K* under the assumption that we have available a sequenthood procedure which is complete in the sense that any true sequent $K =< G$ is demonstrably true by the procedure. For example, our knowledge system might be based on finite clausal sequents for which there is indeed a procedure using resolution which has the completeness property.

For resolution systems it is well known that selection strategies play a vital role in determining the pragmatics of such systems and design of strategies has been an ongoing research activity.

Over the last decade an incomplete system called Prolog has been elaborated and has become hidely used. Prolog has intriguing analogies with Absys — an assertive programming system developed in 1968 by Foster and Elcock. This note attemps to illustrate some issues of incompleteness by comparing some aspects of the two systems.

On Database Systems Development Through Logic

Veronica Dahl
Buenos Aires University, Argentina

The use of logic as a single tool for formalizing and implementing different aspects of database systems in a uniform manner is discussed. The discussion focuses on relational databases with deductive capabilities and very high-level querying and defining features. The computational interpretation of logic is briefly reviewed, and then several pros and cons concerning the description of data, programs, queries, and language parser in terms of logic programs are examined. The inadequacies are discussed, and it is shown that they can be overcome by the introduction of convenient extensions into logic programming. Finally, an experimental database query system with a natural language front end, implemented in PROLOG, is presented as an illustration of these concepts. A description of the latter from the user's point of view and a sample consultation session in Spanish are included.

Translating Spanish into Logic through Logic

Veronica Dahl
Department of Mathematics
University of Buenos Aires
Buenos Aires, Argentina

We discuss the use of logic for natural (NL) processing, both as an internal query language and as a programming tool. Some extensions of standard predicate calculus are motivated

by the first of these roles. A logical system including these extensions is informally described. It incorporates semantic as well as syntactic NL features, and its semantics in a given interpretation (or data base) determines the answer-extraction process. We also present logic-programmed analyser that translates Spanish into this system. It equates semantic agreement with syntactic well-formedness, abd can detect certain presuppositions, resolve certain ambiguities and reflect relations among sets.

Automatic Generation of Explanations of Results from Knowledge Bases

Adrian Walker

IBM Research Laboratory
San Jose, California 95193, USA

It has been pointed out that advice from a knowledge base may only useful if the reasons for the advice can be explained. For example, given the advice "sell all your stock and invest in Corporation X", most people would want reasons.

We consider knowledge based programs written in the language Prolog, and we give a source-source transform which causes such programs to produce explanations of their own results. The transform, which is itself written in Prolog, can reduce or eliminate the programming which is normally needed to provide explanations. Control is provided over the level of detail in an explanation. A maximum-detail explanation is a proof. We give an example in the domain of airline flight booking.

Towards a Programming Language Based on the Notion of Two-Level Grammar

Jan Maluszynski

Software Systems Research Center
Linköping University
581 83 Linköping, Sweden

Institut of Computer Science
Polish Academy of Sciences
00-901 Warsaw PKIn P.O. Box 22, Poland

The paper deals with the problem of computing relations from their abstract non-algorithmic specifications. The formalism under consideration is that of two-level grammars, introduced originally for defining languages. In this formalism the intuitions behind the formal definition can be directly expressed by the grammatical rules, which may have a form close to the statements of natural language. A notion of the relation specified by a two-level grammar is introduced and computability of such relations is discussed. For a class of two-level grammars, called the transparent grammars, an algorithm is outlined for computing the relations specified by the grammars of this class. The transparent grammars turn out to be a generalization of the formalism of Horn clauses, and the algorithm is based on unification. The language generated by a two-level grammar can be used as an additional tool for controlling computations. A transparent two-level grammar can be considered a non-algorithmic program specifying an input/ /output relation. The computational algorithm defines an operational semantics of such programs.

Grammatical Unification

Jan Maluszynski

Software Systems Research Center
Linköping University
581 83 Linköping, Sweden

Institute of Computer Science
Polish Academy of Sciences
00-901 Warsaw PKIN P.O. Box 22, Poland

Jorgen Fischer Nilsson

Department of Computer Science
Technical University of Denmark
Building 343 and 344
2800 Lyngby, Denmark

This paper presents a generalization of the concept of unification introduced by Robinson for resolution logic. Unification is the central procedure for performing manipulation of symbolic structures in resolution theorem proving. Our interest in unification relates however in particular to Horn clause logic programming and more specifically to Prolog systems.

The structures unified in logic programming systems are atoms and terms constructed in the standard way from predicates and functors. The Herbrand universe of a given logic program can be specified by an unambiguous context-free grammar constructed in a standard way.

This grammar dtermines also the syntax of non-ground terms in a sense made clear below. The generalized unification discussed in this paper, called grammatical unification, applies to strings whose structure is determined by an arbitrary unambiguous context-free grammar. It is shown that grammatical unification can be reduced to usual unification. This makes it possible to define a version of Prolog in which the syntax of atoms and terms for some particular application could be specified by the user by means of an arbitrary unambiguous context-free grammar. In this case a logic program could apply to the specific data representations used in the application area.

Dérivation de Programmes Prolog à Partir de Spécifications Algébriques

P. Deransart

Inria
Domaine de Voluceau
BP 105
78153 Le Chesnay, France

Une méthode de dérivation systématique de programme PROLOG à partir de spécification algébriques dites canoniques est présentée. Les programmes obtenus ont un comportement équivalent à celui de la spécification. Ceci permet soit de prouver des propriétés du programme dérivé, soit de prévoir le comportement du programme, en particulier dans les cas délicats où la "résersibilité" du programme est requise.

LISLOG Programmation en Logique en Environnement LISP

S. Bourgault, M. Dincbas, D. Feuerstein

CNET — Lannion/SLC, France

LISLOG se compose d'un interpréteur du langage PROLOG étendu, écrit en LISP ainsi

qu'un éditeur de clauses permettant, sous LISLOG, la création ou la modification des bases de connaissances, clauses, littéraux ou termes. En outre, LISLOG offre la possibilité d'appeler des programmes LISP à l'intérieur des clauses PROLOG.

Ce papier présente brièvement le langage et l'utilisation du système.

Problèmes de Gestion de Mémoire dans les Interpreteurs Prolog

Y. Bekkers, B. Canet, L. Ungaro

INRIA — IRISA Rennes, France

Après une brève présentation du langage Prolog (amputé des prédicats évaluables), nous exposons les problèmes de représentation et de récupération de mémoire. L'accent est mis sur la complexité introduite par l'indéterminisme.

Présentation de PROLOG/CNET Version Pascal/Multics

G. Barberye, T. Joubert, M. Martin

Centre National d'Études des Telecommunications
92131 Issy-les-Moulineaux, France

Dans le cadre des études sur les langages de spécification et outils associés, le CNET Paris A s'intéresse depuis maintenant deux ans aux systèmes PROLOG developpés par le GIA de Marseille-Luminy.

L'outil recherché par l'équipe était alors un système interactif capable d'effectuer du "calcul formel" sur des "arborescences".

Au début de l'année 80, l'équipe s'est familiarisée avec PROLOG dans son implémentation FORTRAN/IRIS 80; cette version a permis de voir les bons côtés du langage et les moins bons qui pouvaient facilement être mis sur le "compte" de l'implémentation.

Ainsi, nous avons acquis une version plus récente de PROLOG fonctionnant sur Exorciser Motorola et introduit un certain nombre de facilités sur cette dernière version.

C'est à l'aide de ce "système" que nous avons développé un outil appelé OASIS intégralement écrit en PROLOG et sur lequel nous avons "traité" des spécifications et des représentations de types abstraits.

Le système de développement Exorciser II se montrant insuffisant pour la poursuite de nos investigations (volume mémoire insuffisant, temps de traitement trop longs ...) nous avons décidé de transporter l'application OASIS et le langage qui lui servait de support: PROLOG.

C'est de cette façon qu'est née la version PROLOG/PASCAL définie.

PROLOG: A Tutorial Introduction

R. A. Sammut and *C. A. Sammut*

Department of Mathematics
and Computer Science
St. Joseph's University
5600 City Road, Philadelphia, PA 19131, USA

PROLOG is a language which realizes the concept of using predicate logic as a programming language. Since its first implementation approximately ten years ago, it has found applications in a variety of "symbol processing" areas such as natural language processing, deductive information retrieval, compiler writing, symbolic algebra, computer-aided design and robot problem-solving.

This paper introduces the fundamental concepts which are unique to programming in PROLOG by developing and analyzing a series of small programs for deductive information retrieval, the solution of the "N-queens" problem and a simple exercise in computer-aided design.

Optimal Fixedpoints of Logic Programs

J.-L. Lassez and *M. J. Maher*

Department of Computer Science
University of Melbourne
Parkville, Victoria, 3052, Australia

From a declarative programming point of view, Manna and Shamir's optimal fixedpoint semantics is far more appealing than the least fixedpoint semantics. However in standard formalisms of recursive programming the optimal fixedpoint is not computable while the least fixedpoint is. In the context of logic programming we show that the optimal fixedpoint is equal to the least fixedpoint and is computable. Furthermore the optimal fixedpoint semantics is consistent with Van Emden and Kowalski's semantics of logic programs.

An Approach to Proving Properties of Non-Terminating Logic Programs

Paolo Ciancarini, Pierpaolo Degano

ISI — Universitá di Pisa
Pisa, Italy

This paper concerns proving properties of first order logic processes, i.e. programs that perform non-terminating computations on infinite data structures. It is based on the concept of symbolic computation cycle, and states that a property holds when it is invariant with respect to a symbolic computation cycle.

Experimenting on Euclidean Domains With a Non-Deterministic Programming Language Based on First Order Logic: PROLOG

Regina Llopis de Trias

División de Matemáticas
Universidad Autónoma
Canto Blanco
Madrid 34, Spain

Blanca Ortega de Zubizarreta

Departamento de Matemáticas Y Ciência
de la Computación
Universidad Simón Bolivar
Caracas, Venezuela

The programming language PROLOG based on first order logic and its clausal syntax can be used in a natural fashion in a symbolic and algebraic system in computers. Its built-in ASSERT and ABOLISH procedures can create an environment for the notions of algebraic domains and categories. An initial algebraic modular system was created working with the euclidean domains integers and polynomials.

A multivariate polynomial manipulator was defined using PROLOG clauses which are more readable than LISP defined lambda-functions. Factorisation in the euclidean domains $\mathbb{Z}$ and Q [x] was done by means of similar algorithms using the non-deterministic capabilities of PROLOG. berlekamp's polynomial factorisation algorithm was implemented and compared with the "non-deterministic" one on polynomials of small degree over the Galois Fields $\mathbb{Z}_2$ and $\mathbb{Z}_3$.

abstracts

The Description in Logic of Large Commercial Data Bases: A Methodology Put to the Test

J. K. Debenham Nswit and *G. M. McGrath*

Telecom., Australia

First-order predicate logic may be interpreted as a programming language: at present there are a variety of interpreters and compilers for this language, PROLOG, as well as a rapidly growing community of users. First-order predicate logic may also be interpreted as a data base language: a so called Virtual Relational Data Base or VRDB. This second interpretation has not attracted as much interest as the former because (i) there was no complete methodology available, and (ii) there are no direct implementations of VRDB's available for general use at present.

In this paper we refine the data base interpretation of logic explicitly in terms of existing DBMS: thus making practical experimentation possible. A database design methodology is presented. This methodology may be used to systematically design models of data specified in logic and convert from specifications to implemented data bases. The methodology has been employed in the design of two large data bases.

Results of these experiments are indicated.

Completeness and Confluence Properties of Kowalski's Clause Graph Calculus

Gert Smolka

Universität Karlsruhe
Fakultät für Informatik
Postfach 63 80, D 7500 Karlsruhe 1
West Germany

This internal report consists of two parts. The first part outlines the main results of my thesis. Further results and the full proofs are given in the second part which is a reprint of the thesis in German language.

As main result it is shown that R. Kowalski's connection graph proof procedure terminates with the empty clause for every unit-refutable clause set, provided that an exhaustive search strategy is employed. This result holds for unrestricted tautology deletion, whereas subsumption requires certain precautions.

The results are shown for an improved version of the connection graph resolution rule which generates fewer links than the original one. The new inference rule not only leads to a smaller search space but it also permits a more efficient implementation.

The proofs are based on refutation trees and are applied immediately at the general level. Hence the unsolved problems resulting from the classical lifting techniques in the context of clause graphs are avoided.

Several counterexamples are presented at the propositional level. For instance it is shown that unrestricted deletion of tautologies destroys completeness for non-unit-refutable clause sets. This also holds. if tautology deletion is restricted as proposed by Bibel.

Finally the confluence of Kowalski's calculus at the propositional level is shown. This proof is based on Bibel's spanning property.

Intelligent Backtracking in Plan-Based Deduction

Stanislaw Matwin, Tomasz Pietrzykowski

Universities of Ottawa and Acadia
Canada

This paper develops a method of mechanical deduction based on graphical representation of the structure of proofs. Attempts to find refutation(s) are recorded in the form of plans, corresponding to a portion of AND/OR graph search space and representing purely deductive structure of derivation.

This method can be applied to any initial base (set of non-necessarily Horn clauses). Unlike the exhaustive (blind) backtracking which treats all the goals deducted in the course of proof as equally probable source of failure, this approach detects the exact source of failure.

In this algorithm only a small fragment of solution space is kept on disk as a collection of pairs, each of which consists of a plan and a graph of constraints. The search strategy and the method of non-redundant processing of individual pairs which leads to a solution (if it exists) is presented. This approach is compared — on a special case — with blind backtracking and an exponential improvement is demonstrated.

Some important implementation problems are discussed and top-level design of the system is presented.

It is proven that the algorithm is partially complete in the following sense: if for a given base a refutation exists, then — provided the algorithm terminates — this refutation is found.

Machine PROLOG pour les Applications d'Intelligence Artificielle: Gestion de Mémoire

Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro

IRISA
Avenue du Général Leclerc
35042 Rennes Cedex, France

Ce papier étudie les problèmes de représentation d'information et de gestion de mémoire dans les interpréteurs PROLOG. L'accent est mis sur la récupération. Des solutions sont présentées comme étude préliminaire à la réalisation d'une machine PROLOG pour les applications d'intelligence artificielle.

Combinatorially Implosive Algorithms

William A. Kornfeld

MIT AI LAB
Cambridge, MA, USA

Applications of parallel processing languages to reducing the average time behavior of search algorithms are discussed. It is argued that a parallel algorithm can dramatically reduce the average time behavior even if the algorithm is run in a time-slicing fashion on a single processor. A language is developed with primitives to facilitate the construction of algorithms of this type.

Equality for Prolog

William A. Kornfeld

MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139, USA

The language Prolog has been extended by allowing the inclusion of theorems about equality. When a unification of two terms that do not unify syntactically is attempted, an equality theorem may be used to prove the

two terms equal. If it is possible to prove that the tgwo terms are equal the unification succeeds with the variable bindings introduced by the equality proof. It is shown that this mechanism significantly improves the power of Prolog. Sophisticated data abstraction with all be advantages of object-oriented programming is available. techniques for passing partially instantiated data are described that extends the "multi-use" capabilities of the language, improve the efficiency of some programs, and allow the implementation of arithmetic relations that are both general and efficient. The modifications to standard Prolog are simple and straightforward and in addition the computational overhead for the extra linguistic power is not significant. Equality theorems will probably play an important role in future logic programming systems.

Logic for Natural Language Analysis

Fernando Pereira

Artificial Intelligence Center
Computer Science and Technology Division
SRI International, USA

This work investigates the use of formal logic as a practical tool for describing the syntax and semantics of a subset of English, and building a computer program to answer data base queries expressed in that subset.

To achieve an intimate connection between logical descriptions and computer programs, all the descriptions given are in the definite clause subset of the predicate calculus, which is the basis of the programming language Prolog. The logical descriptions run directly as efficient Prolog programs.

Three aspects of the use of logic in natural language analysis are covered: formal representation of syntactic rules by means of a grammar formalism based on logic, extraposition grammars; formal semantics for the chosen English subset, appropriate for data base queries; informal semantic and pragmatic rules to translate analysed sentences into their formal semantics.

On these three aspects, the work improves and extends earlier work by Colmerauer and others, where the use of computational logic in language analysis was first introduced.

Implementing Clauses Indexing in Deductive Database Systems

John W. Lloyd

Department of Computer science
University of Melbourne
Australia

The paper presents a file design for handling partial-match queries which has wide application to knowledge-based artificial intelligence systems and relational database systems. The advantages of the design are simplicity of implementation, the ability to cope with dynamic files and the ability to optimize performance with respect to the average number of disk access required to answer a query.

Partial-Match Retrieval for Dynamic Files

John W. Lloyd, K. Ramamohanarao

Department of Computer Science
University of Melbourne
Australia

This paper studies file designs for answering partial-match queries for dynamic files. A partial-match query is a specification of the value of zero or more fields in a record. An answer to a query consists of a listing of all records in the file satisfying the values specified.

The main contribution is a general method whereby certain primary key hashing schemes can be extended to partial-match retrieval schemes. These partial-match retrieval designs can handle arbitrarily dynamic files and can be optimized with respect to the number of page faults required to answer a query.

We illustrate the method by considering in detail the extension of two recent dynamic primary key hashing schems.

The file designs have application to clause indexing in deductive databases.

Dynamic Hashing Schemes

K. Ramamohanarao, John W. Lloyd

Department of Computer Science
University of Melbourne
Australia

In this paper, we study two new dynamic hashing schemes for primary key retrieval. The schemes are related to those of Scoll, Litwin and Larson. The first scheme is particularly simple and elegant and has certain performance advantages over earlier schemes. We give a detailed mathematical analysis of this scheme and also present simulation results. The second scheme is essentially that of Larson. However, we have made a number of changes which considerably simplify his scheme.

The file designs have application to clause indexing in deductive databases.

Partial-Match Retrieval Using Hashing and Descriptors

K. Ramamohanarao, John W. Lloyd, James A. Thom

Department of Computer Science
University of Melbourne
Australia

This paper studies a partial-match retrieval scheme based on hash functions and descriptors. The emphasis is placed on showing how the use of a descriptor file can improve the performance of the scheme. Records in the file are given addresses according to hash functions for each field in the record. Furthermore, each page of the file has associated with it a descriptor, which is a fixed length bit string, determined by the records actually present in the page. Before a page is accessed to see if it contains records in the answer to a query, the descriptor for the page is checked. This check may show that no relevant records are on the page and hence, the page does not have to be accessed. The method is shown to have a very substantial performance advantage over pure hashing schemes, when some fields in the records have large key spaces. A mathematical model of the scheme, plus an algorithm for optimizing performance, is given. The file design has application to clause indexing in deductive databases.

An Introduction to Deductive Database Systems

J. W. Lloyd

Department of Computer Science
University of Melbourne
Australia

This paper gives a tutorial introduction to deductive database systems. Such systems

have developed largely from the combined application of the ideas of logic programming and relational databases. The elegant theoretical framework for deductive database systems is provided by first order logic. Logic is used as a uniform language for data, programs, queries, views and integrity constraints. It is stressed that it is possible to build practical and efficient database systems using these ideas.

Completeness of the Negation as Failure Rule

Joxan Jaffar, Jean-Louis Lassez, John W. Lloyd

Department of Computer Science
University of Melbourne
Australia

Let *P* be a Horn clause logic program and *comp(P)* be its completion in the sense of Clarck. Clark gave a justification for the negation as failure rule by showing that if a ground atom *A* is in the finite failure set of *P*, then ˜*A* is a logical consequence of *comp(P)*, that is, the negation as failure rule is sound. We prove here that the converse also holds, that is, the negation as failure rule is complete.

A Qualitative Model of the Heart for a Medical Expert System

I. Mozetic, I. Bratko, N. Lavrac, T. Zrimec

Institute Jozef Stefan,
Jamova 39, Ljubljana, Yugoslavia
Faculty of Electrical Engineering,
Trzaska 25, Ljubljana
Iskra, Trzaska 2, Ljubljana

The paper describes the diagnostic part of an expert system for the diagnosis and treatment of heart arrhythmias. The ECG interpretation module includes a qualitative model of the heart. The model was used to automatically generate a rule-base which relates an exhaustive dictionary of phisiologically possible combinations of heart arrhythmias to their corresponding ECG descriptions. In addition to the applicability of this knowledge base in the expert system, this result is of interest for the systematisation of medical knowledge.

Expert Systems and Prolog

I. Bratko, Josef Stefan
Institute, Ljubljana, Yougoslavia

The basic features of rule-based systems, pattern-directed, architecture, and PROLOG are discussed, and their uses in the design and implementation of Expert Systems are assessed. The rule-based representation of knowledge is examined with particular reference to "if-then" rules. Rule systems are closely related to a relatively new architecture of software systems called pattern-directed systems and these are discussed in detail. PROLOG, a non-procedural programming language, is well suited to solving problems involving objects and relations between objects and examples are given in order to illustrate the nature of PROLOG programming. Finally, some implementation and applications of PROLOG are given.

Location of Logical Errors in PASCAL Programs with an Appendix on Implementation Problems in Waterloo PROLOG/C

Scott Renner

Semester Report — Fall 1981
Computer Science 397, Section DM
University of Illinois, USA

The KBPA Research Project includes among its objectives the development of programs to aid in the location of logical errors in PASCAL programs. We describe some of the research done in this area in the Fall 1981 semester, and present possible plans for further work next semester.

Trials of one of the main algorithms were made in Waterloo PROLOG/C and problems encountered in this otherwise convenient system are listed in Appendix A.

A Comparison of PROLOG Systems Available at the University of Illinois

Scott Renner

KBPA Project
University of Illinois, USA

In this report we will summarize our findings from our study of four Prolog systems which are available to members of the KBPA project. We are particularly interested in how useful these systems are in implementing Ehud Shapiro's bug locating algorithms (Shapiro, 1981). A fifth Prolog system, Waterloo PROLOG/IBM 370, has beeen acquired by the project. As of this date, it has not been tested, and so it is not included in this report.

Logic Programs With Uncertainties: A Tool for Implementing Rule-Based Systems

Ehud Y. Shapiro

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

One natural way to implement rule-based expert systems is via logic programs. The rules in such systems are usually definite clauses, or can easily be expressed as such, and the inference mechanisms used by such systems are built into the Prolog interpreter, or can be implemented in Prolog without much effort.

The one component of expert systems which is not readily available in logic programs is a language for specifying certainties of rules and data, and a mechanism for computing certainties of conclusions, given certainties of the premises. Clark and McCabe suggest an implementation technique for solving this problem. They augment each predicate in the rule-language with an additional argument, whose value is the certainty of the solution returned in this predicate, and augment the condition of each clause with an additional goal, whose purpose is to compute the certainty of the conclusion of the clause, given the certainties of solutions to the rest of the goals in the condition of the clause.

In this paper we propose a different way of implementing rule-based expert systems within Prolog, in which evaluation of certainties of solutions is carried out at the metalevel, within the logic program interpreter itself. This resulting framework, called *logic programs with uncertainties,* has the following properties:

- It is amenable to theoretical analysis. In particular, a precise semantics can be given to logic programs with uncertainties.
- Standard logic programs are a special case of logic programs with uncertainties. If all

certainty factors are 1, then the semantics defined and the interpreters developed degenerate to the standard semantics and the standard interpreter for logic programs.
- Since the semantics of logic programs with uncertainties is simple, it is easy to apply the debugging algorithms developed.

An Intelligent Backtrack Algorithm for the Communicating Processes of TS-PROLOG

Iván Futó

Inst. for Coordination of Computer techn.
H-1368 Budapest, POB.224, Hungary

An intelligent backtrack algorithm under implementation for the communicating processes of the AI system TS-PROLOG is presented. TS-PROLOG is a PROLOG based discrete simulation and problem solving system. Models in TS-PROLOG are defined using Horn-clauses and a model consists of components and elementary components having their own goals and activities. To every component a process is assigned and during the problem solving these processes communicate with each other. If deadlock situation occurs or data are missing the processes return to a previous state to try new possible alternatives. This backtrack is organised in an *intelligent* way, only those processes are involved which were in *explicitly* or *implicitly* communication with the originally failed one. This decreases considerably the search space of the problem.

Building Libraries in Prolog

Alan Feuer

Bell Laboratories
Murray Hill, New Jersey 07974, USA

While Prolog has proven useful for writing programs in a variety of domains, it suffers from its lack of support for modularity, particularly for building libraries of routines and data. This paper points out some problems with standard Prolog that make libraries inconvenient. It then describes a solution to those problems based on the concepts of modules and database views.

Knowledge representation in Prolog/KR

Hideyuki Nakashima

Wada Lab., Information Engineering Course
University of Tokyo, Graduate School
Bunkyo-ku, Tokyo, 113 Japan

In knowledge representation, we must represent both data objects and a reasoning system working on them. Represented data themselves are not knowledge. The reasoning system which manipulates those data given it meaning. Therefore, a knowledge representation system must have coherent semantics both in representation and manipulation. In current knowledge representation languages, however, reasoning system and procedural knowledge are often described in another procedural language.

Logic, with its declarative and procedural interpretation, is a good candidate as a base for the coherent semantics. A logic programming language Prolog seems promising. Nevertheless, Prolog is currently far from what is required for a knowledge representation system. It lacks facilities to modularize knowledge, to construct hierarchies of concepts, to deal with incomplete knowledge, and so on.

Prolog/KR provides the multiple world mechanism, which fills the gap between Prolog and knowledge representation. With the mechanism, predicate definitions are grouped into worlds to form concepts. Worlds are then combined to form a hierarchy of concepts.

In this paper, the multiple world mechanism and its usage to form conceptual hierarchy are presented.

"Logal": Prolog plus Algorithmic Control Structures

D. C. Dodson, A. L. Rector, J. B. Booke

Department of Community Health
University of Nottingham Medical School
Queens Medical Centre
Nottingham NG7 2UH

In practical applications of Prolog, many predicates involve difficult and opaque uses of control primitives. To relieve these difficulties, a preliminary set of structured (or high-level) control predicates have been developed.

Two specific technical goals are guiding this work. The first of these is to bring together the structured control facilities found desirable in conventional languages and cast them into a logic-programming form. The second is to provide convenient high-level structures for specifying all the sorts of algorithmic routines that can sensibly be performed within a Prolog clause.

Apart from their distinct benefits in writing Prolog applications, the structures described may help to sell Prolog to prospective users by making a deliberately algorithmic programming style within Prolog more respectable and convenient.

Logic Programming in Metalog

M. J. Schoppers, M. T. Harandi

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfied Ave
Urbana, IL 61801, USA

MetaLog is a new logic programming language. It entirely eliminates the "cut", separates declarative and imperative code, and separates rule selection predicates from problem rduction predicates. MetaLog also introduces several new features, such as a multigoal priority in conflict resolution, a differentiation of rule types, and a dynamically modifiable schedule for subgoal solution.

A PROLOG Basis Expert System

Fumio Mizoguchi, Kazuhiro Miwa

Department of Industrial Administracion
Science University of Tokyo
Noda, Chiba 278, Japan
Phone 9471-24-1501 ex. 256

Koichi Furukawa

Inst. For New Generation Computer Technology
Mita 1-4-28, Minatoku
Tokyo 108, Japan

We have described a PROLOG basis expert system implementation named PBES in the framework of rule-based systems. The points

are a follows; the first point in PBES consist of rule-driven, backward and forward reasoning system. That is, the control of reasoning is dependent on the rule driven process which decides the next premise parts described in Horn Clause. Thus, PBES is implemented in the unified control strategies which facilities both forward and backward reasoning. The second point of this study is to compare the system comparisons which developed in the past. In the case of PROLOG basis system, the control strategy is more flexible than EXPERT and EMYCIN in a sense that the user can select whether the reasoning is backward or forward.

The third point is to apply PBES into the realistic situation such as reactor fault diagnostic system. The results of 20 case studies have whown the good agreements with human expert judgements.

Exeter Prolog — An Experimental Prolog Interpreter Based on Standard Lisp

G. Belovari

Kent State University
Kent, Ohio, USA

R. Fogelholm

Royal Institute of technology
Stockholm, Sweden

Exeter Prolog originally referred to the name of a city in devon, England where some of the major initiatives to this work were made in the early eighties. From here onwards, Exeter stands for Experimental Interpreter (to let those oor founders of the city of Exeter be drawn into the computer age).

We present a preliminary report on our Prolog Standard Lisp project which aims at a throrough understanding of the inner workings of a Prolog evaluator in the context of Lisp. Lisp was found to be as an extremely flexible implementation language: it offers built-in management of identifiers, rich tools for the construction of arbitrary data structures, and the automatic recycling of discarded pieces of data. The Prolog system is expected to run on any other Standard Lisp system, and possibly on other Lisp systems, as well, after some interfacing work.

Our project used mainly the VAX 11/780 computer, under the UNIX operating system.

The report is available from Dr. Fogelholm, R. I. T. Stockholm, Sweden.

PROLOG on the DADO Machine: A Parallel System for High-Speed Logic Programming

Stephen Taylor, Christopher Maio, Salvatore J. Stolfo, David E. Shaw

Department of Computer Science
Columbia University
New York, NY 10027, USA

DADO is a highly-parallel, VLSI-based, tree-structured machine designed to provide significant performance improvements in the execution of large production system programs. In this paper, we describe current research aimed at implementing PROLOG within the parallel framework which DADO provides. The implementation allows parallel satisfaction of both disjunctions and conjunctions which occur in the goal tree generated during the execution of a PROLOG program. Local unification routines in each processor allow parallel satisfaction of disjunctive goals while a parallel relational join operation provides a framework to solve conjunctive subgoals. An overview of the techniques currently being implemented and their relationship to the architecture is presented.

DADO: A Tree-Structured Machine Architecture for Production Systems

Salvatore J. Stolfo, David E. Shaw

Department of Computer Science
Columbia University
New York, NY 10027, USA

DADO is a parallel tree-structured machine designed to provide significant performance improvements in the execution of large Production Systems. The DADO machine comprises a large (on the order of a hundred thousand) set of processing elements (PE's), each containing its own processor, a small amount (2K bytes, in the current design) of local random access memory, and a specialized I/O switch. The PE's are interconnected to form a complete binary tree.

This paper describes a general procedure for the parallel execution of production systems on the DADO machine, and outlines in general terms how this procedure can be extended to include commutative and multiple, independent production systems.

ACE: An Expert System Supporting Analysis and Management Decision Making

Salvatore J. Stolfo

Columbia University

Gregg T. Vesonder

Bell Laboratories

Department of Computer Science
Columbia University
New York, NY 10027, USA

ACE, a system for Automated Cable Expertise, is a Knowledge-Based Expert System designed to provide trouble-shooting reports and management analysis for telephone cable maintenance in a timely manner. Many design decisions faced during the construction of ACE were guided by recent successes in expert systems technology, most notably R1/XCON, the Digital Equipment Corporation Vax configuration program. The most significant departure from "standard" expert systems architectures is ACE's use of a conventional data base management system as its primary source of information. Its primary sources of knowledge are the expert users of the database system, and primers on maintenance analysis strategies. The coupling of "knowledge-base" and "data-base" demonstrates in a forceful way the manner in which an expert system can significantly enhance the throughput and quality of data processing environments supporting business management. However further difficult problems must be solved before the expert system approach becomes a standard technique in the data processing industry.

abstracts

Architecture and Applications of DADO: A Large-Scale Parallel Computer for Artificial Intelligence

Salvatore J. Stolfo, Daniel Miranker, David Elliot Shaw

Dept. of Computer Science
Columbia University
New York, NY 10027, USA

As part of our research on parallel architecture and VLSI systems, we have been investigating machine architectures specially adapted to the highly efficient implementation of AI software. In the course of our research we designed DADO, a highly-parallel, VLSI-based, tree-structured machine, and implemented an optimal running time algorithm for Production Systems on a simulator for DADO. Subsequent research has convinced us that DADO can support many other AI applications including the rapid execution of PROLOG programs, as well as a large share of the symbolic processing typical of knowledge-based systems.

In this brief report, we outline the hardware design of a moderate size DADO prototype, comprising 1023 processing elements, currently under construction at Columbia University. We then sketch the software base being implemented on a small 15 element system, which will very soon be operational, including several applications written in PPL/M, a high-level language designed for specifying parallel computation on DADO. Several applications actively under investigation are then briefly described.

Knowledge Retrieval as Limited Inference

Alan M. Frisch, James F. Allen

Computer Science Department
The University of Rochester
Rochester, NY 14627, USA

Artificial intelligence reasoning systems commonly employ a knowledge base module that stores declarative knowledge and provides retrieval facilities. A retriever could range from a simple pattern matcher to a complete logical inference system. In practice, most fall in between these extremes, providing some forms of inference but not others. Unfortunately, most of these retrievers are not precisely defined.

We view knowledge retrieval as a limited form of inference operating on the stored knowledge. This paper is concerned with our method of using first-order predicate calculus to formally specify a limited inference mechanism and to a lesser extent with the techniques for producing an efficient program that meets the specification. Our ideas are illustrated by developing a simplified version of a retriever used in the knowledge base of the Rochester Dialog System. The interesting property of this retriever is that it performs typical semantic network inferences such as inheritance but not arbitrary logical inferences such as modus ponens.

Some Issues of Problem Specification in First-Order Logic

E. W. Elcock

Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A 5B9

Informal overview of the logical space of the queens problem.

R. Fagerholm

Royal Institute of technology
Stockholm, Sweden

Exeter Prolog originally referred to the name of a city in devon, England where some of the major initiatives to this work were made in the early eighties. From here onwards, Exeter stands for Experimental Interpreter (to let those our founders of the city of Exeter be drawn into the computer age).

We present a preliminary report on our Prolog Standard Lisp project which aims at a thorough understanding of the inner workings of a Prolog evaluator in the context of Lisp. Lisp was found to be an extremely flexible implementation language. It offers built-in management of identifiers, rich tools for the construction of arbitrary data structures, and the automatic recycling of discarded pieces of data. The Prolog system is expected to run on any other Standard Lisp system, and possibly on other Lisp systems as well, after some interfacing work.

satisfaction of disjunctive goals while a parallel relational join operation provides a framework to solve conjunctive subgoals. An overview of the techniques currently being implemented and their relationship to the architecture is presented.

DADO: A Tree-Structured Machine Architecture for Production Systems

Salvatore J. Stolfo, David E. Shaw

Department of Computer Science
Columbia University
New York, NY 10027, USA

DADO is a parallel tree-structured machine designed to provide significant performance improvements in the execution of large Production Systems. The DADO machine comprises a large (on the order of a hundred thousand) set of processing elements (PE's), each containing its own processor, a small amount (2K bytes, in the current design) of

and management analysis for telephone cable maintenance in a timely manner. Many design decisions faced during the construction of ACE were guided by recent successes in expert systems technology, most notably R1/XCON, the Digital Equipment Corporation Vax configuration program. The most significant departure from "standard" expert systems architectures is ACE's use of a conventional data base management system as its primary source of information. Its primary sources of knowledge are the expert users of the database system and primers on maintenance analysis strategies. The coupling of "knowledge-base" and "data-base" demonstrates in a forceful way the manner in which an expert system can significantly enhance the throughput and quality of data processing environments supporting business management. However further difficult problems must be solved before the expert system approach becomes a standard technique in the data processing industry.