EDITOR'S FOREWORD

At the first Logic Programming Workshop, held at Debrecen, Hungary, July 1980, it was decided to promote a Logic Programming Newsletter, to be issued at least twice a year. It was also decided that it would be edited in Lisbon, and Lufs Moniz Pereira was appointed editor. The aim of the newsletter is to improve communication and cooperation among the Logic Programming Community as well as to divulge its ideas and achievements to a wider audience.

The newsletter will be sent free of charge to those soliciting it from the editor. However, to keep the newsletter free of charge, avoid soliciting it from the editor if you have easy access to a library or someone receiving it.

How can you participate?

1. It would be useful if each group with an interest in logic programming chose one representative in charge of:
   - keeping the editor informed of local news
   - pressing the group for contributions
   - direct contact with the editor regarding any urgent matters.

   So, choose your representative and send his name, address and telephone number to the editor as soon as possible.

2. Here's a list of different possible types of contribution (perhaps you will think of others):
   - short communications on your work
   - reviews of other people's work
   - abstracts of reports: send a report
   - reports on conferences attended and visits to other groups
   - personal news (posts, changes of address, etc.)
   - exchange of posts and posts available
   - description of grant proposals and contracts
   - illustrative programs to be included in the Newsletter or in the next edition of «How to solve it with Prolog», a compilation of Prolog programs
   - description of your research and development aims, including your policy regarding the institutional setting
   - start a debate (eg. which syntax for Prolog?)
   - comparison of Logic Programming with other programming languages
   - implementation description and evaluation
   - what you would like to see in the newsletter
   - addresses of practitioners
   - listing of papers published in journals and conferences (send your list).

3. Contributions may be sent directly to the editor or through your representative. They should fit, with regular margins, on standard size A4 paper (this size). Write in English, on one side only, with the spacing not endangering legibility, either typewritten or printed on the computer.

   The editor thanks António Porto and Eugénio Oliveira for helping in producing this issue, and acknowledges the financial support of the «Junta Nacional de Investigação Científica e Tecnológica» through the «Centro de Informática da Universidade Nova de Lisboa».

   The next number will be out as soon as there are enough contributions. Please contribute generously.

   The Editor
   LUIS MONIZ PEREIRA
The goal of the proposed research is to develop methods for efficient implementation of logic programs. There are two areas we wish to investigate, both of which are continuations of research conducted by members of the UCI Dataflow Architecture group. One aspect of the proposed research involves development of a non-von Neumann architecture for parallel execution of logic programs, preliminary work in this area is reported by Conery [3]. The second area involves transformation of high level logic specifications into efficient Prolog and/or procedural language programs, and is based on work by Morris [7].

1. A Parallel Processor for Prolog

One possible configuration of a multiprocessor machine for logic programs is to have all processors work on the same goal list, i.e. replace DCEsystem-10 Prolog’s depth-first search with some parallel search method. This is the approach implied by Kowalski (Algorithm Logic + Control) [5], and by Nilsson [8], where Petri nets are mentioned as possible parallel control structures for production systems.

A major difficulty with this approach is possible conflicting assignments for variables that could be generated by unifications in different branches of the search tree. The problem is one of simultaneous access to common memory by processes executing in parallel. This problem is avoided in data-driven systems (c.f. Arvind, Gostelow, and Plouffe [1]).

The attached figure shows a tree, where each node contains a goal list, and descendants of a node are obtained via one unification-and-replacement step. What we propose is a data-driven machine that would process independent branches of the tree using independent processors. The data passed between processors would be goal lists; thus our machine is more properly called a goal-driven processor.

Among the issues to be resolved are:

- Strategies for creating the tree. The figure shows two possibilities: one very ambitious and wasteful of processors, and another that is close to Prolog’s search tree. Are there any other strategies? Which is best? What runtime information can be used to create more efficient search trees? (Work by Clark and McCabe [2] on ordering subgoals and co-routining is relevant here.)

- Methods for allocating processors to goal lists. When a processor has expanded a node, what does it do next? Can it work on a completely independent branch, or should it confine itself to one of the just-created goals? What happens when all of the processors have been allocated?

- Methods for communicating results back to the processor that originated search.

The proposed research program is a “top-down” approach, moving from hand simulations to programmed simulations of an abstract machine (on the order of the Irvine Dataflow base machine) to simulations of a physical multiprocessor machine. This last machine could very well have the same architecture as the Irvine Dataflow machine [4].
Thus state descriptions have a structure which we might call a "bi-clause" as of either of the forms ..-S or A..-S, where A is an atom assigning higher priority to '.' than to '+'.

 execution of several of its parts. This extension consists, on the one hand, in providing an explicit notation to distinguish between processes that run concurrently from processes that run sequentially, and, on the other hand, in introducing a construct for process synchronization. It is hoped that this extension will not betray the essential spirit of HCL programming. For example, we required from the outset that our system should possess equivalent operational and declarative semantics, as is the case with HCL and in fact one of its more distinctive features [3]. Unlike some other works that have appeared recently and which try to enlarge HCL programs so as to include some notion of concurrency [1, 2, 5, 6], our system seems to be the only one to be defined entirely in logical terms. It is not the purpose of this note, however, to compare our system with these other systems (provisionally) our system "state description logic" or SOL, for short. The operational semantics now follows in the usual way: a state may be a logical consequence of P without A or B being logically equivalent to clauses of the logic program is in fact selected in front (S). The next step will be to allow several atoms in front (S) to be resolved simultaneously. With this notion of direct derivation it is easy to define the operational semantics of this system as for HCL. The equivalence between the declarative and the operational semantics follows from a result by Hill [4]. We view a state description S=S 1 + ... + S n as a description of the state of a process, which consists in n processes running concurrently, and described in the present state by S 1 , ..., S n respectively. Each of these n processes may itself consist of several processes running concurrently, and so on, depending on the way S is structured in terms of '. ' and '+' . What is lacking is a mechanism to synchronize these processes.

 We enforce synchronization by organizing the (generalized) clauses into sets and requiring that, in a direct derivation from ..-S to ..-T, if some clause C is used then the remaining clauses belonging to the same set as C must also be used. Put in another way, let us suppose that A 1 S 1 , ..., A n S n is one of the distinguished sets of clauses. This set may be represented by the expression

A 1 , ..., A n S 1 , ..., S n

called an implication, and we assume that A 1 , ..., A n S 1 , ..., S n is the same implication if 1, ..., n is a permutation of 1, ..., n. An SDL program is a finite nonempty set of implications. The definition of a direct derivation from ..-S to ..-T is the same as before with the sole difference that we now use implications instead of (generalized) clauses. The operational semantics now follows in the usual way: a state description S is a logical consequence of a given SDL program if ..-S derives ..-C. (The variables of S are assumed to be existentially quantified.) Notice that in a SDL program P a state description A+B may be a logical consequence of P without A or B being logical consequences of P (consider for example the program A,B+C,D). Thus the semantics of SDL programs must be described in terms of state descriptions and not of atoms, as is the case for HCL programs. Another new situation we must consider is the possibility of some processes being infinite yet meaningful. In this case we shift our attention from the notion of logical consequence to the notion of derivability (of which, as a matter of fact, the notion of logical consequence is a particular case).

As an example of an SDL program (taken from [10]) let us consider a resource and two objects 'a' and 'b', each using the resource in some phase of its activity, with the restriction that the objects cannot use the resource simultaneously. The initial state description and the program follow:

object(a,ownactivity)+resource(available)+object(b,ownactivity)
object(X,ownactivity) ← object(X,requests)
object(X,requests), resource(available) ← object(X,uses), resource(taken)

A NEW PROPOSAL FOR CONCURRENT PROGRAMMING IN LOGIC
Luis Monteiro
Departamento de Informática
Universidade Nova de Lisboa
1899 Lisboa Codex
PORTUGAL

We outline in this note a proposal for an extension to Horn clause predicate logic (HCL, for short) such that, when sets of clauses are interpreted as programs as originally suggested by Kowalski [7, 8], the entire execution of a given program may be viewed as the concurrent execution of several of its parts. This extension consists, on the one hand, in providing an explicit notation to distinguish between processes that run concurrently from processes that run sequentially, and, on the other hand, in introducing a construct for process synchronization. It is hoped that this extension will not betray the essential spirit of HCL programming. For example, we required from the outset that our system should possess equivalent operational and declarative semantics, as is the case with HCL and in fact one of its more distinctive features [3]. Unlike some other works that have appeared recently and which try to enlarge HCL programs so as to include some notion of concurrency [1, 2, 5, 6], our system seems to be the only one to be defined entirely in logical terms. It is not the purpose of this note, however, to compare our system with these other systems (provisionally) our system "state description logic" or SOL, for short. The operational semantics now follows in the usual way: a state may be a logical consequence of P without A or B being logically equivalent to clauses of the logic program is in fact selected in front (S). The next step will be to allow several atoms in front (S) to be resolved simultaneously. With this notion of direct derivation it is easy to define the operational semantics of this system as for HCL. The equivalence between the declarative and the operational semantics follows from a result by Hill [4].

We view a state description S=S 1 + ... + S n as a description of the state of a process, which consists in n processes running concurrently, and described in the present state by S 1 , ..., S n respectively. Each of these n processes may itself consist of several processes running concurrently, and so on, depending on the way S is structured in terms of '. ' and '+' . What is lacking is a mechanism to synchronize these processes.

We enforce synchronization by organizing the (generalized) clauses into sets and requiring that, in a direct derivation from ..-S to ..-T, if some clause C is used then the remaining clauses belonging to the same set as C must also be used. Put in another way, let us suppose that A 1 S 1 , ..., A n S n is one of the distinguished sets of clauses. This set may be represented by the expression

A 1 , ..., A n S 1 , ..., S n

called an implication, and we assume that A 1 , ..., A n S 1 , ..., S n is the same implication if 1, ..., n is a permutation of 1, ..., n. An SDL program is a finite nonempty set of implications. The definition of a direct derivation from ..-S to ..-T is the same as before with the sole difference that we now use implications instead of (generalized) clauses. The operational semantics now follows in the usual way: a state description S is a logical consequence of a given SDL program if ..-S derives ..-C. (The variables of S are assumed to be existentially quantified.) Notice that in a SDL program P a state description A+B may be a logical consequence of P without A or B being logical consequences of P (consider for example the program A,B+C,D). Thus the semantics of SDL programs must be described in terms of state descriptions and not of atoms, as is the case for HCL programs. Another new situation we must consider is the possibility of some processes being infinite yet meaningful. In this case we shift our attention from the notion of logical consequence to the notion of derivability (of which, as a matter of fact, the notion of logical consequence is a particular case).

As an example of an SDL program (taken from [10]) let us consider a resource and two objects 'a' and 'b', each using the resource in some phase of its activity, with the restriction that the objects cannot use the resource simultaneously. The initial state description and the program follow:

object(a,ownactivity)+resource(available)+object(b,ownactivity)
object(X,ownactivity) ← object(X,requests)
object(X,requests), resource(available) ← object(X,uses), resource(taken)

must be completed before the process with origin T starts, and 'S+T' means concurrent execution of the processes with origins S and T.

Let us now be more specific about the operational semantics. We shall write a generic state description S in the form S 1 + ... + S n , n ≥ 0, where no S i is either □ or □+ □ for some state descriptions S and □; we assume that S = □ if n = 0. In this case every S i has the form S 1 , ..., S i where no S j is either □ or □+ □. If each S i is an atom A i

we shall say the state description S=A 1 + ... + A n is flat.

We associate with each state description S a flat state description front (S) defined recursively as follows:

• front (S) = S if S is either □ or an atom;
• front (S.T) = front (S) and front (S+T) = front(S)+front(T)

if S and T are state description distinct from □.

We are now in a position to state precisely the notion of "direct derivation". As a first approximation, the notion that ..-S derives ..-T, where S and T are state descriptions, is precisely as in HCL [7, 8], with the sole difference that the atom selected in S for resolving with a clause of the logic program is in fact selected in front (S). The next step will be to allow several atoms in front (S) to be resolved simultaneously. With this notion of direct derivation it is easy to define the operational semantics of this system as for HCL. The equivalence between the declarative and the operational semantics follows from a result by Hill [4]. We view a state descriptions S=S 1 + ... + S n as a description of the state of a process, which consists in n processes running concurrently, and described in the present state by S 1 , ..., S n respectively. Each of these n processes may itself consist of several processes running concurrently, and so on, depending on the way S is structured in terms of '. ' and '+' . What is lacking is a mechanism to synchronize these processes.

We enforce synchronization by organizing the (generalized) clauses into sets and requiring that, in a direct derivation from ..-S to ..-T, if some clause C is used then the remaining clauses belonging to the same set as C must also be used. Put in another way, let us suppose that A 1 S 1 , ..., A n S n is one of the distinguished sets of clauses. This set may be represented by the expression

A 1 , ..., A n S 1 , ..., S n

called an implication, and we assume that A 1 , ..., A n S 1 , ..., S n is the same implication if 1, ..., n is a permutation of 1, ..., n. An SDL program is a finite nonempty set of implications. The definition of a direct derivation from ..-S to ..-T is the same as before with the sole difference that we now use implications instead of (generalized) clauses. The operational semantics now follows in the usual way: a state description S is a logical consequence of a given SDL program if ..-S derives ..-C. (The variables of S are assumed to be existentially quantified.) Notice that in a SDL program P a state description A+B may be a logical consequence of P without A or B being logical consequences of P (consider for example the program A,B+C,D). Thus the semantics of SDL programs must be described in terms of state descriptions and not of atoms, as is the case for HCL programs. Another new situation we must consider is the possibility of some processes being infinite yet meaningful. In this case we shift our attention from the notion of logical consequence to the notion of derivability (of which, as a matter of fact, the notion of logical consequence is a particular case).

As an example of an SDL program (taken from [10]) let us consider a resource and two objects 'a' and 'b', each using the resource in some phase of its activity, with the restriction that the objects cannot use the resource simultaneously. The initial state description and the program follow:

object(a,ownactivity)+resource(available)+object(b,ownactivity)
object(X,ownactivity) ← object(X,requests)
object(X,requests), resource(available) ← object(X,uses), resource(taken)
A "valuation" is as usual a function mapping variables onto elements of "models", "Herbrand models" of the equivalence between the operational and the declarative semantics of A. It is then possible to define the usual notions of "satisfaction" of all predicates over A. This function must satisfy some requirements, and of a state description being a "semantic consequence" of the SOL symbols occurring in s. Now an interpretation of the SDL program is any semantics, and the presentation of an experimental interpreter for this system written in Prolog. It can also be proved that the operational and the declarative semantics are equivalent.

The work outlined herein will be the subject of a series of papers dealing systematically with a general presentation of the system, the proof of the equivalence between the operational and the declarative semantics, and the presentation of an experimental interpreter for this system written in Prolog.

The loop. By contrast, in a recursively defined loop, both entrances and exits are at the top. For recursively defined functions, the functional value is available to pass at least one result back up the loop. In a relational language such as PROLOG, however, additional arguments are required to pass back results. Thus the PROLOG version of efficient reverse

\begin{verbatim}
reverse(X,Y) :- reverse([X|..Y],[],Y).
\end{verbatim}

requires only two. We argue that our mental representation of loops allows entrances and exits from both ends, and that the additional variables are an unnecessary obfuscation of this mental image.

We propose a new construct for PROLOG which will allow loops to be accessed from both ends, without the need for transporter variables. We do this by allowing PROLOG procedure calls to include what may be regarded as "temporary assertions" which are valid for the scope of the procedure call. We will call these relative assertions. A call P with relative assertion Q is denoted P/Q (some versions of PROLOG use the symbol "//" to denote what is called "cut" in DEC-10 PROLOG. We apologize for any confusion this may cause; the slash symbol seemed irresistably suited for conveying the notion of relative assertion) and may be read as "P given Q" or "P is deducible from Q." The expression may perhaps be assigned a declarative meaning as 0 implies P, although, properly speaking, it is a condition in the metatheory. Observe that the procedural interpretation here is quite different from that of the "implies" in

\begin{verbatim}
subset(X,Y) :- (member(Z,X) implies member(Z,Y)).
\end{verbatim}

A definition of "//" for current interpreted DEC-10 PROLOG follows, together with its use in defining the reverse and concat predicates. By separating out the termination conditions from the loops, we potentially increase the reusability of the loop definitions. Notice that with this definition, recursive calls on slash will stack the relative assertions, i.e. they are local to the particular invocation of slash. Thus relative assertions could be used to maintain an environment across several levels of procedure call, giving the equivalent of dynamic scoping.

The definition below cannot be used in compiled code since compiled languages are not accessible as data structures. We hope, however, that it will prove useful and that implementers will thereby be prompted to provide it at a more basic and efficient level of their system.

Definition and examples:

\begin{verbatim}
P/P :- 1.
true/A :- 1.
(P,Q)/A :- I, P/A, Q/A.
P/A :- clause(P,Q), Q/A.
c([X|..Y],[X|..Z]) :- c(Y,Z).
concat(U,V,W) :- c(U,W)/c([],Y).
r([X|..Y],Z) :- r(Y,[X|..Z]).
revv(X,Y) :- r([Y|Y],Y).
\end{verbatim}
A PROLOG PROGRAM FOR THE 'S-P PROBLEM'

by António Porto

Departamento de Informática
Universidade Nova de Lisboa
1899 LISBOA Codex

Reading through issue 37 of AISB Quarterly my attention was drawn to the 'S-P problem', which I had already seen stated in the well-known Martin Gardner's section on Mathematical Games in Scientific American. This time, however, there was along with the problem a challenge, attributed to John McCarthy, to write an 'AI-flavoured' program that could solve it.

The problem follows:

There are two numbers, M and N, such that 1 < M < N < 100. Mister S is told the sum of the two numbers, Mister P is told their product, and they both know they were told so. The following dialogue takes place:

Mr. P: I don't know the numbers.
Mr. S: I knew you didn't know them; I don't know them either.
Mr. P: Now I know the numbers!
Mr. S: Now I know them too!

What are the numbers?

That issue of AISB Quarterly also contained what was then considered to be the best reply, so far, to the challenge: it was a Prolog (what else?) program written by Martin Nilsson and John Campbell from the University of Exeter.

I decided to write my own Prolog program for this proplem. It has (I hope) a very clear reading, and uses as building blocks for the formulation of the 'S-P problem' the general subproblems of finding if a given problem has one and only one solution, if it has more than one solution, and if every one of its solutions entails a solution to another problem; these subproblems are efficiently defined.

We can imagine many different problems of the same kind, just by changing the dialogue. Corresponding programs, using my approach, would only differ in the top-level formulation of each particular problem, which just reflects the dialogue taking place.

The whole program, as written for the DEC-10 Prolog compiler, is now presented: (by the way, the solution is M=4 and N=13.)

/* DEFINITION OF INFIX OPERATORS */
:- op(800,xfy,[one_and_only_one, allows, verifying, every]).
:- op(800,xfy,[implies, more_than_one, have]).
:- op(750,xf, :).
:- op(700,xfx,given).
/* COMPILER DIRECTIVES */
:- public sp/2, one_and_only_one/2, verifying/2,
    every/2, more_than_one/2, given/2.
:- mode sp(-,-),one_and_only_one(+,+),verifying(+,+),more_than_one(+,+),
    every(+,+),no_record(+),update_record(+),not(+),the_sum_is(-),
    given(+,+),have(+,+),integer(-,+,+),factor(+,+,-).
/* THE PROBLEM */
sp(M,N) :- the_sum_is(S),
    sentence4 :-
    one_and_only_one product(P) given sum(S)
    allows sentence3 verifying Sx=S :
one_and_only_one sum(Sx) given product(P)
allows sentence2 :
   every product(Px) given sum(Sx)
   implies sentence1 :
      more_than_one sum(_) given product(Px),
the_numbers(M,N) have sum_and_product(S,P).

   The following subproblems make use of the DEC-10 Prolog
recording mechanism, which works as follows:

   'recorda(K,T,R)' records term T under key K,
   using reference R.

   'recorded(K,T,R)' accesses term T recorded under key K,
   its reference being R.

   'erase(R)' erases the term recorded with reference R.

S : one_and_only_one X allows Y :- call((X,Y)), recorded(S,_,R),
erase(R), !, fail ;
recorda(S,X allows Y,_,)
fail ) ;
recorded(S,X allows Y,R), erase(R).
S verifying P : one_and_only_one X allows Y :-
call((X,Y)),( not(P),
   no_record(S), !, fail ;
   recorded(S,_,R),
erase(R), !, fail ;
   recorded(S,X allows Y,_,),
   fail ) ;
recorded(S,X allows Y,R), erase(R).
S : more_than_one X :- call(X), recorded(S,-,R), erase(R), ! ;
recorded(S,1,_,), fail ) ;
recorded(S,_,R), erase(R), fail.
S : every X implies Y :- call(X),( not(Y),
   no_record(S), !, fail ;
   update_record(S), fail ) ;
recorded(S,N,R), erase(R), N=2.

no_record(S) :- recorded(S,_,R), erase(R) ;
   true.
update_record(S) :- ( recorded(S,2,_) ;
   recorded(S,1,_,), erase(R), recorded(S,2,_) ;
   recorded(S,1,_) ), !.
not(X) :- call(X), !, fail ;
   true.
the_sum_is(S) :- integer(S,4,198).
the_numbers(M,N) have sum_and_product(S,P) :- integer(M,2,99),
   N is S-M,
P is M*N, !.
product(P) given sum(S) :- S2 is S/2, integer(M,2,S2), P is M*(S-M).
sum(S) given product(P) :- factor(P,2,M), S is M+(P/M).
integer(I,_,_) :- !.
integer(I,Low,Up) :- New_low is Low+1, New_low =< Up,
   integer(I,New_low,Up).
factor(P,M,M) :- O is P mod M.
factor(P,G,M) :- New_guess is G+1, P =< New_guess*New_guess,
   factor(P,New_guess,M).
LOGIC PROGRAMMING WORKSHOP USA

A Logic Programming Workshop was recently organized by Syracuse University, from 8-10 April, and held at Thorntfield, an amenable location on Lake Cazenovia, 30 minutes from Syracuse.

Most of the 60 odd people or so attending were Americans and Canadians, plus a sprinkler of European "veterans". Logic programming, mainly through the use of Prolog, is picking up speed in the States, and is also very much à la mode.

There were 5, 20 and 30 minute informal presentations, grouped into sections. These were Program Design, Control (compile and runtime), Metalanguage, Data Bases, Implementation, Parallelism, Natural Language, and a General Section (applications, extensions, etc.).

Participation was lively, the surroundings beautiful, and the atmosphere friendly.

MICRO-PROLOG 2.0

Micro-Prolog is an interpreter for a subset of Prolog for micro-computers. Its runs on the Z80 processor, under CP/M operating system, in 32 K bytes of memory. The interpreter itself is 8.5 K bytes (written in Z80 assembler). Speed is approximately 120 resolutions/sec on a 2 MHz Z80 with no wait states. It is available on North Star, Heath/Zenith and Z80 processor, using the CP/M operating system.

It is running on North Star Horizon, Research Machines 3802 and Heathkit Zenith disk-based systems. It is hoped that an effective means of distributing materials, either through government agency or educational publishers, will soon be formulated, with software handled, as at present, through Logic Programming Associates.

Language definition. C. J. Daniel Moss is just finishing a Ph. D. thesis on the definition of programming languages. Logic is providing a unified approach to both syntax and semantics. Metamorphosis grammars (also called definite clause grammars) allow one to describe the context sensitive portions of a programming language with unrivalled clarity. The dual semantics of Prolog provides a "denotational" and an operational semantics for programs, and they can also express the axiomatic approach.

Program transformation. The work on transformation of algorithms is continuing and the classification of algorithms via synthesis has been extended by Keith Clark, Derek Brough and Phil Vasey.

Loop trapping. The previous investigations by Derek Brough into loop trapping for logic programs has been applied to micro-Prolog to simplify its use in the schools project.

"LOGIC PROGRAMMING CONFERENCE"  

The next Logic Programming event will take place at Luminy, an altogether beautiful campus just outside Marseille, France, in the middle of September 1982. The exact date and arrangements are not yet known, but it is to be a major event for the logic programming community.

"HOW TO SOLVE IT WITH PROLOG"

Copies of the August 80 2nd edition of this compilation of Prolog programs may be obtained by writing to:  

Hélder Coelho  
Laboratório Nacional de Engenharia Civil  
Av. do Brasil  
1799 Lisboa Codex  

"PROLOG BIBLIOGRAPHY"

An updated listing of publications on Prolog may also be obtained from Hélder Coelho, at the address above.

RESEARCH GRANTS AT MARYLAND

Professor Jack Minker informed us that he holds two grants that may be of interest:

(a) Investigations of a predicate logic language for problem solving (NASA).
(b) Investigations of the use of predicate logic in deductive database systems (NSF).

WORK IN PROGRESS AT THE K. U. LEUVEN/ BELGIUM

In 1979, we have developed a portable Prolog interpreter written in Pascal (space efficient, i.e. tail recursion optimisation).

Last year, we have developed a new interpreter in the language C for the UNIX operating system. In this new interpreter, all names of constants are placed on a file, this results in more working space, but also in some overhead for input/output. Recently, we have connected this interpreter to a simple relational database. When the interpreter accesses this database, it asks for all tuples matching a certain pattern, it pushes all these tuples on a stack. Then it consumes these tuples one by one (the normal backtracking mechanism).

For more information, write to:

Maurice Bruynooghe  
K. U. LEUVEN  
Afdeling Toegepaste Wiskunde en Programmatie  
Celestijnenlaan 200 B  
B-3030 Heverlee/Belgium
I was invited to VLDB to participate in a panel on the relationship between database theory and practice. Predictably, my contribution concentrated on the role that logic programming can play in the database field. The reception to my talk was encouraging as was the general attitude towards logic and databases. I got the impression that the database research community has had enough of relational database theory (though not with implementation) and is now ready to look further afield in such directions as logic and artificial intelligence.

A separate panel was organised to report on a workshop held in Colorado earlier this year specifically to explore relationships between databases, abstract data structures and artificial intelligence. There were several papers presented at the VLDB conference which used logic for query optimisation and integrity checking. I think there will be a lot more activity using logic in these two areas.

The next VLDB will be held in Cannes from 9-11 September 1981. The call for papers explicitly included the following topics:

- Database and Logic
- Natural Languages
- Artificial Intelligence
- Programming Language and Databases
- Knowledge Based Systems

The deadline for submission of papers was 1 March 1981.

**IIUW-Prolog**

Feliks Kluzniak  
Institute of Informatics  
Warsaw University  
P.O.B. 1210  
00-901 Warszawa, Poland

IIUW-Prolog is a Prolog interpreter written in 1979 at the Institute of Informatics, Warsaw University. The program is written entirely in standard Pascal and the language it supports is very similar to a Marseilles version described in Ph. Roussel’s 1975 manual.

While the data structures were designed from the scratch to take full advantage of Pascal’s type-definition and data-packing features, their basic philosophy is also that of Battani’s and Melon’s 1973 Marseilles interpreter. Note, however, that no bootstrapping is employed (*) — the resulting program—reading speed, accompanied by exhaustive diagnostics, makes IIUW-prolog especially suitable for novice Prolog programmers.

The interpreter performs particularly well on the CDC 6000 series computers, thanks to the quality of the ETH Pascal compiler and its ability to get the most out of data-structure packing. We measured the cost — in CPU time — of solving standard Warplan problems on a CYBER 73: the results were comparable to those published by Warren for the Marseilles interpreter on an IBM 360/67. In this test the cost of data packing offsets the difference in processor speed, but better memory utilisation makes it worthwhile. We have always found 72000 (octal) words satisfactory for large programs, and most student jobs can run within the standard limits set for Fortran “quickies”. Thus, the performance of IIUW-Prolog seems significantly better than that of the standard Lisp interpreter for this range of machines.

The implementation techniques used in IIUW-Prolog are slightly obsolete, but it was not our aim to advance the state of the art. All we needed was a cheap but reasonably efficient interpreter which we could use to teach Prolog in an academic community that is rather hard-pressed for computer time. The cost of the whole effort, from the initial design to an almost bug-free version, was very low indeed. The present author did it single-handedly in two months, of which a week was spent punching cards and three weeks were used for debugging on a batch system with 3-4 runs a day. (I admit, through, that I put a lot of overtime and did not write the documentation).

After using the interpreter for one year we found it worth-while to add a number of new standard procedures (i.e. evaluable predicates), tracing facilities on so on. By mid-’81 this effort will be over and IIUW-Prolog will probably take over as the standard big-machine implementation in Poland. We will also gladly supply it to anyone who will be interested enough to send us a tape.

(*) While metamorphosis grammars are processed by brute force, the problem of Prolog’s variable syntax was rather satisfactorily solved by applying the well-known (though rarely used) algorithm originally designed for the translation of expressions to Reverse Polish notation.

**THE PROLOG TEAM AT WARSAW UNIVERSITY**

Prolog is not yet widely known in Poland, though research workers at several universities have shown interest in the language, and some are even using it. Our group at Warsaw University, however, is still the only one that is actively concerned with logic programming as such. The group now consists of a core of three persons (Janusz S. Bien, Ph. D., Feliks Kluzniak, M. Sc., Stanislaw Szpakowicz, Ph. D.) and a few cooperating students and programmers. We are mainly interested in implementing Prolog, teaching Prolog and Prolog applications, especially in the field of computational linguistics. Our activities in these three areas are summarized below.

There are at present two Prolog implementations being used in Poland. The first, officially distributed one is a variation of the original Marseilles interpreter; it was cleaned out and extended (e.g. by adding tracing facilities), and adapted for the ICL 1900-compatible Polish ODRA 1305 computer. The other version is a new interpreter written in Pascal for the CDC CYBER (see a note on IIUW-Prolog in this newsletter). We are now in the process of porting it to an R-32 (the local version of an IBM 360).

Other implementation efforts are under way. One of these is a Warren-like compiler for the CDC CYBER, which is being written by M. Lazinski for his M. Sc. degree.

A regular Prolog course is being taught to computer science students since 1979 (we presented at the Debrecen Workshop a short report on our teaching method). We also do our best to spread knowledge of Prolog outside Warsaw University; an important result in this field is a textbook on Prolog for professional programmers, research workers in computer science and students. The textbook has now been submitted to a Warsaw publisher (Wydawnictwa Naukowo-Techniczne).

Two rather large application programs are worth mentioning. One is a surface-syntactic analyser of written Polish (by Szpakowicz); the other is an experimental interactive railway timetable information system with natural language interface (by Szpakowicz and Marek Swidzinski).


Stanislaw Szpakowicz
FORTHCOMING BOOKS

"Programming in Prolog", by William F. Clocksin and Christopher S. Mellish, from the Dept. of Artificial Intelligence at Edinburgh, is the long waited for primer on Prolog, by experienced practitioners. It is to be published by Edinburgh University Press this year. Here are some extracts from its Preface:

Until now, there has been no textbook with the aim of teaching Prolog as a practical programming language. It is perhaps a tribute to Prolog that so many people have been motivated to learn it by referring to the necessarily concise reference manuals, a few published papers, and by the orally transmitted folkore of the modern computing community. However, as Prolog is beginning to be introduced to large numbers of undergraduate and postgraduate students, many of our colleagues have expressed a great need for a tutorial guide to learning Prolog. We hope this little book will go some way towards meeting this need.

This book was designed to be read sequentially. Each chapter is divided into several sections, and we advise the reader to attempt the exercises that are at the end of most sections. The solutions to many of the exercises appear at the end of the book. Chapter 1 is a tutorial introduction that is intended to give the reader a "feel" for what is required to program in Prolog. The fundamental ideas of Prolog are introduced, and the reader is advised to study them carefully. Chapter 2 presents a more complete discussion of points that are introduced in Chapter 1. Chapter 3 deals with data structures and derives some small example programs. Chapter 4 treats the subject of backtracking in more detail, and introduces the "cut" symbol, which is used to control backtracking. Chapter 5 introduces the facilities that are available for input and output. Chapter 6 describes each built-in predicate in the standard "core" of Prolog. Chapter 7 is a potpourri of example programs collected from many sources, together with an explanation of how they are written. Chapter 8 offers some advice of debugging Prolog programs, and provides an alternative model of control flow. Chapter 9 introduces the Grammar Rule syntax, and examines the design decisions for some aspects of analysing natural language by using Grammar Rules. Chapter 10 describes the relation of Prolog to its origins in mathematical theorem proving and logic programming. Chapter 11 specifies a number of projects on which interested readers may wish to practise their programming ability.

"LOGIC PROGRAMMING WORKSHOP 1"

A book coming out of the workshop held last July at Debrecen, Hungary, is being edited by Keith Clark and Sten-Åke Tarnlund. The book contains re-written versions of many of the papers presented at Debrecen, as well as some entirely new contributions. All were refereed by the present Logic Programming Workshop 2 Committee.

"MATHEMATICAL LOGIC PROGRAMMING COLLOQUIUM 78"

The proceedings of this Colloquium, held at Salgotarjan, Hungary, in September 1978, are now under print by North-Holland, and should be out before next summer.
NEW PAPERS

from Syracuse:

Programming with Full First Order Logic

Kenneth A. Bowen

An automatic deduction system based on a modification of Gentzen’s sequent system LJ is presented and its use as the basis for a logic programming system is described. The system is a natural extension of Horn clause logic programming systems in that when all of the formulas in the input sequent are atomic, the behavior of the system mimics that of LUSH resolution systems. The use of such systems in program development systems and in database management systems is discussed.

Loglisp — An Alternative to Prolog

J. A. Robinson  E. E. Sibert

(No abstract provided)

Logic Programming in Lisp

J. A. Robinson  E. E. Sibert

This document describes version V1M1 of LOGLISP, an extension of LISP in which one can do logic programming [Kowalski 1974, 1979]. The logic programming system within LOGLISP is called LOGIC. Thus we have: LOGLISP = LOGIC + LISP.

LOGIC differs in a number of ways from the well-known PROLOG implementations of logic programming [Roussel 1975], [Warren 1977], [Roberts 1977], [Clark 1979]. The most noteworthy difference is that LOGIC is simply a set of new LISP primitives designed to be used freely within LISP programs. These primitives are invoked in the ordinary LISP manner by function calls from the terminal or from within other LISP programs. They return their results as LISP data objects which can be subjected to analysis and manipulation. Each of the logical procedures comprising a LOGIC knowledge base is a LISP data object (like the definition of an ordinary LISP procedure) on the property list of the identifier which is its name.

Thus one calls LOGIC from within LISP. It is also possible to call LISP from within LOGIC. The identifiers used as logical predicate symbols, function symbols and individual constants within a knowledge base or query can be given a LISP meaning by the ordinary LISP method of definition or assignment. Some identifiers (CAR, CONS, PLUS, etc.) already have a LISP meaning imposed by the system. Thus every logic construct (term, or atomic sentence) is capable of being interpreted as a LISP construct. During the deduction cycle of LOGIC each logic construct is “evaluated” as a LISP construct, according to a suitably generalized notion of evaluation.

The effect of this LISP-simplification step within each deduction step is to make available to the LOGIC programmer virtually the full power of LISP. This makes trivially easy the “building-in” of “immediately evaluable” notions — but far more than that. In particular, LOGIC calls can be made from within LOGIC calls.

from Waterloo:

Predicate Logic as a Language for Parallel Programming

by M. H. van Emden, G. J. de Lucena * & H. de M. Silva

We describe the formulation, execution, semanticization, and verification within first-order predicate logic of programs in Kahn’s model of computation. The relations computed by process activations are defined in logic. The state of a network of communicating parallel processes is specified in a single statement of logic which is a concise textual representation of such a network. The state is understood to comprise the configuration of the network of process activations, the contents of the channels, as well as the state of each sequential computation within a process activation.

It is possible to derive within logic results from the process definitions and from the state specification in such a way that each stage of the derivation can again be interpreted as a state of a parallel computation and that the transitions between stages is also directly meaningful in terms of Kahn’s model of computation.

We show that dataflow programs in Lucid are closely related to our representation of these programs in logic. We give an example of partial verification of a terminating program. Finally, we sketch the application of recent results on greatest fixpoints and infinitary Herbrand universes to verification of nonterminating programs.

from Maryland:

On Optimizing the Evaluation of a Set of Expressions

John Grant and Jack Minker

A branch-and-bound type algorithm is developed to optimize the evaluation of a set of expressions. This algorithm proceeds in a depth-first manner and achieves an optimal solution. The algorithm is applied to optimize the evaluation of sets of relational expressions. Analogies to the heuristic information associated with the A*-algorithm are investigated. Examples are presented illustrating the use of the algorithm. Fragmatics associated with the algorithm are discussed. Using the same framework, we present a new method to optimize the evaluation of Boolean expressions.

Optimization in Deductive and Conventional Relational Database Systems

John Grant and Jack Minker

A deductive relational database system is one which permits new relations to be derived from given relations stored in a conventional relational database system, and from axioms. It has been shown that a query in a deductive relational database system can be transformed, using the axioms, into a query that involves searches only over the relational database. The transformed query results in a set of conjuncts which generally share similar or not identical searches that must be made of the indexes and the tables storing the relations. The purpose of this paper is to describe a “global” optimizing algorithm which accounts for similarities between conjuncts.
The algorithm consists of two major parts: the preprocessor and the optimizer. The preprocessor is used once for a given set of axioms and indexes. Its functions are to: transform each atomic query type into a group of formulas, list all possible access methods for single tables and join-supported joins and to calculate costs for the access methods. The optimizer is used to select a method of evaluation of the formulae which answers the query in the shortest possible time. Details concerning the preprocessor and the optimizer are provided. An example is given that shows the effectiveness of "global" optimization in contrast to optimizing the retrieval of individual conjuncts. The changes needed to incorporate semantic knowledge into the algorithm are also given.

A Set-Oriented Predicate Logic Programming Language

Jack Minker

A predicate logic language based on types and set operations is presented. The use of set operations is shown to alleviate some problems associated with backtracking in nondeterministic systems.

The basic syntax and semantics of the typed-set-oriented language is specified. It is shown that types and set operations may be embedded as part of the unification algorithm. This permits a uniform way to handle types and set operations, and permits dynamic type checking.

The inference mechanism and bookkeeping features employed with the system provide debugging features for the user. A discussion of some of the features in the language is presented. We discuss both the limitations of some of the features, and how a number of features described may be incorporated within a language such as PROLOG by modification to the control structure.

We report on an experiment in interactive reasoning with FOL. The subject of the reasoning is elementary algebra. The main point of the paper is to show how the use of meta-theoretic knowledge results in improving the quality of the resulting proofs in that in this environment they are both easier to find and easier to understand.

Evaluating Functions Defined in First Order Logic

by L. Aiello

After a short introduction to FOL, an interactive reasoning system for first order logic, we present a way of extending the use of the FOL evaluator by showing how systems of (mutually recursive) function definitions formulated in first order logic can be translated into programs. This allows function definitions (syntactic objects) to be treated as programs (semantic objects). The advantages of this translation are illustrated.

The Call-by-Name Semantics of a Clause Language with Functions

by M. Bellia, P. Degano and G. Levi

The paper presents a language which extends TEL, a functional language, with (somewhat constrained) Horn clauses. The resulting language provides some features which are characteristic of relational languages, such as procedures with more than one output. We have defined an operational semantics, based on a lazy evaluation rule. Finally, a call-by-name semantics is given, which is an extension of the tarskian model theoretical semantics.

From term Rewriting Systems to Distributed Programs Specifications

by M. Bellia, E. Dameri, P. Degano, G. Levi, M. Martelli

The paper presents a formal model for distributed systems of computing agents, which is based on extended term rewriting systems. An operational semantics is given, which neatly mirrors both the non-deterministic and the parallel features of systems of computing agents. The formalism we introduce has an immediate interpretation in terms of first order logic. Thus, we provide it with a fixed-point semantics, closely related to the model theoretic semantics of first order theories.

from Marseille:

Dialogues en Français avec un Ordinateur

Paul Sabatier

( now at Universidade Nova de Lisboa)

We present a complete system of natural language communication with a computer, in which a casual user can describe and modify an open world dealing with persons and objects (any proper noun). Persons give, lend, exchange objects they possess and/or they hold. The user can ask any question about the current relations of possession between the different persons and objects. Written in PROLOG, this system provides a logic for actions, questions, presuppositions and ellipsis sentences.

PROLOG: A Language for Implementing Expert Systems

K. L. Clark, F. G. McCabe

Dept. of Computing

imperial College, London SW7 ZBZ

November 1980

ABSTRACT

We briefly describe the logic programming language PROLOG concentrating on those aspects of the language that make it suitable for implementing expert systems. We show how features of expert systems such as:

1) inference generated ed requests for data,
2) probabilistic reasoning,
3) explanation of behaviour

can be easily programmed in PROLOG. We illustrate each of these features by showing how a fault finder expert could be programmed in PROLOG.
<table>
<thead>
<tr>
<th>Research Centre</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dept. de Informática</td>
<td>Universidade Nova de Lisboa</td>
</tr>
<tr>
<td></td>
<td>Quinta do Cabeço</td>
</tr>
<tr>
<td></td>
<td>1899 Lisboa Codex Portugal</td>
</tr>
<tr>
<td>Dept. of Artificial Intelligence</td>
<td>University of Edinburgh</td>
</tr>
<tr>
<td></td>
<td>9 Hope Park Sq.</td>
</tr>
<tr>
<td></td>
<td>Edinburgh EH8 9NW UK</td>
</tr>
<tr>
<td>Dept. of Computing and Control</td>
<td>Imperial College</td>
</tr>
<tr>
<td></td>
<td>180 Queen’s Gate</td>
</tr>
<tr>
<td></td>
<td>London SW7 2BZ UK</td>
</tr>
<tr>
<td>Groupe d'Intelligence Artificielle</td>
<td>Universite d'Aix-Marseille II</td>
</tr>
<tr>
<td></td>
<td>70 route Leon Lachamps Case 901</td>
</tr>
<tr>
<td></td>
<td>13288 Marseille Cedex 2 France</td>
</tr>
<tr>
<td>Afdeling Toegepaste</td>
<td>Wiskunde en Programmatie</td>
</tr>
<tr>
<td></td>
<td>Katholieke universiteit Leuven</td>
</tr>
<tr>
<td></td>
<td>Celestijnenlaan 200 A</td>
</tr>
<tr>
<td></td>
<td>B—3030 Haverlee Belgie</td>
</tr>
<tr>
<td>Dept. of Information Processing</td>
<td>and Computer Science</td>
</tr>
<tr>
<td></td>
<td>Stockholms Universitet</td>
</tr>
<tr>
<td></td>
<td>S-106 91 Stockholm</td>
</tr>
<tr>
<td></td>
<td>Sweden</td>
</tr>
<tr>
<td>Ist. di Scienze dell' Informazione</td>
<td>Universita di Pisa</td>
</tr>
<tr>
<td></td>
<td>Corso Italia 40</td>
</tr>
<tr>
<td></td>
<td>I-56100 Pisa Italia</td>
</tr>
<tr>
<td>Institut fur Informatik</td>
<td>Universitat Karlsruhe</td>
</tr>
<tr>
<td></td>
<td>Postfach 6380</td>
</tr>
<tr>
<td></td>
<td>D-7500 Karlsruhe 1 Deutschland</td>
</tr>
<tr>
<td>SZKI</td>
<td>P.O.B. 224</td>
</tr>
<tr>
<td></td>
<td>Budapest H-1368 Hungary</td>
</tr>
<tr>
<td>Institute of Informatics</td>
<td>University of Watsaw</td>
</tr>
<tr>
<td></td>
<td>P.O.box 1210</td>
</tr>
<tr>
<td></td>
<td>00-901 Warszawa Poland</td>
</tr>
<tr>
<td>School of Computer and Information</td>
<td>Science</td>
</tr>
<tr>
<td></td>
<td>Syracuse University</td>
</tr>
<tr>
<td></td>
<td>Syracuse, New York 13210</td>
</tr>
<tr>
<td></td>
<td>USA</td>
</tr>
<tr>
<td>Dept. of Information and Computer</td>
<td>Science</td>
</tr>
<tr>
<td></td>
<td>University of California</td>
</tr>
<tr>
<td></td>
<td>Irvine, California 92717</td>
</tr>
<tr>
<td></td>
<td>USA</td>
</tr>
<tr>
<td>Dept. of Computer Science</td>
<td>University of Waterloo</td>
</tr>
<tr>
<td></td>
<td>Waterloo, Ontario</td>
</tr>
<tr>
<td></td>
<td>Canada</td>
</tr>
</tbody>
</table>