3

LOGIC PROGRAMMING NEWSLETTER

VERÃO DE 1982 SUMMER 1982

CONTENTS

EDITOR

Prof. 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

ACKNOWLEDGEMENT

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

COLABORATION

The editor thanks ANTÓNIO PORTO for his help.

CONTRIBUTIONS

Send to the editor, typed or printed, if possible double spaced, one side of the paper only and a maximum width of 15 cm.

DESIGN, TYPESETTING AND PRINTING

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

EDITOR'S FOREWORD

● *Next September, 14-17, takes place at the University of Marseilles the First International Logic Programming Conference, for which 35 papers have been accepted, from all over the world. Great expectation surrounds this event, where important contributions to the field will be presented. People interested in participating or in the proceedings should contact:*

Prof. ALAIN COLMERAUER
Groupe d'Intelligence Artificielle
Université d'Aix-Marseille II, case 901
route Léon Lachamp 70
13288 Marseille 9 cedex
France ☎ (91) 413 428

● *It has been this newsletters's policy to publish new numbers as soon as there are enough contributions. If you wish to receive it more often simply contribute more often! And please note the change of address. In consequence of the inevitable confusion during this change, many newsletters were sent by surface mail, taking some three months to reach California (an average of 3 km per day!). We regret the inconvenience and apoligise for the delay.*

● *A special issue devoted to the comparison of Prolog and other logic programming languages implementations, capabilities and performances, seems to be timely. The editor would like to receive suggestions on this matter.*

● *The newsletter has received another Government subsidy, which should guarantee another two numbers. This subsidy is attributed to scientific publications in the launching stage, but cannot become the regular support of all the expenses after that. To help ensure the uninterrupted continuation of this publication, we propose that, on a voluntary basis, all those who can contribute do so, by sending a least $6.00 US or equivalent, per number received, to the editor, payable to the Centro de Informática da Universidade Nova de Lisboa. This way we hope to continue sending the newsletter air mail to everyone, whether or not they have been able to contribute.*

PERPETUAL PROCESSES — AN UNEXPLOITED PROLOG TECHNIQUE

David H. D. Warren
Artificial Intelligence Center
SRI International, USA

In current Prolog programming practice, processes run for as short a time as possible, "output" the desired results, and then die. Terms created during program execution are normally regarded as "temporary" information. To keep some information "permanently", it is generally felt that it has to be stored as clauses. Thus one of Prolog's great attractions — its powerful handling of structured data — is dimmed somewhat by the fact that these structures exist only transiently.

However, recent improvements in Prolog implementation, namely tail recursion optimisation together with garbage collection, make possible a new style of Prolog programming in which processes can continue running indefinitely, building up large structures and maintaining them over an extended period of time. Given that the system provides a means of saving and restoring Prolog states, there is now no reason why terms should not be used for storing permanent information.

The purpose of this note is to stimulate people to try out the "perpetual processes" style of programming, and to take advantage of it in developing new applications. The technique seems to open up new possibilities for building programs which provide the user with completely self-contained environments. Examples would be systems for program development, and for text processing. It even appears plausible to build the equivalent of a conventional operating system using this technique, given a sufficiently large virtual memory.

To illustrate the use of "perpetual processes", and to help convince the reader that it is entirely practical, I will briefly describe a simple text editor I have implemented using the technique. Being only a line oriented editor, it is relatively pedestrian compared with the latest screen editors. Nevertheless, it has proved quite handy for editing Prolog source files from within Prolog.

The text of a file is stored as two lists of lists, representing the lines of the file above and below a certain point that is the focus of attention. The lines above the focus point are listed in reverse order. Each line is just a list of characters. Moving up and down the file is accomplished simply by transferring lines from one list to the other. Most editing commands that actually change the file affect only the line immediately above the focus point, ie. the first line in the "above" list.

Because the editor is writen in "pure" Prolog, all the editing operations actually result in the creation of new copies of the structure representing the file. However the way the file is represented, and the nature of the editing commands, mean that the amount of copying is generally quite small compared with the overall size of the file. The pure Prolog approach makes it potentially easy to "undo" changes, although in the present editor this feature is only used to a limited extent: commands which fail, such as a search for a line that does not exist, produce no effect.

The heart of the editor is a determinate, (indirectly) tail recursive procedure called 'edit', which repetitively displays the line immediately above the current focus point, gets the next editing command from the user, and then obeys it:

```
edit(Above,Below) :-
    show_last_line(Above),
    get_string(Command),
    obey(Command,Above,Below).
```

A typical editing command is:

```
>foo(__) :-
```

which means: search forward for a line beginning with "foo(" and ending with ") :-". The clause responsible for obeying this command is:

```
obey ([62|String],A,B) :- seek(String,A,B,A1,B1), !, edit(A1,B1).
```

where 62 is just the ASCII code for ">". The procedure 'seek' is defined by:

```
seek(S,A,B,A,B) :- last_line_matches(A,S), !.
seek(S,A,[L|B],A1,B1) :- seek(S,[L|A],B,A1,B1).
```

Because the 'edit' cycle is determinate, tail recursion optimisation ensures that the recursion stack does not grow in size, and that storage representing previous versions of the file is eventually reclaimed by garbage collection. Thus there is nothing to prevent the user from carrying on editing indefinitely.

With the editor running compiled on the DEC-10, most editing commands are performed apparently instantaneously, even commands which require searching through a substantial volume of text. About the only operation that is not instantaneous is reading the file into memory in the first place. This, plus the relatively small virtual memory on the DEC-10, limit the size of file that can comfortably be handled, but for a normal-sized Prolog source file (of between 1 and 5 pages, say) there is no problem.

The way text is represented in this editor suggests an interesting possibility for extension, which might be called a "document-oriented" editor. The text of the document would be represented as a tree structure, with the title and main headings at the top level, and with ordinary text and further subsections at deeper levels. With this kind of organisation, it would be easier to find one's way around the document without unnecessary searching, and operations to do text formatting could readily be included.

This note is an extended abstract of a talk presented at the Prolog Programming Environments Workshop, Linkoping, Sweden, March 1982.

A SMALL INTERPRETER FOR DISTRIBUTED LOGIC

Luís Monteiro

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

INTRODUCTION

This communication presents an interpreter for Distributed Logic [Mon 81a] [Mon 81b] [Mon 82] (DL for short), written in the Edinburgh Prolog version for the RT-11 [Clo 80]. Our main concern in writing this interpreter was "to see the theory work", so we sacrificed to a large extent the efficiency of the interpreter to its readability. In the sections that follow we introduce the language in which DL programs are written, make some comments on the interpreter, present its listing, and show two example programs.

THE LANGUAGE

(1) The interpreter is called 'DTLOG'. It is based on the most recent version of DL, referred to in the squel as 'DL.1' [Mon 82].

(2) DL.1 enriches HCL (Horn Clause Logic) with the following concepts:

- *(2.1) computing agents;*
- *(2.2) configurations;*
- *(2.3) transition rules;*
- *(2.4) transitions;*
- *(2.5) an extension of the resolution principle to deal with the above notions.*

We now comment on each of these notions in turn.

(3) (3.1) COMPUTING AGENTS: These are ordinary HCL terms.

(3.2) CONFIGURATIONS: Computing agents are organized into configurations, which are special terms. They are built up with computing agents and the operators 'Λ' (nullary) and '+', '.' (binary). These operators are called respectively the "null configuration" and the "concurrent" and "sequential" compositions; '+' and '.' are associative and have 'Λ' as neutral element. In writing DL programs for DTLOG we use 'skip', '<>' and '&' instead of 'Λ', '+' and '.' respectively.

(3.3) TRANSITION RULES: Expressions of the form

a1,...,an –> c1,...,cn

where n>0, the a's are agents and the c's are configurations. In DTLOG we restrict n to n=1 or n=2, and call the transition rules unary or binary respectively. They are written

a –> c

or

(a1,a1) ––> (c1,c2)

where a longer arrow is employed in binary transition rules merely to increase the efficiency of the interpreter. The logical status of a transition rule is that of an atomic formula. The user is expected to use them normally (but not necessarily) in the heads of clauses.

(3.4) TRANSITIONS: Expressions of the form

c => d

where c, d are configurations. We allow

c^

as an abbreviation for c=>skip. In writing c=>d, DTLOG assumes that d contains no occurrence of 'skip' if d =/= skip. The logical status of a transition is that of an atomic formula. The user is expected to use them normally (but not necessarily) in the bodies of clauses. The operational interpretation of c=>d is "execute c until it has the form d, applying suitable transition rules to c".

(3.5) RESOLUTION PRINCIPLE: DL programs are, like HCL programs, sets of Horn clauses. To deal with the extra notions, the resolution principle has been suitably extended. We are only interested in linear top-down refutation strategies. The resolution of a negative clause and a definite clause to yield a new negative clause is as in Prolog, with the exception of goals of the form c=>d, as explained below.

COMMENTS ON THE INTERPRETER

The interpreter has been written employing techniques developed mainly by L. Moniz Pereira [Per 82].

(1) When presented with a user program, DTLOG expects to solve (execute) goals of the form X=>Z, where X and Z are configurations. (A goal of the form X^ is transformed into X=>skip; any other goal is immediately transferred to the Prolog interpreter.) If X and Z are unifiable then the task terminates successfully. Otherwise it performs a transition step on X to obtain Y and tries to solve Y=>Z. The transition step is accomplished by the predicate step(X,Y).

(2) A transition step is subdivided in our stages:

(2.1) In the first stage, configuration X is split up into its "top" T and the "context" C in which T occurs in X (see example below). The context C, however, has distinct variables in the place of holes. As a first approach, T is the list of all agents which occur in the top of X. As a second approach, each element in the list T is in fact a pair, written A–>V, formed by an agent A occurring in the top of X and a variable V which ocupies in C the same position as ocupied by A in X. This stage is accomplished by the predicate top(X,T-[],C). (Note the use of difference lists, to avoid the explicit definition of the concatenation of two lists.) As an example, if we start with

X = (a<>b)&c <> d&e

then we obtain

T = [a–>U, b–>V, d–>W] C = (U<>V)&c <> W&e

where U,V,W are distinct variables.

(2.2) In the second stage, the list T is picked up from the difference list T-[] produced by top and executed, as specified by predicate exec(T). To execute T means to execute every member of T. To execute the pair A–>V means one of the three

following things:

(i) *to execute the predicate* A−>V;
(ii) to find some other member B−>W *in* T *and to execute one of the predicates* (A,B)−−>(V,W) *or* (B,A)−−>(W,V);
(iii) *to unify* A *and* V *(this corresponds to postponing the execution of* A*).*

The strategy followed by DTLOG *is to execute the members of* T *from left to right and try the execution alternatives (i-iii) in this order. Notice that after the execution of* T *the variables occurring in the top of* C *are instantiated to configurations. Continuing with the previous example, suppose the only clauses involving the agents* a, b, d *where*

b−>c1. (b,d)−−>(c2, c3). (a, b)−−>(skip, skip).

The outlined execution strategy would give us the following instance of C:

C = (skip<>skip)&c <> d&e

(2.3) The simplification stage eliminates from C *all occurrences of* 'skip' *(unless* C *is itself* 'skip'*), the result being* Y *(remember skip is the neutral element for* <> *and* &*). In the previous example we would obtain*

Y = c <> d&e

(2.4) The final stage checks whether the system specified by the DL *program is in a deadlock situation or not. Deadlock occurs iff the only possibility for executing the members of* T *is (2.2-iii) (assuming there are no agents for which* a−>a *or* (a,b)−−>(a,b)*). This is true iff* X *and* Y *are identical. Hence the final stage in the execution of* step(X,Y) *consists in verifying that* X *and* Y *are not identical.*

(3) The system automatically recovers from a deadlock situation by backtracking. If for some reason this is not desired, the cut symbol '!' should be inserted after the call 'exec(T)'.

(4) As a further facility, ordinary predicates can be turned into agents and inserted in configurations by writing a predicate P *in the form* agt(P)*. The following clause should then be added to the interpreter below:*

```
exec([agt(P)->skip|T]) :- !, P, exec(T).
```

THE INTERPRETER

```
/*   DTLOG : an interpreter for distributed logic   */

?-op(240, xf,     '^').   /* X^ is equivalent to X=>skip   */
?-op(230,xfx,    '->').   /* unary transition rule predicate   */
?-op(230,xfx,   '-->').   /* binary transition rule predicate   */
?-op(230,xfx,    '=>').   /* transition predicate   */
?-op(215,xfy,    '<>').   /* concurrent composition   */
?-op(200,xfy,     '&').   /* sequential composition   */

X^ :- X=>skip.
X=>X.
X=>Z :- step(X,Y), Y=>Z.

step(X,Y) :- top(X,T-[],C), exec(T), simplify(C,Y), compare(X,Y).

top(X1<>X2, T-T2, C1<>C2) :- !, top(X1, T-T1, C1), top(X2, T1-T2, C2).
top(X&Y, T-T1, C&Y) :- !, top(X, T-T1, C).
top(X, [X->C|T]-T, C).

exec([]).
exec([X->C|T]) :- X->C, exec(T).
exec([X->C|T]) :- delete(Y->D, T, T1),
                  ((X,Y)-->(C,D); (Y,X)-->(D,C)),
                  exec(T1).
exec([X->X|T]) :- exec(T).

simplify(skip<>X,Y) :- !, simplify(X,Y).
simplify(X<>skip,Y) :- !, simplify(X,Y).
simplify(X<>V,Z) :- simplify(X,Y), simplify(V,W),
                    (W=skip, Z=Y; Z=(Y<>W)), !.
simplify(skip&X,X) :- !.
simplify(X&W,Z) :- simplify(X,Y), (Y=skip, Z=W; Z=(Y&W)), !.
simplify(X,X).

compare(X,Y) :- X\==Y.          /* avoids looping */

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

EXAMPLES

(1) Our first example is a simplified version of the five philosophers problem. We only consider two philosophers (and two forks), since our main purpose is to specify a system with a deadlock. Each fork (0 or 1) is either up or down. Each philosopher (0 or 1) repeats for a specified number of life-cycles the actions of picking up a fork, picking up the other fork, putting down the first fork and putting down the second fork. The program follows:

```
philosophers(Lifespan) :-
     phil(0, Lifespan) <> phil(1,Lifespan) <> forkdown(0)
                                           <> forkdown(1)
                                           =>
                                  forkdown(0) <> forkdown(1).
phil(I,0) -> skip.
phil(I,N) -> getfork(I) & getfork(J) & putfork(I) & putfork(J)
             & phil(I,M)
          :- N>0, M is N-1, J is (I+1) mod 2.

(getfork(I), forkdown(I)) --> (skip, forkup(I)).

(putfork(I), forkup(I)) --> (skip, forkdown(I)).
```

(2) We are given two nonempty finite disjoint sets of integers, S0 *and* T0, and are required to produce two sets S *and* T *such that:*

(i) S *u* T = S0 *u* T0*;*
(ii) #S = #S0, #T = #T0 *('#' stands for cardinality);*
(iii) *every element of* S *is smaller than any element of* T.

We assume there are an S-process *and a* T-process *responsible for the* S-sets *and* T-sets *respectively. The* S- *and* T-processes *start by finding respectively the maximum value* X *and the minimum value* Y *of their input sets. These values are then exchanged and a check is made*

whether X<Y. If so, both processes stop and the exchange is not accepted. Otherwise the exchange is accepted and the processes repeat the actions already described.

```
partition(S0,T0,S,T) :- procS(S0) <> procT(T0) =>
                                        endS(S) <> endT(T)

procS(S0) -> exchgS(X,Y) & contS(X,Y,S) :- max(S0,X,S).

procT(T0) -> exchangT(X,Y) & contT(X,Y,T) :- min(T0,Y,T).

(exchgS(X,Y), exchgT(X,Y)) --> (skip,skip).

contS(X,Y,S) -> endS([X|S]) :- X<Y.
contS(X,Y,S) -> procS([Y|S]) :- Y<X.

contT(X,Y,T) -> endT([Y|T]) :- X<Y.
contT(X,Y,T) -> procT([X|T]) :- Y<X.

max([W|S],X,[W|Q]) :- max(S,X,Q), W<X, !.
max([X|S],X,S).

min([V|T],Y,[V|R]) :- min(T,Y,R), Y<V, !.
min([Y|T],Y,T).
```

REFERENCES

[Clo 80] CLOCKSIN, W. F., MELLISH, C. S.: "The RT-11 Prolog System", Dept. of A. I., University of Edinburgh, 1980.

[Mon 81a] MONTEIRO, L.: "A new proposal for concurrent programming in logic", Logic Programming Newsletter 1, Universidade Nova de Lisboa, Spring 1981.

[Mon 81b] MONTEIRO, L.: "Distributed logic: a logical system for specifying concurrency", CIUNL-5/81, Universidade Nova de Lisboa, 1981.

[Mon 82] MONTEIRO, L.: "A new presentation of distributed logic", CIUNL-7/82, Universidade Nova de Lisboa, 1982.

[Per 82] PEREIRA, L. Moniz: "Logic control with logic", First International Conference on Logic Programming, Marseilles 1982.

PROLOG FOR PROGRAMMERS

(AN OUTLINE OF A TEACHING METHOD) (1)

Feliks Kluźniak, Stanisław Szpakowicz

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

ABSTRACT

Professional programmers often consider Prolog too eccentric for their tastes. It may be easier for them to appreciate the language if we present it in terms of conventional programming notions. We must also demonstrate that it is competitive with other languages on their own terms. The usual presentation, stressing the programming-in-logic origin of Prolog and its problem solving explanation, is inadequate in this respect. A new teaching approach must be developed. This paper is a brief outline of the rationale for and the main features of such an approach.

INTRODUCTION

Prolog is usually presented as an exemplification of logic programming. The procedural interpretation has always been given as a secondary, i.e. somehow less important, way of viewing Prolog programs. Moreover, it has been strongly tinged with a problem solving flavour. This is hardly surprising, as a large part of the logic programming community has a marked artificial intelligence background.

After several years of experience with learning and teaching Prolog we have come to the conclusion that the usual presentation is often inappropriate for programmers who lack this background. The purely "logical" approach (cf. [4]) might well be the best method of introducing programming to non-programmers. It is less suitable for practising programmers, who find it hard to believe that Prolog is a convenient tool for solving practical problems.

Of course, one of the reasons is the relative inefficiency of many available implementations. The main difficulty, though, is to teach the programmer how to incorporate techniques specific to the language into his familiar patterns of thought. Prolog does not easily fit the established patterns, and nobody can be asked simply to throw them away and start learning his trade all over again.

Consequently, if Prolog is to be accepted as a general-purpose programming language rather than as a very special tool for some artificial intelligence tasks (2), a serious effort must be made to facilitate the programmer's transition to a different style of thinking about programming. Prolog ought to be made fully comprehensible within the "normal" (algorithmic, operational) conceptual framework (3). Naturally, one cannot conceal the fact that the language is somewhat unusual, but its peculiarities must be made plausible within this framework (as syntactic innovations designed to increase program clarity, slight generalisations of conventional mechanisms which enhance expressive power, etc.). In short, the aim is to shown that Prolog is a normal programming language, and is different inasmuch as it is better.

Once this is accepted, the logic programming interpretation of Prolog will become increasingly more important while the language is being used. It will have been introduced simply as a useful programming tool (e.g. the declarative interpretation of clauses helps to understand them in spite of the inherent complexity of their execution) but its repeated use will make the user accept it as his own way of thinking (4). The transition will then be complete, the conceptual framework painlessly enlarged and the initial, "normal" explanation obsolete.

This rather long introduction is intended to justify a new way of presenting Prolog fundamentals in the first chapter of our text book

(1) This paper is based on our presentation at the Logic Programing Workshop, Debrecen 1980; the editors of the proceedings deemed it "too insubstantial" without "more sophisticated examples of Prolog programs".

(2) The situation of Prolog today is comparable to that of Lisp in the days when it was generally treated only as a string processing language.

(3) Warren's programmer-oriented presentation (e.g. [5]), though commendable, is not comprehensive enough. In many respects our approach is in the same spirit as that of van Emden [2].

(4) Look how the concept of type in Pascal has gained acceptance with programmers who started with Fortran.

prepared for a Polish publisher [3]. We do not claim that our attempt is completely satisfactory, but we think it is sufficiently novel to make it worth presenting to the logic programming community.

Our introductory course of Prolog fundamentals lasts about ten hours, the book version occupies over 60 typewritten pages. We shall, therefore, limit ourselves to a brief outline of the way we try to answer the four principal questions posed by a programmer who attempts to learn Prolog, viz.

— how to deal with terms and what to make of unification,
— how to interpret the multiclause form of procedures,
— what is backtracking and how to use it,
— how to interpret and use the cut procedure (the slash).

TERMS AND UNIFICATION

Our treatment of this subject is based on the idea of treating terms as descriptions of data types. It stems from the observation that a term with variables represents not only a concrete object but also the class of objects into which it may be transformed through instantiation of variables. Thus, a variable denotes the "universal type", a ground term — a class of only one object.

A variable per se presents no difficulties. It may be likened to the **const** *formal parameter of original Pascal [6], i.e. an object whose exact nature is unknown until it becomes (permanently) instantiated. According to a more technical interpretation, variables are always automatically dereferenced. The dereferencing makes it impossible to distinguish between a pointer and the object pointed at. There is, therefore, no question of "resetting" a variable.*

Functional notation is a simple way of describing the nature of composite objects. The intended meaning of a composite object — such as (14,20) — is unclear unless it is accompanied by a type description consisting of a mnemonically suggestive name, such as HOUR (14,20) *or* PRICE (14,20).

"Fix" functors [5] are introduced to simplify the written form of terms with nested parentheses. It is also this concern for program readability that makes us name objects by terms that directly describe the relevant aspects of their structure.

The expressive power of Prolog's notation is easily illustrated by the following type specifications (we use the Marseilles convention of prefixing variable names with an asterisk):

**A.*B.*C.NIL (a list of exactly three elements),*
**A.*B.*C (a list of at least two elements),*
**A.*A.*C (a list which has the same objects as its first two elements [6]), etc.*

In one respect Pascal is more powerful: apart from providing a conventional name, its type definition describes the component types of a composite object. In Prolog we cannot define eg. a proper list that ends with a NIL, *without using a checking procedure which excludes improper data. However, this is a small price to pay for generality exceeding — in some respects — even that of the generic program units of Ada [1].*

All these points are rather spectacularly illustrated by an unsophisticated example. Consider a procedure which is supposed to compute the head and tail of a list. Obviously, its heading will be something like +CARCDR (???, ???, ???) *[7]. One of the parameters must be a list, so we place an appropriate type description in the only natural position, namely in the heading, e.g.*

```
+CARCDR (*CAR.*CDR, ???, ???).
```

It now becomes clear that a complete specification of the procedure may be written as

```
+CARCDR (*CAR.*CDR, *CAR, *CDR).
```

Applying the same arguments to a procedure which CONStructs *a list, we see that*

```
+CONSCARCDR (*CAR.*CDR, *CAR, *CDR).
```

is about the most concise and clear complete specification of both procedures we can hope to obtain. Its elegance is not a trick: it simply reflects our knowledge that the procedures are complementary.

In principle, such specifications are executable only if we can invent a parameter-passing mechanism which accesses the type description of the formal and actual parameters and performs all the necessary actions. Unification is such a mechanism and its simplicity seems miraculous in this context.

MULTICLAUSE PROCEDURES

Whenever a procedure is executed, the choice between its possible control paths is determined by the external environment. It is widely accepted that the influence of the environment should always be made as explicit as possible: the execution of an ideal procedure depends only on the properties of its actual parameters. The nature of this dependency, however, is still largely obscured in conventional programming languages by the piecemeal and indirect fashion in which these properties are used to determine a control path.

Not so in Prolog. Here we have precisely as many control paths as there are clauses in the procedure. The choice between them can often be made explicitly dependent on parameter properties, thanks to the generality of terms as type descriptions. We can even take advantage of it to specify the properties of the whole set of parameters, as in the following example:

```
+ELEMENT (*X, *X.*Z).
+ELEMENT (*X, *Y.*Z)   -ELEMENT (*X, *Z).
```

Here, the choice is determined by mutual relations between both parameters.

BACKTRACKING

The clause headings in a procedure do not always contain descriptions of disjoint classes of parameter sets, and these descriptions

---

[5] The name is, perhaps, awkward, but it helps to avoid the confusion that arises from calling them "operators".

[6] There is also the dual interpretation: different objects can share components (no wonder, if variables are regarded as pointers).

[7] We have chosen the original Marseilles syntax because it permits the perspicuous notation for procedure "declaration" (a plus) and "call" (a minus).

might not cover every case. A call with a set of parameters that does not fall into any class, e.g.

```
−ELEMENT (ELEM, NOTALIST).
```

is obviously erroneous in our model. The procedure cannot be activated because of a type conflict between the actual and formal parameters.

In the majority of modern programming languages an exception is raised when a runtime error is encountered. It can be intercepted by an error-handler specified by the programmer. It is therefore quite natural to interpret backtracking as an error propagation process: the "superfluous" clauses of some procedures are the error-handlers. The method of error propagation (not up the dynamic activation chain) is unusual, but it presents no conceptual difficulties. One might even argue that it is easier to visualise undoing a computation than undoing the current state of the control stack.

We use this error-handling mechanism to provide "extended type checking": a procedure can be executed in order to examine properties of parameters that cannot be expressed in term notation. The failure of LESS *in the clause*

```
+INSERT (*NODE, TREE (*ROOT, *LEFT, *RIGHT))
        −LESS (*NODE, *ROOT) −INSERT (*NODE, *LEFT).
```

has exactly the same effect as a mismatch of the actual and formal parameters.

Backtracking can also be "misused" to implement generators such as the first call of ELEMENT *in*

```
+INTERSECT (*L1, *L2) −ELEMENT (*X, *L1) −ELEMENT (*X, *L2).
```

The elegance of this example makes it highly desirable to regard such "illicit" use of the error propagation mechanism as a legitimate, powerful programming technique. In other words, backtracking is shown to be more than just error-handling.

This example also serves as an introduction to the declarative interpretation of clauses. It may be difficult to understand INTERSECT *in operational terms (try to trace its execution even with simple data !), but the correctness of "two sets intersect if some object belongs to both of them" is obvious ([8]).*

THE CUT PROCEDURE

Some generators can run indefinitely, producing more and more output. There should be a way of stopping them once we are satisfied with the results obtained so far. Invocation of the cut procedure (the slash) means that we accept the results produced till now, and we do not want any others ([9]).

The cut should be used sparingly, as it makes declarative interpretation difficult. However, one of its uses significantly increases the expressive power of Prolog: a cut combined with backtracking gives the effect of negation. The procedure

```
+NOT (*X)  −*X  −/  −FAIL.
+NOT (*X).
```

enables us to camouflage this trick; and the extreme simplicity and power of the variable literal go a long way toward making the language even more attractive.

FINAL REMARK

A coherent and complete programmer-oriented introduction to Prolog following the ideas presented in this paper was given in 1980/81 at Warsaw University. Although the course does not seem perfect, the main ideas have proven entirely satisfactory.

ACKNOWLEDGEMENT

We are grateful toRobert Kowalski, for suggesting considerable improvements to our style and for directing our attention to reference [2].

REFERENCES

[1] Preliminary ADA reference Manual. SIGPLAN Notices, *14*, 6, 1979.
[2] M. H. van EMDEN: "Programming in Resolution Logic". Machine Intelligence 8, pp. 266-299, 1977.
[3] F. KLUŹNIAK, S. SZPAKOWICZ: "Prolog". Wydawnictwa Naukowo-Tehniczhe, Warszawa (in preparation).
[4] R. KOWALSKI: "Logic for Problem Solving". North-Holland, 1979.
[5] D. H. D. WARREN: "Logic Programming and Compiler Writing". DAI Memo 44, University of Edinburgh, 1977.
[6] N. WIRTH: "The Programming Language PASCAL". Acta Informatica *1*, pp. 35-63, 1971.

([8]) Strictly speaking, more is needed to describe this procedure in detail. We must, for instance, take care to mention and justify reservations about lists that contain non-ground terms which can be different but unifiable.

([9]) The cut is similar to the COMMIT *operation in data-base programming. Notice also that its form in the Marseilles syntax is a slight variation of the acceptance mark (√) used by teachers grading examination papers, in checklists, etc.*

A PORTABLE PROLOG TRACING PACKAGE

Libor A. Spacek
Department of Computer Science
University of Essex, Colchester, England

The nature of Prolog's procedural semantics makes imperative the need for a tracing facility fully capable of describing the backtracking behaviour.

We are aware of only one such package to date, namely the one incorporated into the DEC-10 Prolog-10, version 3.3 (1981) by Byrd, Pereira and Warren and distributed by Edinburgh University. However, there are now many more Prolog implementations and new ones are appearing almost daily. In our experience with teaching Prolog, this language is better understood if a good tracing facility is available right from the beginning.

Prolog's control mechanism, making use of unification and pattern directed invocation, is ideally suited to writing such a package in Prolog

itself. The package can then be loaded and interpreted together with the user's program. This has obvious advantages of portability and savings of store when the package is not needed. Both of these are important considerations for the latest microprocessor based implementations.

In order to ensure portability we have tried to use only standard Prolog predicates which are likely to be available in most implementations, even though not necessarily under the same name. For instance, we avoid using "depth(D)" which unifies D with the number of direct ancestors of the current goal. This Prolog-10 predicate would have been convenient for our purpose but it is not generally available.

Tracing package

```
trace(X):-

    (traceon;assert(traceon)), % Switch on the tracing if not on.

% Now trace(X) asserts two short clauses for the term X.
% They ensure that a call is made to the tracing diagnostics clause
% for X and from there to the user's clause(s) for X every time that
% X is invoked.

% The following clause switches the tracing on just before the user's
% clauses for X are entered. This enables the tracing of recursive calls.

    asserta((X:- assert(traceon),fail)),

% The diagnostics clause itself will be inserted at the top:

    asserta((X:-

        traceon,!,          % and used only when traceon is present.
        retract(traceon),   % So that call(X) will not come back here.
        enterex(X,D2),!,
            (
             call(X),
             forwardx(X,D2),
             backwardx(X,D2);

             failx(X,D2),
             !,fail
             )
    )).

% The rest of this package can be put into a separate file and compiled.

% The following is done before each new call on X:

enterx(X,D2):-

    (stack(D1,_);D1 is 0,asserta(stack(0,1))), % Initialise stack.
    D2 is D1 + 1,           % Increment depth of recursion and
    asserta(stack(D2,1)),   % set solutions counter to one.
    writeit(D2,'-',X,'call',1).

% The following is done for all exits from X:

forwardx(X,D2):-

    retract(stack(D2,B)),
    stack(D1,Blast),
    writeit(D2,'+',X,'solution',B),
    B2 is B + 1,
    asserta(stack(D2,B2)),    % Increment the solutions counter.
    asserta(stack(D1,Blast)), % Decrement depth of recursion.
    !.                        % Do not backtrack into this clause.

% The following is used when backtracking to X:

backwardx(X,D2):-

    (true;                    % When going foward do nothing.

     stack(D2,B),             % Do this when backtracking.
     asserta(stack(D2,B)),    % Retrieve and put at the top the record
     writeit(D2,'-',X,'backtracking',B),  % for the backtracking goal.
     !,fail                   % But do not disturb the flow of control.
    ).

% The following is done when X fails:

failx(X,D2):-
            stack(D2,N),
            clearstack(D2,N),
            writeit(D2,'?',X,'failed',' ').

clearstack(Dgiven,N):-

    stack(D,B),!,             % The cuts in this clause are there only to
    (                         % allow optimisation of tail recursion.
     D >= Dgiven,
     retract(stack(D,B)),!,
     clearstack(Dgiven,N);    % Clear all successors of the failed goal
     (
      N > 1,                  % as many times as it has been
      retract(stack(_,_)),    % backtracked into.
      Nm1 is N - 1,
      clearstack(Dgiven,Nm1);

      true
     )
    ).

writeit(D,Sign,X,Message,Number):-

    Tabn is D - 1,
    n1,
    tab(Tabn),
    write(Sign),
    write(D),
    write(' '),write(X),
    write(' '),write(Message),
    write(' '),write(Number).

untrace(X):-

    clause(X,Q1),
    retract((X:-Q1)),
    clause(X,Q2),
    retract((X:-Q2)),
    abolish(stack,2),         % retract all instances of stack.
    !.
```

```
% An example program to illustrate top-down and bottom-up
% search of a tree hierarchy formed by transitive relations.

parent(john,eve).
parent(john,steve).
parent(jane,eve).
parent(jane,steve).
parent(steve,phil).
parent(ada,phil).
parent(phil,gill).

ancestor(X,Z):- parent(X,Z).
ancestor(X,Z):- var(Z), !,
                parent(X,Y),
                ancestor(Y,Z).
ancestor(X,Z):- parent(Y,Z),      % If Z is known search for ancestors
                ancestor(X,Y).    % in bottom up direction.
```

Now follows a session with the tracing package and the above example program:

```
Prolog-10  version 3.3
Copyright (C) 1981 by D. Warren, F. Pereira and L. Byrd

| ?- ['trace.pr1','exampl.pr1'].

trace.pr1 consulted    668 words    0.21 sec.

exampl.pr1 consulted    146 words    0.05 sec.

yes
| ?- trace(ancestor(_,_)).

yes
| ?-(ancestor(jane,P).

-1 ancestor(jane,_31) call 1
+1 ancestor(jane,eve) solution 1
P = eve ;

-1 ancestor(jane,eve) backtracking 2
+1 ancestor(jane,steve) solution 2
P = steve ;

-1 ancestor(jane,steve) backtracking 3
  -2 ancestor(eve,_31) call 1
   ?2 ancestor(eve,_31) failed
  -2 ancestor(steve,_31) call 1
  +2 ancestor(steve,phil) solution 1
+1 ancestor(jane,phil) solution 3
P = phil ;

-1 ancestor(jane,phil) backtracking 4
  -2 ancestor(steve,phil) backtracking 2
    -3 ancestor(phil,_31) call 1
    +3 ancestor(phil,gill) solution 1
  +2 ancestor(steve,gill) solution 2
+1 ancestor(jane,gill) solution 4
P = gill ;

-1 ancestor(jane,gill) backtracking 5
  -2 ancestor(steve,gill) backtracking 3
    -3 ancestor(phil,gill) backtracking 2
      -4 ancestor(gill,_31) call 1
       ?4 ancestor(gill,_31) failed
    ?3 ancestor(phil,_31) failed
  ?2 ancestor(steve,_31) failed
?1 ancestor(jane,_31) failed
no
| ?- ancestor (P,gill).

-1 ancestor(_24,gill) call 1
+1 ancestor(phil,gill) solution 1
P = phil ;

-1 ancestor(phil,gill) backtracking 2
  -2 ancestor(_24,phil) call 1
  +2 ancestor(steve,phil) solution 1
+1 ancestor(steve,gill) solution 2
P = steve ;

-1 ancestor(steve,gill) backtracking 3
  -2 ancestor(steve,phil) backtracking 2
  +2 ancestor(ada,phil) solution 2
+1 ancestor(ada,gill) solution 3
P = ada ;

-1 ancestor(ada,gill) backtracking 4
  -2 ancestor(ada,phil) backtracking 3
    -3 ancestor(_24,steve) call 1
    +3 ancestor(john,steve) solution 1
  +2 ancestor(john,phil) solution 3
+1 ancestor(john,gill) solution 4
P = john ;

-1 ancestor(john,gill) backtracking 5
  -2 ancestor(john,phil) backtracking 4
    -3 ancestor(john,steve) backtracking 2
    +3 ancestor(jane,steve) solution 2
  +2 ancestor(jane,phil) solution 4
+1 ancestor(jane,gill) solution 5
P = jane ;

-1 ancestor(jane,gill) backtracking 6
  -2 ancestor(jane,phil) backtracking 5
    -3 ancestor(jane,steve) backtracking 3
      -4 ancestor(_24,john) call 1
       ?4 ancestor(_24,john) failed
      -4 ancestor(_24,jane) call 1
       ?4 ancestor(_24,jane) failed
    ?3 ancestor(_24,steve) failed
  -2 ancestor(_24,ada) call 1
   ?2 ancestor(_24,ada) failed
   ?2 ancestor(_24,phil) failed
 ?1 ancestor(_24,gill) failed

no
```

SIMULATING COROUTINING FOR THE 8 QUEENS PROBLEM

John Gallagher

Department of Computer Science,
Trinity College, Dublin, Ireland

1.1 *SUMMARY*

Logic programs to solve the eight queens problem have been discussed by several authors [1, 2, 3]. In [1], the problem is used to illustrate a coroutining method, while in [2] and [3] an intelligent backtracking mechanism is described. In both these approaches, the performance of a program, which is inefficient when run using the standard Prolog search strategy, is improved by extra control in the interpreter. The approach here is to take the same program as a starting point, and derive from it a new program which is run efficiently by the usual interpreter. The new program simulates a coroutining behaviour in the original program, but a more complicated coroutining than that provided by the interpreter in [1].

A systematic transformation of an inefficient program into an efficient one is an alternative to extra run-time control, allowing one to keep the advantages of the simplicity of the usual search strategy, which can be compiled into an even more efficient form.

1.2. *A LOGIC PROGRAM FOR THE PROBLEM*

The eight queens problem is to place eight queens on a chessboard so that none attacks any other. If one takes into account that all the queens must lie on separate rows and columns, a solution to the problem can be represented by some permutation of the numbers 1 to 8, giving the column numbers of the queen in each of the eight rows. The permutation [1,2,3,4,5,6,7,8], for instance, is the one in which the queens all lie along a diagonal.

The predicate permutation(L,M) *means that* M *is some permutation of* L.

As answer is then a permutation in which no two queens lie on the same diagonal; this property is called safe(Perm).

Thus the program at the top level is

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

The procedures for permutation and safe are defined as follows. (The notation [X,..L] *for lists means that* X *is the first item in the list, and* L *is the rest of the list.)*

```
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.
```

1.3. *TWO PROBLEMS WITH THIS PROGRAM*

(1) The standard interpreter for logic programs gives a depth first search. Therefore, given the query solution(Perm)?, *the interpreter must solve* permutation([1,...8],Perm) *completely, before passing on to check* safe(Perm). *In other words, this strategy places all the queens on the board before checking whether any of them is attacked.*

(2) The inefficiency is compounded by the backtracking mechanism. When two queens are found on the same diagonal, safe(Perm) *fails, and a new permutation is generated. However, the new permutation may well leave the conflicting queens where they were, thus ensuring another failure in* safe(Perm).

Coroutining alleviates the first problem. The programmer notes that the variable Perm *occurs in both top level goals, and so annotates the program to indicate that the interpreter is to solve these goals cooperatively. Whenever the predicate* permutation([1,...8],Perm) *produces a new partial* permutation, *the predicate* safe(Perm) *consumes it, checking that it is consistent.*

The procedure safe(Perm) *is expanded depth first until it "needs" the next queen, that is, until just before a variable within* Perm *is unified with a non-variable. Unfortunately, this method, which is used in [1], only checks the safety of the first queen during coroutining, as the authors point out. This is because* safe(Perm) *checks the safety of each queen in turn, starting with the first. The program remains inefficient; either the definition of safe must be changed, or a more powerful coroutining mechanism must be used.*

The intelligent backtracking method described in [2, 3] analyses the causes of failure when two queens are found to be in conflict, and responds by backtracting to a goal which has caused the failure. This means that one of the queens which was conflicting will be moved. A much better performance follows, but it is still necessary to generate a complete permutation before any safety checks are done.

1.4. *A TRANSFORMATION OF THE PROGRAM*

The program below simulates coroutining by dealing explicitly with

two lists of goals, one generated by permutation([1,...8],Perm), *and the other by* safe(Perm).

A list of goals is represented

G1:G2:...:Gn (An infix function ":" may be defined.)

A predicate

solve(Goals1, Goals2)

means that the two lists of goals, Goals1 *and* Goals2, *are waiting to be solved.* Goals1 *results from* permutation, *and* Goals2 *from* safe. *During the computation, it is required to switch between goals in the two lists.*

It must be decided at what point to suspend computation of the permutation and transfer to the list of goals generated by safe. This depends on finding the points in the program at which the shared variable, Perm, *is given a new value. Whenever* Perm *or a variable within* Perm *is unified with a non-variable, control must pass to the procedure* safe, *so that the compatibility of the new value can be tested.*

An analysis of the program shows that there are two such transfer points. Whenever the permutation procedure is called, a variable L *is unified with either* [X,..L1] *or with* []. *Secondly, the clause* remove(X,[X,..L],L). *unifies* X, *(which is a variable within* Perm*), with a constant.*

Whenever either of these points is reached, therefore, control might pass to safe. *However, it may be detected that the consumption of the terms* [X,..L] *or* [] *by* safe *is guaranteed, and does not result in the solution of any new goals. Therefore it is only necessary to coroutine after the clause* remove(X,[X,..L],L) *is used.*

What happens to the second list of goals at this point? Whenever safe([X,..L]) *is expanded, two goals arise,* nodiagonal(X,L,1) *and* safe(L). *Since both of these contain the shared variable* L, *it is desirable to continue the coroutining on both these goals. When* nodiagonal(X, [X1,..L],N) *is expanded, three goals,* noattack(X,X1,N), N1 is N+1, *and* nodiagonal(X,L,N1) *arise. Of these, the first two may be executed immediately, since* X, X1 *and* N *are all known. After a number of expansions of* safe(Perm), *a list of goals of the following form has arisen.*

```
nodiagonal(X1,L,N1):nodiagonal(X2,L,N2): ... :safe(L).
```

Each time control passes to this list of goals, it is transformed by expanding out each goal and solving all the noattack *goals. The predicate*

transform(Goals, Newgoals)

represents the state of the list before and after the transformation. The top level of the program is

```
solution(Perm) <-
        solve(permutation([1,...8],Perm), safe(Perm)).
```

The definitions of solve and transform are as follows.

```
solve(permutation(L,[Q,..M]), Goals) <-
        remove(Q,L,L1),
        transform(Goals, Newgoals),
        solve(permutation(L1,M), Newgoals).
solve(permutation([],[]), Goals).

transform(safe([Q,..List]), nodiagonal(Q,List,1):safe(List)).
transform(nodiagonal(Q,[Q1,..L],N):G, nodiagonal(Q,L,N1):H) <-
        noattack(Q,Q1,N),
        N1 is N+1,
        transform(G,H).
```

The definitions of remove and noattack are as above.

1.5. *REMARKS*

The above program effectively places each queen and then checks whether it conflicts with any queen which is already on the board. Intelligent backtracking might further improve it, but the need for it has diminished because failures are detected at a much earlier stage.

The program resembles the original program. The definitions of permutation, safe *and* nodiagonal *are incorporated into the new definitions* solve *and* transform.

The analysis of the original program, sketched above, bears a resemblance to the dataflow approach to logic programs [4], in that it concerns the propagation of values through the program.

Logic programs, because of their simple syntax and the flexibility of their control component, are much more suited to this kind of manipulation, than programs in other languages.

REFERENCES

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

[2] BRUYNOOGHE, M.: "Analysis of dependencies to improve the behaviour of logic programs". Proc. 5th Conf. Automated Deduction, Springer-Verlag 1980.

[3] PEREIRA, L. M. and PORTO, A.: "Selective backtracking for logic programs". Proc. 5th Conf. Automated Deduction Springer-Verlag, 1980.

[4] MORRIS, P.: "A dataflow interpreter for logic programs". Logic Programming Workshop, Hungary 1980.

AVL-TREE INSERTION RE-VISITED

Philip Vasey

Department of Computing
Imperial College, London, England

An investigation of Maarten van Emden's logic program for AVL-tree insertion (LOGIC PROGRAMMING NEWSLETTER 2), shows that it fails to find a solution to the query

INSERT(avl(avl(NIL, 1, −, NIL) , 3 , < , NIL) , 2 , *t , *c)

even though there should be a valid instance, namely

*t = avl(avl(NIL, 1, −, NIL) , 2 , − , avl(Nil, 3, −, NIL))

*c = NO

A closer analysis of the program reveals that a third assertion for the TABLE2 *relation is required.*

TABLE2(− , − , −).

MORAL: *even logic programs should be verified, or better still they should be derived from an intuitively correct axiomatisation.*

PURE LISP IN PURE PROLOG

Luís Moniz Pereira
António Porto
Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

An evaluator for pure Lisp in pure Prolog is presented below:

1) eval(E, U, R) *takes an* S-expression *and evaluates it to* R, *in the context of association list* U *comprising two element lists pairing atoms to their associated values*
2) *It features the Prolog system predicates:*

X=Y	X unifies with Y
atom(A)	A is an atom
integer(I)	I is an integer
atomic(A)	A is an atom or integer

3) *Prolog syntax is used for lists. As usual in Lisp 'false' is represented by the empty list, in this case '[]'*
4) *Examples of calls are:*

```
?- eval( [ff,X], [ [X,[[1,2],3]] ], R).        gives  R=1

?- lisp.
[alt,[1,2,3,4,5]].                             gives  [1,3,5]
```

5) *'equal' is made primitive rather than the implementation oriented concept 'eq'*
6) *numeric functions and predicates are left out*
7) *space may be recovered by garbage collecting each cycle:*

```
lisp :- repeat, solve( (read(E), eval(E,[],R), write(R), nl,nl) ), fail.

solve(G) :- G, !.
```

where 'repeat' is a system predicate that always solves again

8) *'assert' is used as an optional convenience for storing functions interactively*

```
?- op(10,fx,').

lisp :- read(E), eval( E, [], R), write(R), nl, nl, lisp.

eval( A, U, R) :- atomic(A),
                  ( ( integer(A) ; A=[] ; A=true ), R=A ;
                    assoc( A, U, [_,R]) ;
                      error ).

eval( [quote,X], _, X).

eval( 'X, _, X).

eval( [cond,[T,B]|L], U, R) :- eval( T, U, ET),
                               ( ET=true, eval( B, U, R) ;
                                 eval( [cond|L], U, R) ).

eval( [cond], _, []).

eval( [list,[X|L]], U, [EX|EL]) :- eval( X, U, EX), eval( [list,L], U, EL).

eval( [list], _, []).

eval( [car,X], U, Y) :- eval( X, U, EX) ( EX=[Y|_] ; error ).

eval( [cdr,X], U, Y) :- eval( X, U, EX) ( EX=[_|Y] ; error ).

eval( [cons,X,Y], U, [EX|EY]) :- eval( X, U, EX), eval( Y, U, EY).

eval( [atom,X], U, R) :- eval( X, U, EX), ( atomic(EX), R=true ; R=[] ).

eval( [equal,X,Y], U, R) :- X=Y ;
                            eval( X, U, EX),
                            eval( Y, U, EY),
                            ( EX=EY, R=true ; R=[] ).

eval( [F|L], U, R) :- assoc( F, U, P),
                      ( P=[_,EF], eval( [EF|L], U, R) ; error ).

eval( [[lambda,V,E]|A], U, R) :- evalist( A, U, EA),
                                 pair( V, EA, P),
                                 append( P, U, W),
                                 eval( E, W, R).

eval( [not,X], U, R) :- eval( X, U, EX), ( EX=true, R=[] ; R=true ).

eval( [and, X,Y], U, R) :- eval( X, U, EX), ( EX=[], R=[] ; eval( Y, U, R) ).

eval( [or, X,Y], U, R) :- eval( X, U, EX), ( EX=[], eval( Y, U, R) ; R=EX ).

eval( [defun,N,A,E], _, N) :- assert( definition( N, [lambda,A,E]) ).

eval( [eval,X], U, R) :- eval( X, U, EX), eval( EX, U, R).

/* extra notation */

eval( [null,X], U, R) :- eval( [equal,X,[]], U, R).

eval( [if,C,A,B], U, R) :- eval( [cond,[C,A],[true,B]], U, R).

/* association list */

assoc( X, _, [_,R]) :- definition( X, R).

assoc X, [[Y,VY]|U], R) :- X=Y, R=[Y,VY] ; assoc( X, U, R).

/* examples of defined functions*/

definition( ff, [lambda,[x],[if,
                             [atom,x],
                             x,
                             [ff,[car,x]]]] ).

definition( alt, [lambda,[u],[if,
                              [null,u],
                              [],
                              [if,
                                 [null,[cdr,u]],
                                 u,
                                 [cons,[car,u],[alt,[cdr,[cdr,u]]]]]]]
          ).

/* utilities */

error :- write(error), tab(2), abort.

evalist([H|T],U,[EH|ET]) :- eval( H, U, EH), evalist(T,U,ET).
evalist(  []  ,_,  []  ).

pair([X|Y],[U|V],[[X,U]|P]) :- pair(Y,V,P).
pair(  []  ,  []  ,  []  ).

append([H|T],L,[H|R]) :- append(T,L,R).
append(  []  ,L,  L  ).
```

REFERENCE: JOHN McCARTHY and CAROLYN TALCOTT: "Lisp Programming and Proving" (draft), Stanford University 1981.

abstracts

A Note on Garbage Collection in Prolog Interpreters

Maurice Bruynooghe

Department Computerwetenschappen
K. U. Leuven, Heverlee, Belgium

The principal runtime structures needed to execute Prolog programs are the environment stack and, depending on the method, either a global or a copy stack. Methods have been developed to remove all unnecessary information from the environment stack. The global copy stack, however, can contain very large data structures. Some of them can become inaccessible. Although bactracking always releases them, this can be insufficient for large programs and garbage collection can be necessary. Marking of the global copy stack normally starts from all references to it, i.e. from the environment stack. The paper describes a method which reduces the number of starting points by considering the state of the computation. Some data structures are recognized as garbage, although accessible, because they are not needed to complete the computation.

Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs

Maurice Bruynooghe

Department Computerwetenschappen
K. U. Leuven, Heverlee, Belgium

Prolog programs are very error-prone, small typographical errors do not result in compile time errors but in programs with different unintended meanings. The paper contains a proposal to improve the situation. It suggests to add redundant information about the flow of data through clauses and about the possible values of arguments and gives a method to analyse the consistency between this additional information and the text of the Prolog clauses.

An Algorithm for Interpreting Prolog Programs

M. H. van Emden

Department of Computer Science
University of Waterloo, Waterloo, Canada

We describe in pseudocode the main routine of a Prolog interpreter. The algorithm is developed in several steps. Initially a search algorithm for trees is described. After a review of the Prolog theorem prover, the tree-search algorithm is applied to the search space of the theorem prover. Several inefficiencies of the result are then eliminated by the introduction of proof trees and structure sharing.

An Applicative Language for Highly Parallel Programming

András Domán

Institute for Co-ordination
of Computer Techniques
1054, Budapest, Akadémia u. 16. Hungary

Applicative languages based on functional semantics, examplify the perhaps unsurprising fact that mathematics, with its considerable power and history, may also be a programming language — in a quite natural way. An obvious consequence of this is that our /programming/ problems should be formulated as mathematical problems by means of an appropriate procedural from.

The aim of our paper is twofold. First, through the presentation of a functional language /PARAFLOC/ developed and implemented by us, we intend to illustrate the possibility of a simple mathematical description suitable for high-level programming. Second, we show that functional languages represent a very natural way of algorithm description not only for sequential, but also parallel execution without the user having to consider parallelism as a specific feature either on the algorithmic or the programming level.

Logic Programming in the Modelling of Machine Parts

B. E. Molnár, A. Márkus

Computer and Automation Institute
Hungarian Academy of Sciences,
Budapest, Hungary

The representation of the professional knowledge of the mechanical engineer has to cope with two problems: the representation of the procedural knowledge and that of the experience gained from objects designed earlier. Logical programming gives a means to integrate the efforts towards solving the problem.

As an application of the above mentioned ideas experiments are presented with a program written in PROLOG for modelling machine parts with the aim of generating a computer aided classification of machine parts.

Fixture Design by Prolog

J. Farkas, J. Filemon

Technical University Budapest,
Budapest, Hungary

A. Márkus, Zs. Márkusz

Computer and Automation Institute,
Hungarian Academy of Sciences
Budapest, Hungary

This paper describes a program for designing fixtures built out of a set of given elements. Its input is a description of the shape of the workpiece, the machining, and the co-ordinates of the supports and of the clamps to be realized. It generates a sequence of drafts of fixtures automatically. The program has been written in Prolog, the favourite language of logic programming. Since this system is meant to be a prototype of some future developments, even the methodological points have been explained in detail. Finally some estimates of the limits of this approach are given.

Applicative Communicating Processes in First Order Logic

Marco Bellia, Pierpaolo Degano and *Giorgio Levi*

Istituto di Scienze dell'Informazione
Università di Pisa, Pisa, Italy

Enrico Dameri

Systems & Management SpA,
Area Tecnologie Software
Pisa, Italy

Maurizio Martelli

Istituto CNUCe - C.N.R., Pisa, Italy

We describe a first order applicative language for the specification of deterministic systems of communicating computing agents à la Kahn-MacQueen. Both the sequential and parallel interpreter we give are based on lazy evaluation, are demand driven and can handle infinite streams and non-terminating proce-

dures. An equivalent least fixed-point semantics is then presented which neatly copes with the above features of the language. It is worth noting that computations in our logical based model can be considered as formal proofs, thus making formal reasoning about programs easier.

From Term Rewriting Systems to Distributed Programs Specifications

M. Bellia, E. Dameri, P. Degano, G. Levi and *M. Martelli*

Istituto di Elaborazione dell'Informazione — C.N.R., via S. Maria, 46, 56100 Pisa, Italy

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.

Epilog: a Language for Extended Programming in Logic

António Porto

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

Epilog is a Prolog-based language for logic programming with extensions for powerful control of the execution. The extensions are based on having several (procedurally distinct) *AND* connectives instead of just one as in Prolog. The language allows for mixed/meta-level statements, and Prolog's *cut* has been replaced by higher-level constructs.

This paper describes Epilog along with the construction of an interpreter for it, written in Prolog, aiming at a clear understanding of the intended procedural semantics. Some examples of application are shown.

ORBI-An Expert System for Environmental Resource Evaluation Through Natural Language

Luís Moniz Pereira
Paul Sabatier
Eugénio Oliveira

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

We describe a computer system, interrogatable in a flexible subset of Portuguese and implemented in Prolog on a small machine, which embodies and assimilates expert knowledge on environmental biophysical resource evaluation, is capable of explaining the application of that knowledge to a territorial data base, and also of answering questions about its linguistic abilities. The system was developed in one year, under contract with the Portuguese Department of the Environment.

A New Presentation of Distributed Logic

Luís Monteiro

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

Distributed logic is presented as the logical formalization of a general abstract model of concurrency, called a distributed transition system. The operational and the declarative definitions of the semantics of distributed logic are outlined.

An Informative, Adaptable and Efficient Natural Language Consultable Database System

Jean François Pique

Faculté de Medicine
Université de Marseille
13288 Marseille, France

Paul Sabatier

Universidade Nova de Lisboa
Portugal

Within a unique formalism, logic is a powerful theoretical basis allowing to represent a database and to express the different (syntactic, semantic and deductive) processings involved when this database is consulted in natural language.

In this framework, we describe a complete system using a three truth-valued logic rigorously defined. This logic allows a very fine representation of questions semantics and lays the theoretical basis for the creation of an informative system consulted by casual or non-expert users.

On different points, we compare the performances of our system with those of some related ones, and outline the possible extensions.

Logic Control With Logic

Luís Moniz Pereira

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

Methods are presented for controlling top-down executions with backtracking of logic programs, that rely on writing interpreters in logic to perform that control. This has advantages of clearness, modularity and adaptiveness.

Many examples are given. The Epilog language is introduced.

The Semantics of Parallelism and co-Routining in Logic Programming

L. Moniz Pereira and *L. F. Monteiro*

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica — Portugal

We begin with an introduction to a simple but powerful logic programming language called Prolog, in order to provide a rigorous context of presentation of ideas and results.

Next we present definitions of sequential, parallel and co-routined executions of programs, in strictly logic programming terms, and go on to define in logic a parallel interpreter for logic programs, obtained by a simple program transformation from a purely sequential interpreter. We then show how similar transformations may be directly applied to programs to obtain transforms that achieve parallelism or

co-routining without recourse to special interpreters. Afterwards, we apply our results to data base lookup and to problems arising from the use of negation as nonderivability, and suggest the basis of a rudimentary control language for logic programs.

Included in "Mathematical Logic in Computer Science", North-Holland 1981.

Knowledge Engineering Techniques and Tools for Expert Systems

René Reboh

Software Systems Research Center
Linköping University, S-58183 Linköping, Sweden
(dissertation)

Techniques and tools to assist in several phases of the knowledge-engineering process for developing an expert system are explored.

A sophisticated domain-independent network editor is described that uses knowledge about the representation and computational formalisms of the host consultation system to watch over the knowledge-engineering process and to give the knowledge engineer a convenient environment for developing, debugging, and maintaining the knowledge base.

We also illustrate how partial matching techniques can assist in maintaining the consistency of the knowledge base (in form and content) as it grows, and can support a variety of features that will enhance the interaction between the system and the user and make a knowledge-based consultation system behave more intelligently.

Although these techniques and features are illustrated in terms of the Prospector environment it will be clear to the reader how these techniques can be applied in other environments.

A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation

Henryk Jan Komorowski

Software Systems Research Center,
Linköping University
S-581 83 Linköping, Sweden
(dissertation)

We investigate partial evaluation of Prolog programs as a part of a theory of interactive, incremental programming. The goal of of this investigation is to provide formally correct, interactive programming tools for program transformation.

An abstract Prolog machine is introduced. The machine is systematically extended to an abstract partial evaluation Prolog machine. Three fundamental partial evaluation transformations are introduced and proved to preserve meaning of programs: pruning, forward data structure propagation, and opening (which also provides backward data structure propagation). The theoretical investigation is then extended to account for relations between logic and partial evaluation.

An implementation of a partial evaluation system is then developed from the formal specification. The system is well integrated and efficiently implemented in the Qlog programming environment. Several examples illustrate the mechanism and applications of partial evalution.

Finally, we outline how meta-rules that control the execution of the Prolog program can be incorporated into the system in a clean way. Such rules are familiar from artificial intelligence research. They could be used in future programming environments as specialized metatheories which support the programmer in particular tasks of programming.

Logig Programming Based on a Natural Deduction System

A. Seif Haridi

UPMAIL
Computing Science Department
Uppsala University, Uppsala, Sweden
(dissertation

A novel approach to the subject of logic programming is introduced. A natural deduction system is proposed and its viability is demonstrated as the basis of the computational mechanism of logic programs instead of resolution. The immediate and significant consequence is that many logical statements which earlier have been considered as specification are now executable, in particular statements that are of non-clausal form. We develop a programming theory which allows us to study and specify computations proof--theoretically. By means of this theory we show how diverse topics and concepts of logic programming are treated and understood in a coherent way. Among the topics treated are Horn clauses and more general statements for expressing equivalence, first order functions, infinite data structures, negation and 'for-all' type of computations. All these topics are embodied in a logic programming language that is under implementation. A computational, but still proof-theoretical, substitute for unification is described and its correctness is proved. It is an algorithm that generates a certain kind of proofin our system. We associate computation rules with the statements expressed in our language to provide control information for running programs effficiently. These rules are general enough to widen the scope of the operational understanding of a logic program statement from Kowalski's procedural reading to the concept of co-operating agents (processes) working on infinite data structures.

Program Transformation by Data Structure Mapping

Ake Hansson and *Sten-Åke Tärnlund*

UPMAIL
Computing Science Department
Uppsala University, Uppsala, Sweden

Suppose that we have a program P and a data structure d and we want to arrive at an isomorphic program P' that, for example, is more efficient or more didactic than P. A well known method to solve this problem intuitively is to substitute a new data structure d' for d and to manipulate P so we get an equivalent program P'. A nice example of this idea is the informal development of the heap sort algorithm on binary trees to an algorithm on arrays.

Our purpose is to present a formalization of a 1–1 function between lists and d-lists so we can develop an isomorphic program on d-lists as data structure from a program on lists by a formal derivation e.g., in a first order natural deduction system. The success of this method is dependent on whether or not there are dextrous mappings between data structures, and the merit of a mapping function may be reflected by the length (number of steps) in such a derivation.

Program transformation can be viewed as a special case of program synthesis when the

abstracts

specification is a program, but it can also be used as an auxiliary method for deriving programs from abstract specifications (abstract in the sense that no data structure is specified, and moreover, several programs can be derived).

Properties of a Logic Programming Language

Hansson, A.; Haridi, S., Tärnlund S.-Å.

UPMAIL
Computing Science Department,
Uppsala University, Uppsala, Sweden

We have developed a logic programming system based on natural deduction. It consists of a class of statements which is a superclass of Horn clauses. We can run logical statements that earlier have been considered as specifications. For example, the language contains the logical constants negation, equivalence, universal quantifier and identity that give the notions of definitions of functions and relations, infinite data structures and virtual classes. Computation rules provide control information for running programs efficiently and, for example, we have the concept of cooperating processes giving us computations on infinite data structures that terminate.

A Programming Language Based on a Natural Deduction System

Sten-Åke Tärnlund

UPMAIL
Computing Science Department
Uppsala University, Uppsala, Sweden

We shall take up a programming language based on natural deduction. Our presentation starts with the subset of Horn clause logic on which Prolog is based. It continues with inference rules that provide a programming language on full predicate logic. For example, we have negation at the object level and virtual classes. Identity is added to the language and this provides us with the notion of a function. Moreover, identity makes a unification algorithm redundant. The ideas of evaluated and non-evaluated terms as well as infinite objects are introduced. A few simple computation rules are discussed, in particular control on the inference rule 'and-introduction', from which we may obtain several well-known computation rules, e.g. coroutining and lazy evaluation. Furthermore we treat computations on infinite data stuctures and prove that they terminate. Finally, we treat computations on negated statements.

(Re)Implementing Prolog in Lisp or YAQ - Yet Another QLOG

Mats Carlsson

UPMAIL
Computing Science Department
Uppsala University, Uppsala, Sweden

The report describes an embedding of Prolog into a portable Lisp programming environment. The syntax, data types, I/O, and debugging tools are inherited from the host language. Space saving aspects of the implementation are discussed. There is an optional occur checker and a novel feature called variable arity.

Deductive Modeling of Human Cognition

Goran Hagërt

Department of Psychology
Uppsala University, Sweden

Sten-'Ake Tärnlund

UPMAIL
Computing Science Department
Uppsala University, Uppsala, Sweden

Deductive analysis (DA) is presented as an approach to the study and simulation of human cognitive processes. DA is composed of a psychological theory and a methodology that can shed light on mental phenomena. The cognitive theory is embedded in the problem space and the control structure hypotheses. The methodology consists of logical derivations of computer models from an abstract specification. The item--recognition task and the three-term series task are analyzed for purpose of illustration. Several aspects of human cognition in these environments are discussed. It is argued that DA brings new notions to the study of human cognition, for instance, to design sets of models and to distinguish between empirically equivalent models.

Natded, a Derivation Editor

Agneta Eriksson and *Anna-Lena Johansson*

UPMAIL
Computing Science Department
Uppsala University, Uppsala, Sweden

We introduce the derivation editor, NATDED, by taking up McCarthy's challenge that a program for concatenating lists is associative. That is to say, given three lists X, Y, and Z, concatenating Y to X and concatenating Z to this result gives the same list as concatenating Z to Y and concatenating this result to X. The first step to prove this property is of course to characterize it formally. The second step is to prove the theorem and we would like to get a formal proof that keeps the structure of a simpler informal argument. The example helps us also to illustrate this, and we shall especially focus on two properties of the derivation editor: first, we illustrate the possibility to get structures of proofs; and next, the potential to write composite rules. Finally we comment on the implementation.

A PROLOG Implementation of Query-by-Example

J. Neves

Department of Computer Science
Universidade do Minho, Braga, Portugal

R. C. Backhouse and *S. O. Anderson*

Department of Computer Science
Heriot-Watt University, Edinburgh EH1 2HJ, UK

Data bases can be conveniently represented as a set of clauses in a subset of the predicate calculus known as Horn clausal form. This form of predicate calculus has been implemented as the programming language Prolog (Programming in Logic). Query-by--Example is a simple interface to relational data bases which requires the user to formulate a query by filling in tables of example results.

The similarity between QBE syntax and Prolog goals has been noted in the literature.

abstracts

This paper realises that correspondence by providing a translator from QBE to Prolog goals. Our conclusion is that Prolog is a good tool for implementing relational data bases.

Issues such as the inclusion of general rules in Prolog data bases and how to express them in QBE, and the interpretation of *not* in Prolog suggest that with more work the marriage of QBE and Prolog will result in a more powerful and accessible data base system than either provide separately.

Descriptions and Qualifiers

Mark Wallace

The Computer Studies Group
The University
Southampton, S09 5NH, UK

Natural language conveys information by *referring* to things and asserting how they are *related*. We can parse sentences into 'descriptions' and 'qualifiers' which have an analagous semantics. Descriptions can represent names ("Fred"); determiners and numbers ("the 2 party leaders", "anyone"); phrases with embedded clauses ("the man who came to dinner"); and natural language functions ("*both parents'* of each pupil"). We discuss the formal query language of descriptions and qualifiers with reference to the effect of determiners on meaning and database searching. We report on an implemented natural language system based on descriptions and qualifiers.

Restriction Grammar in Prolog

Lynette Hirschman and *Karl Puder*

Research and Development Activity
Federal and Special Systems Group
Burroughs Corporation
Paoli, Pensylvania 19301, USA

This paper describes the implementation of Restriction Grammar in DEC-10 Prolog. Restriction Grammar (RG) is derived from a style of grammar developed at the Linguistic String Project of New York University; it consists of a set of context-free BNF definitions, augmented by grammatical constraints or *restrictions*. During parsing, a tree is automatically generated to represent the context-free rules that have been applied. A special circular data structure for tree terms implements multiple links between each node and its parent, its first child, and its left and right siblings, allowing free traversal of the tree. The restrictions access the necessary contextual information by climbing through the tree and testing its structure, rather than by parameter passing, as in DCG's. An interpreter hides from the user the parameters for tree construction, in addition to those for the input and output word stream. The result is a highly inspectable, easily modifiable grammar which transfers many of the book-keeping tasks (e.g., parse tree generation, parameter passing) from the user to the RG interpreter or translator.

Bi-Directional Inference

João P. Martins, Donald P. McKay and *Stuart C, Shapiro*

Department of Computer Science
State University of New York at Buffalo
Amherst, NY 14226, U.S.A.

We present an overview of SNIP, the SNePS Inference Package, and discuss the interaction between forward and backward inference. Such interaction is called bi-directional inference and corresponds to a bi-directional search. Bi-directional inference sets up a conversational context and focuses a system's attention towards the interests of the user, cutting down the fan out of pure forward or pure backward chaining. We show an example of bi-directional inference and compare the results obtained using such inference with the results obtained using backward or forward inference only.

SNePSLOG User's Manual

Donald P. McKay and *João Martins*

Department of Computer Science
State University of New York at Buffalo
Amherst, NY 14226, U.S.A.

SNePSLOG is a logic programming interface to SNePS. It uses SNAlculus, the basic predicate calculus augmented with SNePS logical connectives. The system consists of two ATN grammars, one for parsing and one for generation. In addition, there is a toplevel READ-EVAL-PRINT loop which takes the place of the SNePS toplevel. The SNePSLOG language is described by a context free grammar.

A Prolog Implementation of a Large System on a Small Machine

Luís Moniz Pereira
António Porto

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

This paper describes a natural-language question-answering system for aiding in the planning of research investment which was completely written in Prolog (database included) and runs on a microcomputer.

We emphasize the techniques employed to get such a system to run on a small machine.

Towards a Logical Reconstruction of Relational Database Theory

Raymond Reiter

Department of Computer Science
Rutgers University
New Brunswick, NJ08903, USA

Insofar as database theory can be said to owe a debt to logic, the currency on loan is *model theoretic* in the sense that a database can be viewed as a particular kind of first order interpretation, and query evaluation is a process of truth functional evaluation of first order formulae with respect to this interpretation. It is this model theoretic paradigm which leads, for example, to many valued propositional logics for databases with null values.

In this paper I argue that a *proof theoretic* view of databases is possible, and indeed is much more fruitful. Specifically, I show how relational databases can be seen as special theories of first order logic, namely theories incorporating the following assumptions:

1. The domain closure assumption: The individuals occurring in the database are all and only the existing individuals.
2. The unique name assumption: Individuals with distinct names are distinct.

3. The closed world assumption: The only possible instances of a relation are those implied by the database.

It will follow that a proof theoretic paradigm for relational databases provides a correct treatment of

1. Query evaluation for databases with incomplete information, including null values.
2. Integrity constraints and their enforcement.
3. Conceptual modelling and the extension of the relational model to incorporate more real world semantics.

Associative Evaluation of PROLOG Programs

Katsuhiko Nakamura

Tokyo Denki Univ.
Department of Systems Engineering
Hatoyama, Saitama 350-03, Japan

An evaluation method for PROLOG programs is represented, which is employed in the H-PROLOG interpreter. In this method, hash memories are used to store several kinds of information for the purpose of high speed access and efficient comparison of data. The main working storage is a hash memory and contains variable-value (term) pairs called bindings. The notion of binding is extended so that it is referred by its variable name and a label called a context which is generated at each application of a clause (a procedure). A binding contains another context to determine whether it is "alive" or "dead". Another hash memory contains the "monocopy" lists which represent subterms in a program and the indices of clauses. The system written in the *C* language is simple because of employment of the data structures based on the hash techniques and of LISP functions.

A Prolog Implementation of the Knuth-Bendix Reduction System

Armando Matos

Departamento de Engenharia Electrotécnica
Universidade do Porto
Rua dos Bragas, 4099 Porto Codex, Portugal

I have developed a Prolog program implementing the Knuth-Bendix Reduction System ([1]); most of the examples in [1], notably the most difficult one (example 16) were tried and verified (in a couple of cases with apparently shorter sequences of intermediate axioms).

Interestingly enough, the program runs in a small mini-computer (PDP 11-03) using the Edinburgh RT-11 interpreter with similar execution times to those reported in [1].

Two aspects of programming deserve mention: in the first place, if terms of the theory are represented as Prolog terms and the "built-in" Prolog unification is used (as I have done) care should be taken due to the absence of the occur check in most Prolog implementations. In the second place, the candidate selection for critical pairs requires some form of heuristic control (I have used a complexity measure) so that powerful axioms are generated as early as possible.

It would be interesting to run this example on a bigger machine (using a compiler) and to program more general reduction systems (see [2]).

A detailed report of this work is available from the author.

[1] Knuth; Bendix — Simple Word Problems in Universal Algebras — in Computational Problems in Abstract Algebras. Ed. J. Leech, Pergammon Press, pp. 263-267, 1970.

[2] Peterson; Stickel — Complete Sets of Reductions for some equational theories. JACM, V.28 No. 2, pp. 233-264, 1981.

A Logical Approach to Simulation

Iván Futó

Institute for Coordination
of Computer Techniques
Budapest H-1368 POB. 224, Hungary

Tamás Gergely

Research Institute for Applied
Computer Science
Budapest H-1536, POB 227, Hungary

Computer simulation plays an important role within the solution of those problems which are connected with analysis or synthesis of objects of high complexity. The main characteristics of simulation models and their development are analysed. In order to support the development a consistent family of formal notions are briefly introduced within the frame of mathematical logic. The theory thus obtained is called simulation logic. Declarative and procedural aspects of simulation models can be handled in a unique way by the use of a constructive part of the first order classical logic. Horn formulas which were the basis of logic programming now become the basis for logic simulation. TS-PROLOG, the simulation language of logic simulation is developed and its usage illustrated by modelling a decentralised control system worked out in detail.

A Discrete Simulation System Based on Artificial Intelligence Methods

Iván Futó and *János Szeredi*

Institute for Coordination of Computer
Techniques
Budapest, Hungary

A discrete event simulation system based on the AI language PROLOG is presented. The system called T-PROLOG extends the traditional possibilities of simulation languages toward automatic problem solving by using backtrack in time and automatic model modification depending on logical deductions. As T-PROLOG is an interactive tool, the user has the possibility to interrupt the simulation run to modify the model or to force it to return to a previous state for trying new possible alternatives. It admits the construction of goal-oriented or goal-seeking models with variable structure. Models are defined in a restricted version of the first order predicate calculus using Horn clauses.

To appear in "Discrete Simulation and Related Fields" ed. A. Javor, North-Holland Amsterdam, 1982, pp. 135-150.

LDM — A Program Specification Support System

Zs. Farkas, P. Szeredi, E. Sántáné-Tóth

Institute for Coordination of Computer
Techniques (SzKI)
Akadémia utca 17.
H-1054 Budapest, Hungary

LDM is a software development method based on the ideas of logic programming and

abstracts

the Vienna Development Method. The LDM language can be considered as a constructivity preserving extension of PROLOG with notions useful for describing programs; these extra notions originate mainly from VDM. This paper deals with the use of LDM in the specification and design phase of program development. An interactive support system is introduced, which helps formulating specifications in LDM and checking them by executions (which is made possible by constructivity). Besides listing the commands of the system an example is shown to illustrate not only the LDM language but the use of the system as well. Finally, the realization of the system as a PROLOG program is outlined.

First International Logic Programming Conference, Marseilles, France, September 1982.

Module Concepts for Prolog

Peter Szeredi

Institute for Coordination of Computer Techniques (SzKI)
H-1368 Budapest, POB 224, Hungary

We outline the problems of introducing modularity to the PROLOG language, overview three present implementations featuring some level of modularity, and discuss the consequences of the modularity models introduced.

Presented at the Workshop on "PROLOG Programming Environments" Linköping, Sweden, 24-26 March 1982.

The MPROLOG Programming Environment: Today and Tomorrow

Péter Köves

Institute for Coordination of Computer Techniques (SzKI)
Akadémia utca 17.
H-1054 Budapest, Hungary

The state of the MPROLOG system, as of february 1982, is described. The program development subsystem (PDSS) is introduced. The services of the envisaged programming laboratory are briefly described.

MPROLOG is a modular PROLOG system developed by the Institute for Coordination of Computer Techniques (SzKI). The system was designed to support modular programming in PROLOG and efficient program development and execution. One of the most important design criteria was that the system itself should be as highly portable as possible.

The MPROLOG system currently consists of four components: the pretranslator, the consolidator, the interpreter and the program development subsystem (PDSS).

The paper describes the current state of the system. It then goes on to outline some plans for future development. These include the completion of a compiler by the end of 1982 and the upgrading of the PDSS into the MPPL: the MPROLOG Programming Laboratory. The latter incorporates a source level global optimizer (including intermodule optimization), a full-scale focussing mechanism for syntax-driven editing, and support of group work through a dedicated software data base.

STOP PRESS

A new version of the "User's Guide DEC system-10 Prolog" is available, authored by D. Bowen, and can be obtained from the Department of Artificial Intelligence at Edinburgh.

research centres addresses

Here are some more:

Department of Computer Science
The University of British Columbia
2075 Wesbrook Mall
Vancouver B.C. V6T 1W5 Canada

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

INRIA
Domaine de Voluceau — Rocquencourt
BP 105
78150 Le Chesnay France

SZAMKI
P.O.B. 227
Budapest H-1536 Hungary

SZTAKI
P.O.B. 63
Budapest H-1502 Hungary

SZAMOK
P.O.B. 146
Budapest H-1502 Hungary

NIM IGUSZI
P.O.B. 33
Budapest H-1363 Hungary

Dept. of Computer Science
Trinity College
201 Pearse Street
Dublin 2 Ireland

I.E.I-C.N.R
Via S. Maria 46
I-56100 Pisa Italy

Electrotechnical Laboratory
Sakura-Mura, Niihari-Gun
Ibaraki Japan

Software System Research Center
Linköping University
S-581 83 Linkoping Sweden

Computer Research Center
Hewlett Packard Laboratories
1501 Page Mill Road
Palo Alto CA 94304 USA

IBM Research Laboratory
5600 Cottle Road
San Jose CA 94143 USA

Dept. of E.E.C.S.
University of Santa Clara
Santa Clara CA 95053 USA

Logicon — Operating Systems
21031 Ventura Blvd
Woodland Hills CA 91364 USA

Dept. of Computer Science
Duke University
Durham NC 27706 USA

Dept. of Computer Science
Rutgers University
New Brunswick NJ 08903 USA

RADS/ISIS
Griffis A.F.B. NY 13341 USA

IBM Systems Research Institute
205 East 42nd Street
New York NY 10017 USA

Dept. of Computer Science
University of Texas at Austin
Austin TX 78712 USA

Dept. of Computer Science & Informatics
Josef Stefan Institute
University of Edvard Kardelj
Ljubljana Yugoslavia