LOGIC PROGRAMMING WORKSHOP

14-16 Juli 1980

LOGIC PROGRAMMING WORKSHOP

14 - 16 July

1980

TABLE OF CONTENTS

AUTHOR INDEX

AN IMPROVED PROLOG IMPLEMENTATION WHICH OPTIMISES TAIL RECURSION

David H D Warren
Department of Artificial Intelligence
University of Edinburgh

1. ABSTRACT

This paper describes some recent improvements to the DEC-10 implementation of Prolog, concentrating on one of them - "tail recursion" optimisation. We argue that this optimisation should be included as a matter of course in any new Prolog implementation, and give a detailed account of what is involved.

2. INTRODUCTION

The purpose of this paper is to report on some improvements recently made to DEC-10 Prolog [3]. Familiarity with the Prolog language [5] [9] [8] is assumed, and for a full understanding of the more technical parts of this paper, the reader should refer to [7] for details of the original DEC-10 implementation. Other useful accounts of Prolog implementation include [1] [4] [2].

From the user's point of view, perhaps the most important of the new improvements are those which make the system more convenient to use. These are an extended interpreter with better debugging aids produced by Lawrence Byrd [to be reported on separately], and an "in-core" compiler.

The in-core compiler is an integral part of the Prolog system which enables Prolog modules to be compiled directly into the same address space, ready for immediate execution. This is in contrast to the original stand-alone compiler, which was of the more usual kind where modules have to be separately compiled into files of relocatable code, needing to be passed through a linkage editor to produce a runnable program. Like most such compilers, it was very tedious to use for interactive program development. With the in-core compiler, compiling a module is just as easy for the user as reading it in to the interpreter. Compilation has the advantage that the resulting code runs 10 or 20 times faster and requires much less working storage, but the disadvantages that the compilation itself is several times slower and most of the debugging aids, such as tracing, are no longer applicable.

The new system operates as two swappable segments containing respectively the interpreter and the compiler, so the store occupied is not a great deal more than before. My only further comment on the in-core compiler is that it took less effort to produce than I expected!

The remainder of this paper concentrates on one other major improvement - tail recursion optimisation (TRO). A preliminary report on this improvement appeared in [8]. At that time I had designed and partially implemented a completely new Prolog implementation, incorporating TRO and other associated improvements. It later became obvious that it would take a lot of work to bring this implementation to practical fruition. In particular, it was not clear whether the associated improvements could be made compatible with the needs of garbage collection. As I shall argue below, the full potential of TRO is only realised in conjunction with a garbage collector. I therefore abandoned the idea of reimplementing from scratch, and instead pursued the less ambitious course of making the minimum changes to the existing DEC-10 system necessary to support TRO. This entailed much less work, but has meant that not all the potential of

TRO has been realised, particularly as regards speed.

3. OVERVIEW OF TAIL RECURSION OPTIMISATION

Like most high-level languages, Prolog requires a stack (called the local stack) to hold frames, one for each active procedure, containing bookkeeping information together with the value cells for local variables. For most languages, the stack frame can be discarded at the time the procedure returns its result. With Prolog, however, this is not in general possible, since procedures may be non-determinate, ie. they can return several alternative results. The stack frame is therefore not generally reclaimed until backtracking occurs, after the procedure has generated all of its results. However, if the Prolog system can detect that it is generating the last alternative result of a procedure, it is possible to reclaim the stack frame immediately on return from the procedure, as in a conventional language, and many Prolog systems (including DEC-10 Prolog) do this.

Detection of the determinate situation is therefore quite important, and normally requires either that the procedure be appropriately indexed yielding only one candidate clause for matching, or that determinacy be signalled or imposed by the programmer through Prolog's "cut" operator. (Hence much of the importance both of indexing and of cut).

The additional improvement of tail recursion optimisation rests on the observation that one does not need to wait until a determinate procedure returns in order to reclaim its stack frame. Instead it is possible to recover the stack frame at the time the last goal in the procedure is invoked. ie. If the Prolog system has reached the last goal in a clause, and there are no remaining backtrack points within the procedure to which that clause belongs, then the current stack frame can be overwritten by the stack frame for the procedure about to be invoked. (Note that it is not necessary for the last goal to be a recursive call; a better name for the improvement would be "last call" optimisation).

To get away with this rather underhand manoeuvre, it is obviously essential to extract from the old stack frame all information that will be needed subsequently. In particular it is vital not to leave behind any pointers to the old information. Hence the following departures from previous practice in Prolog implementation.

1. Instead of information about the caller, a procedure is now passed a continuation, consisting of a pointer to the actual goal to be executed next, together with its associated stack frame. Thus the continuation does not necessarily correspond to the immediate parent procedure, but in fact indicates the most recent ancestor with further goals left to execute.

2. A called procedure is no longer able to access its arguments merely by referring to the information about its caller (since the caller's stack frame may well have been discarded). Instead the arguments of a call have to be copied into registers, to be subsequently stored in the callee's stack frame as extra bookkeeping information looking exactly like ordinary (local) variable cells. This scheme has the incidental advantage that certain unification steps become null operations, namely those steps concerning unification with the first occurrence of a variable at the outermost level in the head of a clause where that variable is local (ie. it does not occur in a compound term). The value cell for such a variable can be identified with the location for the corresponding procedure argument, and so

unification does not need to initialise it.

3. A final pitfall which has to be avoided concerns the case where an argument to a procedure is (a reference to) an uninstantiated variable in the stack frame about to be discarded. Possible solutions to this problem, including the one actually adopted, are discussed in a later section.

So much for what the TRO is, what are its benefits?

The most obvious benefit is a saving of (local) stack space. For example, the procedure "quicksort" now only requires a stack of size order log N instead of order N. And a determinate, directly tail recursive procedure such as "concatenate" now never uses more than one stack frame. However this benefit alone is not as significant as it might seem. Firstly, the TRO is only recovering earlier space which would anyway be recovered later. Secondly, we have ignored the fact that most Prolog procedures create new structures which cannot be stored on the local stack. So the effect on total working storage requirements (which is all the user is aware of) is unlikely to be dramatic.

For the space saving to really pay off, it is necessary to have a garbage collector which can recover the space occupied by structures which are no longer capable of being referred to. Such a garbage collector was already included in DEC-10 Prolog. The real worth of the TRO is then that, not only does it recover local stack frames, but also it allows structures which are only accessible through such stack frames to be garbage collected. A determinate tail recursive procedure can therefore potentially continue executing indefinitely, even though it creates new structures. A typical example would look like:-

```
cycle(S) :- modify(S,S1), cycle(S1).
```

where 'modify' is a determinate procedure which transforms a structure S into a new structure S1. The structure might be a term representing the state of a database, or a term representing the conversation so far in a natural language question answering system, for example.

This ability of the TRO in conjunction with garbage collection to make certain kinds of Prolog program feasible for the first time is, I think, the main argument for introducing it.

A further benefit of the TRO, which particularly appeals to me, is that it also saves time, although it could be argued that the saving is not, by itself, of major significance in practice. The saving arises because certain bookkeeping operations can be coalesced (so fewer such actions occur during an execution), and moreover in certain cases much work can be saved in the process. In particular, when one stack frame overwrites another, part of the bookkeeping information can be retained intact, and where a clause contains no more than one goal in its body, bookkeeping information can be kept in registers without the need for saving and restoring. The net effect is to remove most of the efficiency overheads of recursive procedures with respect to corresponding iterative loops in a conventional language. There is also the saving, already mentioned, of certain unification steps, which can be particularly significant in procedures where many clauses have to be examined for a possible match.

The actual speed improvement achieved when the TRO was incorporated in our DEC-10 implementation ranges from 6% for examples of non-determinate procedures to 56% for 'concatenate' and 68% for 'length' (of a list). Note that this makes the speed of Prolog 'concatenate' almost identical to that of the corresponding Lisp version compiled with the DEC-10 Stanford Lisp

compiler (assuming no change in the latter speed since 1977). The important special case of a unit clause had effectively already been optimised in the original implementation, so the improvements achieved in the best cases were not as typical of the general situation as had been hoped for.

TRO is, of course, not a new idea (cf. for example SCHEME [6]), although it seems that it has seldom been incorporated in software in widespread use. The reason is probably that, because most languages include explicit iterative constructs, implementors feel the burden can be left on the programmer to recognise the iterative situation. However this state of affairs seems hard to justify, since TRO is relatively easy to implement (see below).

It is important to notice that TRO is more widely applicable in Prolog than in other languages such as Lisp. For example the Prolog procedure for 'concatenate':-

```
concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

is susceptible to TRO, giving essentially the following iterative version:-

```
concatenate(L1,L2,L3) =
  (while L1 is a non-empty list
   do
      let List be a new record with 2 fields;
      head(List) := head(L1);
      L1 := tail(L1);
      field pointed to by L3 := List;
      L3 := a pointer to tail(List)
   repeat;
   field pointed to by L3 := L2;
   return)
```

However for the Lisp version:-

```
concatenate(L1,L2) =
  (null(L1) -> L2,
   T -> cons(car(L1),concatenate(tail(L1),L2)))
```

straightforward TRO does not yield an iterative version, since the last function call to be executed is the call to 'cons', not the call to 'concatenate'. It will be seen that the iterative version requires the handling of partially completed structures and the passing around of pointers to the corresponding "holes". This is available as a matter of course in a Prolog implementation, since it is a feature of the language (the "logical variable"), but the same is not true of Lisp. Now because, in cases like this, the iterative version can only be programmed using the very low-level concepts of pointers and pointer assignments, it seems even more unreasonable for the job to be left to the programmer rather than the implementation.

The following section gives a detailed description of what is involved in implementing the TRO. The reader familiar with the standard implementation will see that TRO introduces very little extra complexity, provided it is designed in from the outset. For this reason, I think the TRO should be included as a matter of course in any new Prolog implementation, even if, without the inclusion of a garbage collector, the full potential is not realised.

4. DESIGN DETAILS FOR TAIL RECURSION OPTIMISATION

This account aims to be as self-contained as possible, but the reader should refer to [7] for certain details. Note that the version actually implemented differs slightly from the more ideal design given here, because of constraints imposed by adapting the existing implementation.

4.1. Data Areas

The main data areas are the code area, containing data representing the program itself, and three areas operated as stacks, the local stack, the global stack and the trail. Each procedure invocation leads to the creation of an environment comprising three stack frames, one on each stack. The local frame contains information that is only required during the execution of the procedure concerned, namely bookkeeping information and the value cells for local variables. The global frame contains information representing the new structures (complex terms) created by the procedure. For structure-sharing implementations, this will comprise just the value cells for global variables. The trail frame contains the addresses of variable cells which have been assigned to during unification and which must be reset to "unassigned" on backtracking.

4.2. Registers

The current state of a Prolog computation is defined by certain registers containing pointers into the main data areas. These registers are:-

L	latest local frame	(V)
G	latest global frame	(V1)
TR	top of trail	(TR)
CP	continuation point (goals to be executed next)	(A)
CL	continuation local frame	(X)
CG	continuation global frame	(X1)
BP	backtrack point (alternative clauses)	(FL)
BL	backtrack local frame	(VV)
BG	backtrack global frame	(VV1)

where the names in brackets are those used in [7] for roughly corresponding registers. In addition, there are registers:-

A1, A2, ... etc.

representing the arguments of a procedure call. These registers contain constructed terms, ie. representations of atoms, compound terms or references to variable cells.

4.3. Bookkeeping Items

Each local frame contains space for six items of bookkeeeping information, referred to as:-

CP(l)	CL(l)
G(l)	TR(l)
BP(l)	BL(l)

where l is the address of the local frame. These items are the values of the corresponding registers at the time the procedure was invoked. The items TR, BP and BL are only needed if the procedure is a choice point.

4.4. Shadows

Certain registers are not strictly essential, since they merely "shadow" certain stack locations:-

```
G = G(L)          CP = CP(L)
CG = G(CL)        CL = CL(L)
BG = G(BL)
```

However the use of these non-essential registers is likely to be more efficient. (Certainly this is the case in the DEC-10 implementation). In particular, BG is involved in checking whether a variable assignment needs to be trailed.

4.5. Procedure Arguments

The local frame also contains n locations (where n is the arity of the procedure) into which the procedure's arguments are stored. These locations are called:-

```
A1(l), A2(l), ... etc.
```

where l is the address of the local frame. These locations look just like ordinary variable cells, except that the value of the cell cannot be "unassigned".

4.6. Analysis of a Prolog Program into Basic Operations

Let us now analyse the procedural aspect of a Prolog program into some basic operations which allow for the tail recursion optimisation. The detailed implementation of each basic operation will then be described later. The naive analysis of a general clause:-

```
P :- Q, R, S.
```

would be:-

```
match P
enter
call Q
call R
call S
exit
```

However to allow for the tail recursion optimisation, the last call must be treated differently; the actual analysis is:-

```
match P
enter
call Q
call R
depart S
```

Thus, in effect, "depart = call + exit". Clauses with no goal in the body (a unit clause), or with just one goal in the body (a doublet clause), are treated as special cases:-

```
match P          match P
return           proceed Q
```

One can summarise this as "return = enter + exit" and "proceed = enter + depart".

The operations corresponding to a procedure P comprising clauses C1, C2, C3 are:-

```
arrive P
choice
try C1
try C2
nochoice
```

```
        trust C3
```

If there is just one clause, the operations reduce to merely:-

```
        arrive P
        trust C1
```

Note that if the procedure is indexed, instead of just one sequence of candidate clauses, there will be a set of alternative subsequences, one for each different key; for keys with just one candidate (a common case), only a single trust action will be necessary, as in the second of the two alternatives above.

4.7. Description of the Basic Operations

The details of each basic operation, apart from the unification operation 'match', are now described. Two other basic operations are also covered. These are 'fail', the backtracking operation which occurs when unification fails, and 'cut', the operation corresponding to the Prolog control primitive.

```
call
    {Load arguments and invoke procedure with new continuation.}
    load A1,... with arguments from CL and CG;
    CP := remaining goals;
    goto procedure

enter
    {Complete the current local and global frames.}
    save CP, CL, G into L;
    CL := L;  CG := G;
    L := L + size of local frame;
    check L is not full;
    G := G + size of global frame;
    check G is not full

exit
    {Resume at continuation, discarding the local frame if the current
    procedure is determinate.}
    if BL < CL then L := CL;
    restore CP, CL from CL;
    CG := G(CL);
    goto CP

depart
    {cf. "call + exit". Load arguments, and invoke the new procedure with
    the old continuation, overwriting the local frame if current procedure
    is determinate.}
    load A1,... with arguments from CL and CG;
    if BL < CL then L := CL;
    restore CP, CL from CL;
    CG := G(CL);
    goto procedure

proceed
    {cf. "enter + depart". Load arguments, complete the local frame only
    if the current procedure is non-determinate, complete the global frame,
    and invoke the new procedure with the existing continuation.}
```

```
    load A1,... with arguments from L and G;
    if BL = L
    then
       (save CP, CL, G at L;
        L := L + size of local frame;
       check L is not full );
    G := G + size of global frame;
    check G is not full;
    goto procedure

return
    {cf. "enter + exit". Complete the local frame only if the procedure is
    non-determinate, complete the global frame, and resume at the current
    continuation.}
    if BL = L
    then
       (save CP, CL, G at L;
        L := L + size of local frame;
        check L is not full );
    G := G + size of global frame;
    check G is not full;
    goto CP

arrive
    {Store the procedure's arguments.}
    store A1,... into L

choice
    {Create backtracking point.}
    save BP, BL, TR at L;
    BL := L;  BG := G

try
    {Select a clause with other alternatives.}
    BP := other clauses;
    goto clause

nochoice
    {Remove the backtracking point.}
    restore BP, BL from L;
    BG := G(BL)

trust
    {Select a clause with no other alternatives.}
    goto clause

fail
    {Restore the state corresponding to the latest backtracking point.}
    if BL < L
    then
       (L := BL;  G := BG;
        restore CP, CL from L;
        CG := G(CL) );
    undo TR as far as TR(L);
    goto BP
```

```
cut
    {Remove backtracking points created since the current procedure was
    invoked, delete trail entries no longer relevant, and remove any local
    frames which still remain after the current one.}
    if not BL < CL
    then
       (until BL(BL) < CL do BL := BL(BL);
        restore BP, BL, TRO from BL;
        BG := G(BL);
        tidy TR as far as TRO;
        L := CL + size of local frame )
```

4.8. Design Considerations

There are certain points to notice in the above design.

1. As already mentioned, it is assumed that it is advantageous to maintain the shadow registers (CP, CL, CG, G, BG) distinct from the stack locations they shadow. In implementation environments where there is no point in this, the shadow registers can be regarded as pseudonyms for the corresponding stack locations, and much of the saving and restoring of registers can be avoided.

2. For completely determinate procedures (ie. procedure activations within which the action choice is not executed), there is absolutely no saving or restoring of the "backtracking registers" BP, BL and TR. Note that the cut operation therefore has to be slightly more expensive in the general case, since it has to trace back down the BL chain.

3. The saving of the other registers (CP, CL and G) is not performed at the beginning of the procedure, but is postponed as late as possible in the hope that it will not be necessary to preserve the local frame. It may appear that this approach is disadvantageous in the non-determinate case, since there is then the overhead of repeatedly saving the same information for each clause entered. However, in experimental comparisons of the two approaches, I have not found any example, even among very non-determinate programs, where "late saving" is slower. The reason, I think, is that the extra overhead mentioned is balanced by reduced overheads in calling a procedure where no clause matches.

4. Registers CP, CL and CG are restored by 'depart' since otherwise it would be necessary for 'return' to restore these registers. The reasoning behind this is that 'return's are more frequent than 'depart's, and again experimental evidence supports this decision.

5. In the operations 'enter', 'proceed' and 'return', if there is no global frame, then obviously there is no need either to increment the G register or to check it for global stack overflow.

4.9. Treatment of Procedure Arguments

Certain details concerning the handling of procedure arguments have not yet been discussed. In particular we have to be able to guarantee that no "dangling reference" is left to a variable cell in a discarded stack frame.

One way to do this would be to fully "dereference" all procedure arguments, and to forgo discarding the stack frame if any of the dereferenced values were a dangling reference. This has the disadvantage

that it involves quite a lot of extra work at runtime.

As an alternative, one can try to spot the possibility of a dangling reference entirely at "compile-time", and only permit the stack frame to be discarded where it can be guaranteed always safe to do so. A variation of this, the approach actually adopted, is to force any variables which might otherwise have given rise to a dangling reference to be stored in the global stack. Both these options involve little or no run-time overhead, but are less efficient at conserving stack space. I have adopted the second option because it was the simplest to incorporate in the existing implementation and because hopefully the garbage collector will be able eventually to reclaim the extra global variables.

In the implemented version, no attempt is made to fully dereference procedure arguments. If an argument is a variable, the argument register is loaded with a reference to that variable's cell, unless it can be guaranteed (at compile-time) that the variable is instantiated (to something other than a reference to its own local stack frame), in which case the value of the variable's cell is loaded. This guarantee can be made if the variable has an occurrence in the head of a clause and satisfies the normal conditions for being a local. To understand why this is so, consult [7]. The TRO then requires that a variable be deemed global if a reference to that variable is passed as an argument to the last goal in the clause. Thus the only variables made global which would otherwise have been local are those which (i) occur in the last goal, and (ii) only occur in the body. In practice this is quite a small minority of variables.

5. ACKNOWLEDGEMENTS

The improvements described were implemented with the help and encouragement of Fernando Pereira and Lawrence Byrd. The work was supported by a British Science Research Council grant.

6. REFERENCES

1. Bruynooghe M. An interpreter for predicate logic programs : Part 1. Applied Maths & Programming Division, Katholieke Univ Leuven, Belgium, 1976. Report CW 10

2. Colmerauer A, Kanoui H and van Caneghem M. Etude et realisation d'un systeme Prolog. Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1979.

3. Pereira L M, Pereira F and Warren D H D. User's Guide to DECsystem-10 Prolog. Dept of Artificial Intelligence, University of Edinburgh, 1978.

4. Roberts G M. An implementation of Prolog. Master Th., Dept of Computer Science, Univ of Waterloo, Canada, 1977.

5. Roussel P. Prolog : Manuel de Reference et d'Utilisation. Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1975.

6. Steele G L. RABBIT: A Compiler for SCHEME. Master Th., MIT, May 1978. AI-TR-474

7. Warren D H D. Implementing Prolog - compiling predicate logic programs. Dept of Artificial Intelligence, Univ of Edinburgh, 1977. Research Reports 39 & 40

8. Warren D H D. Prolog on the DECsystem-10. Expert Systems in the Micro-Electronic Age, 1979.
9. Warren D H D, Pereira L M and Pereira F. Prolog - the language and its implementation compared with Lisp. SIGPLAN/SIGART Newsletter, ACM Symposium on AI and Programming Languages, August, 1977.

The memory management of PROLOG implementations

Maurice Bruynooghe
Katholieke Universiteit Leuven
Afdeling Toegepaste Wiskunde en Programmatie
Celestijnenlaan 200A, B-3030 Heverlee, Belgium

Abstract

We describe the top down execution of logic programs and the concepts of computation rule and search rule. We show that especially PROLOG's depth first search rule results in important simplifications of the necessary runtime structures. At a high level, we describe an interpreter with its runtime structure. This implementation is probably closer to implementation techniques of Algol-like languages than to structure sharing. We claim that this high level interpreter is a fair description of the best known implementations. At the same high-level - without considering the actual representation of the bindings of variables - we discuss the different opportunities to save space by popping the environment stack of the run time structure. Finally, we discuss the problems posed to space saving by the representation of the bindings of the variables. We show that structure sharing, although superior for a general resolution theorem prover, is not the only possibility to handle the bindings of the variables.

1 Logic programs - PROLOG

Logic programs [6]

A logic program comprises a set of procedures and a goal statement. A procedure or Horn clause has the form $B \leftarrow A_1,\ldots,A_n$ $(n \geq 0)$ with B and A_i literals. Literals have the form $R(t_1,\ldots,t_k)$ $(k > 0)$ with R a k-adic relation and the t_i terms i.e. constants (first symbol an upper case letter), variables (first symbol a lower case letter) or expressions of the form $f(t_1,\ldots,t_m)$ $(m > 0)$ with f a m-ary function symbol and the t_i again terms). Such a clause can be read either as a logical fact : 'for all values of the variables, B is true if A_1 and .. and A_n are true' or as a procedure to solve the problem B : 'to solve B, solve A_1 and . . and A_n. A goal statement has the form $\leftarrow A_1,..,A_n$ $(n > 0)$ with the A_i literals It has also two ways of reading, a declarative one : 'there does not exist values for the variables such that A_1 and ... A_n are true' and a procedural one : 'find values for the variables solving the problems A_1 and .. and A_n'.

A procedure $B \leftarrow A_1,\ldots,A_n$ can be used to solve a problem when the heading B of the procedure matches the literal ('call') representing the problem. With 'match' we mean that the heading and the call must agree to consider the same problem, namely the most general instance of the call for which the procedure can be used. This matching process ('unification') creates a substitution consisting of components $x \leftarrow t$. Such a component indicates that, for agreement between call and heading, it is necessary to replace the variable x by the term t, in other words, to restrict the values for the variable x to values of the form t. Some components of the substitution can be related to the variables in the call (output) and eventually affect the remaining problems, while other components are related to variables of the procedure (input) and affect the new subproblems created by the application of this procedure (instances of the calls A_i in the body).

To execute a program, the search rule selects the initial goal statement $\leftarrow A_1,\ldots,A_n$, demands the computation rule to select a call $A_i = R(t_1,\ldots,t_p)$, applies a procedure $R(t'_1,\ldots,t'_p) \leftarrow B_1,\ldots,B_m$ matching that call with a

substitution ('most general unifier') θ and derives the new goal statement <- $(A_1,\ldots,A_{i-1},B_1,\ldots,B_m,A_{i+1},\ldots,A_n)\ \theta$. Then the search rule starts over again : selecting a goal statement, eventually activating the computation rule and deriving a new goal statement. New goal statements are active; selected goal statements become inactive once all procedures matching the selected call have been applied. The search can be represented by an OR-tree, the search tree. The nodes are goal statements. The descendants of a node are the alternative goal statements derivable from the goal statement in that node. The search is finished once all nodes are inactive. The terminal nodes of the tree are either unsolvable goal statements or empty goal statements. The empty goal statements represent solutions. The composition of the substitutions used on the path from the root node to an empty node, applied on the variables of the initial goal statement gives the desired result. To complete the search, only the active goal statements are needed.

A goal statement can be represented by an AND-tree, the proof tree. The root node has as immediate descendants the subgoals A_i of the initial goal statement <-- $A_1,\ldots,A_n$. A subgoal A_i being executed by a procedure B <-- $B_1,\ldots,B_m$ get as descendants the subgoals $B_1,\ldots,B_m$. The goal statement corresponding to a proof tree is given by the nonempty tips. The proof trees corresponding to the successive statements of a single branch in the search tree can easily be derived from each other by applying the selected procedure on the call chosen by the computation rule.

PROLOG

The search rule of PROLOG explores the search tree depth first, PROLOG always works on the most recent active goal statement (the current goal statement). This strategy greatly simplifies the needed runtime structures. Indeed, all active goal statements are on the branch leading from the root to the current goal statement. As indicated above, their proof trees can easily be derived from each other. It is possible to have only the proof tree corresponding to the current goal statement and to use backtracking to restore the others.

The computation rule of PROLOG always selects the left most subgoal in the proof tree. It means that subgoals are solved sequential, execution of a subgoal A_{i+1} is only started when the subgoal A_i is completely solved.

The subgoals are expanded from left to right; on backtracking, their expansions are removed from right to left. A node corresponding to the selected subgoal in an active goal statement is called a backtrackpoint. The nodes created by execution of the last backtrackpoint and by later steps (deterministic ones because only one procedure matches the call) are part of the current segment of the proof tree. On backtracking, the current segment is removed and what remains of the proof tree becomes the current goal statement.

2. Runtime structure interpreter

As discussed above, the run-time structure needs to represent the proof tree corresponding to the current goal statement and needs to be able to restore the proof trees corresponding to the backtrackpoints.

Part of the prooftree are the literals representing the subproblems. Usually, these literals contain variables. Before applying a procedure, it is necessary that the goal statement and the procedure use different names for their variables. A straightforward solution consists of making a copy of the procedure with unique names for the variables. This results in a inefficient

use of memory; each time a procedure is used, a new copy is made. It is preferable to use, like in Algol, reentrant code for the representation of procedures, such that all instances of the same literal share the code describing that literal. This is possible by using binding environments. The pure code is always accessed in the context of a particular binding environment. When accessing a variable in the reentrant code, the corresponding binding environment is consulted. Thus, literals can be presented by a pointer to the pure code and a pointer to the binding environment. Because all literals of the same procedure body share the same environment, it is convenient to store the binding environment in the common father node.

We can distinguisn two kinds of nodes in the prooftree. We have the tipnodes which are the unsolved subgoals of the current goal statement and the nontipnodes which are the partially solved problems. Except of the left most tipnode, the one to be selected by the computation rule (the current subgoal), all tipnodes are part of instances of procedures B <-- $B_1,\ldots,B_m$ with their first literal B_1 either as current subgoal or as nontipnode. Assuming that the pure code allows to find the right hand brothers B_{i+1}, B_{i+2}, ... of any subgoal B_i, and knowing that their execution demands the same binding-environment, an explicit representation of the tipnodes in the prooftree is not necessary, they can be accessed either through non tipnodes or through the current subgoal.

For the nontipnodes it is convenient to use a stack ('environment stack'). The subgoals being executed are pushed on the stack, on backtracking, the stack is popped.

Finally, we have to consider the unification between call and procedure heading. Besides creating a binding environment for the procedure, also the binding environment associated with the call (and, eventually other parts of the run time structure devoted to the representation of the variables) is updated. All updates not undone by popping the current segment of the environmentstack need to be saved. For this purpose it is preferable to use a stack . ('trail'). Each backtrackpoint contains a pointer to the top of the trail (the begin of a new current segment). The current segment of the trail contains all changes made to the representation of the prooftree associated with the last backtrackpoint, which are not undone by popping the current segment of the environmentstack.

The state of the computation is characterized by :
* CURR-CALL : a pointer to the pure code of the current subgoal.
* CURR-ENV : a pointer to the node containing the binding environment of the current call (the 'father' of the current call).
* CURR-PROC : a pointer to the pure code of the procedure to be applied on the current call.
* LASTBACK : a pointer to the last backtrackpoint.

The current goal statement is given by the current subgoal, its righthand brothers and the right-hand brothers of all its ancestors.

To backtrack, the current segment of the environment stack is popped (the node pointed by LASTBACK becomes the current subgoal), all updates noted on the current segment of the trail are undone and the current segment of the trail is popped.

To summarize, a 'deterministic' node contains
* CALL : a pointer to the pure code of the call.This pointer gives also access to the righthand brothers.
* FATHER : a pointer to the fatner of this node.This father node contains the binding environment associated with CALL, it also gives access to

the righthand brothers of all ancestors of CALL.
* a binding environment for the variables of the procedure applied on call.

A backtrackpoint also contains :
* BACK : a pointer to the previous backtrackpoint.
* PROCEDURE : a pointer to the pure code of the next procedure to be applied on CALL (gives access to all untried procedures).
* TRAIL : a pointer to the trail.

Algorithm

1. Push a node on the environmentstack with
   - CALL := CURR-CALL
   - FATHER := CURR-ENV
   - Find 'the first successor of CURR-PROC which possibly matches' CURR-CALL, if none then we have a deterministic node
     else we have a backtrackpoint (a bit of CALL or FATHER can be used to indicate the difference)
     and we have to complete the node with :
     BACK := LASTBACK, LASTBACK becomes the new node
     PROCEDURE := the next possibly matching procedure
     TRAIL := top of the trail
   - A binding environment for the variables of CURR-PROC
2. Unification between
   CURR-CALL with its binding environment CURR-ENV and
   the heading of CURR-PROC with its binding environment in the new node.

   All changes to the environmentstack not restricted to the current segment are noted on the trail.

   CURR-CALL := first call in the body of CURR-PROC
   CURR-ENV := the new node of the environmentstack
3. If successful unification
   then find the next unsolved subgoal (if none : a solution is derived):
   while CURR-CALL = NIL do
     CURR-CALL := the successor of CALL in CURR-ENV
     CURR-ENV := FATHER of CURR-ENV
   else backtrack : (if LASTBACK = NIL then end of computation)
   - use TRAIL pointer of LASTBACK to undo changes which are not in the current segment of the environment stack and pop current segment of the trail.
   - CURR-CALL := CALL of LASTBACK
   - CURR-ENV := FATHER of LASTBACK
   - CURR-PROC := PROCEDURE of LASTBACK
   - LASTBACK := BACK of LASTBACK and pop the current segment of the environmentstack (all nodes inclusive the one pointed by the old value of LASTBACK)

Notes
- In the first PROLOG implementation [1], [5], the idea to use reentrant code for the procedures was rather based on Boyer and Moore's structure sharing, for resolution theorem provers [2], not on implementation techniques for Algol.
- Most implementations do not make a distinction between deterministic nodes and backtrackpoints ([1], [7]) the first implementation [1] even do not keep track of the last backtrackpoint, it pops nodes one by one.
- The fact that a deterministic node is more space efficient than a backtracknode, and that deterministic nodes play a crucial role in the space saving techniques described in the next section makes it worthwhile

to spend some effort in determining 'the first procedure which possibly matches a call'. The extremes are 'the first procedure' and 'full unification between heading and call'.

3. Opportunities to pop the environmentstack - Pitfalls

a. Completing a deterministic subgoal

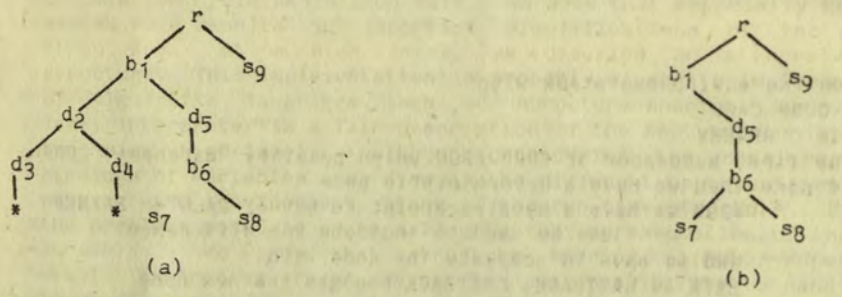


Fig. 1. Dropping a deterministic subtree

In fig. 1. r denotes the root of the proof tree, the d_i denote deterministic nodes, the b_i denote backtrackpoints, the s_i denote unsolved subgoals and the * denote empty sets of subgoals. The subgoal corresponding to d_2 is completely solved, the solution is unique (no backtrackpoints in the subtree). This completed subtree can be dropped from the proof tree (Fig. 1.b) without affecting the behaviour of the algorithm. Indeed, we have the same current goal statement and the goal statements corresponding to the backtrackpoints (b_1,b_6) are also the same.

This technique corresponds to the situation in a conventional language : the activation record is popped when returning from a procedure.

Due to the detection and handling of 'tail end recursion' (see section 3.b), the application of this technique is restricted to the situation depicted in Fig. 2. where only one procedure matches the call and the body is empty.

Detection of the situation by the interpreter (Fig. 2.b) : CURR-CALL = Nil and CURR-ENV is a deterministic node.

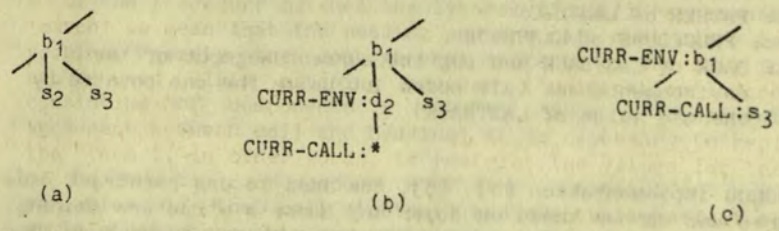


Fig. 2. A call s_2 ,only one procedure matches the call, the procedure has an empty body.

Action by the algorithm : CURR-CALL := successor of CALL in CURR-ENV, CURR-ENV := FATHER of CURR-ENV and pop the top node of the environment stack (new top is either CURR-ENV or LASTBACK).

b. Tail end recursion

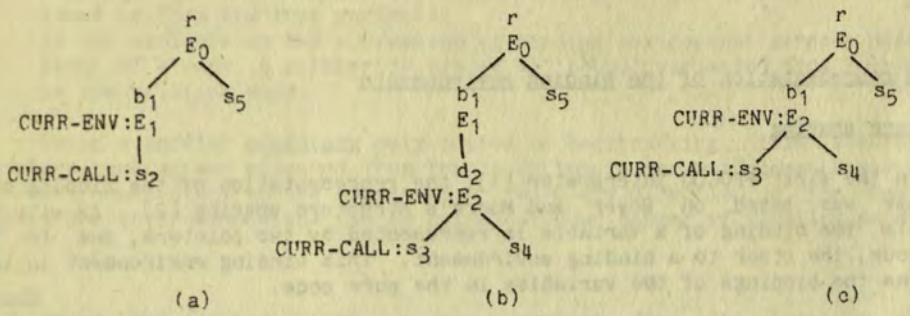


Fig. 3. Replacing tail end recursion by iteration.

The situation is depicted in Fig. 3. The E_i denote the binding environments which are part of the nodes. In Fig. 3.a, s_2 is the last call of a procedure surviving in the proof tree. (its eventually lefthand brothers have been popped). Only one procedure is matching. (Fig. 3.b). The benaviour of the interpreter on the proof trees of Fig. 3.b and Fig. 3.c is the same. Indeed : (1) the same current goal statement (with the same binding environments!) and (2) the goal statements corresponding to the backtrackpoints are also identical. The node describing the call s_2 has been collapsed with its father (E_2 replaces E_1 and the node s_2 is popped). The same node can now be used to execute s_3 (iteration over the same space). This situation is typical for recursive calls with tail end recursion. The recursion is replaced by an iteration. It results in great savings when there is deep recursion.

Detection of the situation by the interpreter : (prior to the unification. Fig. 3.a - due to the representation of the binding environments, it is preferable to swap E_1 and E_2 before unification). The successor of CURR-CALL is Nil and LASTBACK is at least as for from the top as CURR-ENV. (thus CURR-CALL deterministic and its lefthand brothers popped).

Action : prior to unification : swap binding environments of the new node and the node CURR-ENV; after unification (Fig. 3.b) : CURR-ENV := FATHER of CURR-ENV, pop the top node (CURR-ENV is the new top).

Note : to handle '/', a special feature in PROLOG which transforms some backtracknodes into deterministic nodes, it is necessary to know when either a node has real children or its children are, due to tail end recursion, further descendants. A bit of the FATHER or CALL fields can be used to indicate the difference.

c. Pitfalls

We take care that the interpreter did not need to access the popped nodes, however, up to now we did not consider the representation of the binding environments. The above reasoning is only correct when the remaining binding-environments do not refer to the removed ones : when eventual references are all oriented from top to bottom of the environmentstack. The first PROLOG interpreter [1] violated this condition, as a consequence it could only pop the stack on backtracking. The attempt to pop the stack when completing a

determinate subgoal drove the author [3], [4] and David Warren [7] to develop other methods for the handling of the binding environments. The author was probably the first to consider also 'tail end recursion' (implementation : end 1977).

4. The representation of the binding environments

Structure sharing

In the first PROLOG interpreter [1], the representation of the binding of a variable was based on Boyer and Moore's structure sharing [2]. As with the subgoals, the binding of a variable is represented by two pointers, one to the pure code, the other to a binding environment. This binding environment in turn contains the bindings of the variables in the pure code.

With this schema, the unification algorithm can be confronted with three basic situations where it has to bind a variable.

1. A variable x which is free in environment E_i and a variable y which is free in environment E_j
   Suppose that E_i is more recent (or identical) than E_j, then, the variable x in E_i is bound to the pure code of y and to the environment E_j. The pointer from E_i to E_j is oriented to the bottom of the environmentstack.
2. A variable x which is free in environment E_i and a term t with environment E_j and E_i more recent than E_j. x is bound to the pure code of t and to the environment E_j. Also this pointer is oriented to the bottom of the stack.
3. Same as (2) but E_j is more recent than E_i.
   Also here, x has to be bound to the pure code of t and the environment E_j but the pointer from E_i to E_j is oriented to the top of the environmentstack. As a consequence, the space saving techniques of the previous section are impossible.

To solve the problem, Warren [7] observes that the trouble is due to the variables occurring in the terms of the pure code. He calls such variables global, the others he calls local. He divides the binding environment into a global part and a local part. Also in the pure code, he makes a distinction between global and local variables. Local environments are placed on the environmentstack but the global ones are placed on a special global stack. This stack is only popped on backtracking. Now unification starts with two literals of pure code, each with a local and global environment. The algorithm takes care that pointers between environments are either oriented from top to bottom in the environmentstack or, from the environmentstack to the global stack. This is possible, because, whenever a free variable is matched against a term, all variables occurring in the term are, by convention, in the global environment.

Note : The user can declare ('mode declaration') that some terms will never be matched against a free variable, this makes it possible to classify more variables as local.

Copying pure code [3] [4]

The binding of a variable can be represented by a single pointer to a direct representation of the value. However, pure code containing variables cannot be a part of such a direct representation. Whenever a free variable is matched against a term of pure code containing variables, a copy of that term is made.
In this copy, the pure code for the variables is replaced as follows.

- is the variable free in the corresponding binding environment, then the copy get a free variable and the variable in the binding environment is bound to this new free variable.
- is the variable in the corresponding binding environment already bound to a part of a copy, a pointer is placed (a 'bound' variable) from the new copy to the existing copy.

Using a special copystack only popped on backtracking, this assures that pointers are either oriented from top to bottom in the environmentstack or from the environmentstack to the copystack or, have any direction in the copystack. Again, the environmentstack can be popped without danger of dangling pointers.

Notes

1. Warren has pointed out he can drop variables from the local environment once they will not be referenced during the further execution. More specific, local variables with only one occurrence ('void' variables) do not need a place on the environmentstack, the unification algorithm knows they are free; local variables occurring only in the heading can be dropped after unification of call and heading. Indeed, they are not referenced in the body of the procedure. The same, but for all variables, is true in the copying method.
2. Some parts of the global/copy stack can become inaccessible. Garbage collection and compaction is possible.

5. Discussion

To get an idea of the space efficiency of both approaches, we can compare the storage needs of nodes and binding environments.

With copying, a deterministic node needs 2 fields (CALL, FATHER); with structure sharing, 3 fields are needed (also a pointer to the global binding environment).

A backtrackpoint needs 6 fields with both methods (CALL, FATHER, BACK, PROCEDURE, TRAIL, pointer to global/copystack).

With structure sharing,the binding environment associated with the use of a procedure needs 2 fields for each variable. The division between local (on the environmentstack) and global (on the global stack) variables is determined by the definition of the procedure. With copying, one field for each variable is needed on the environmentstack. The space needed for copies is harder to determine. It depends on the pattern of the call which copies are made. The space needed by a copy depends on the chosen representation. We give two possibilities :

a. to copy a term containing variables, of the form $f(t_1,\ldots,t_n)$, we can use n+1 fields, one field to identify the functor f and one pointer to the representation of each argument. Such a representation gives fast access to the i-th argument.
b. With nested terms, it is possible to avoid the pointers to the arguments by placing the argument one after another. Then, only one field for each symbol is sufficient. This representation is more compact but the access to the i-th argument is slower.

With both representations, the copy of a variable is either a free variable or a pointer to the value of the variable.

For the procedures of the benchmark given in [7], we computed the space occupied by the binding environment of a typical call. For the 23 procedures

having a nonempty binding environment, the total space occupied by the 23 typical binding environments is as follows :

- Structure sharing : in total 216 fields are needed of which 130 in the global stack, with mode declarations, only 82 fields are needed on the global stack.
- Copying : with representation a, a total of 184 fields is sufficient of which only 83 on the copystack. With representation b, the copystack (and the total) is reduced by 10 fields.

This shows that structure sharing needs mode declarations to reduce the global stack to a size comparable with the copystack. Also, with copying, the space needed on the environmentstack is slightly smaller (also a deterministic node needs less space).

These results have to be taken with care, individual cases where copying is substantial worse than structure sharing are possible, i.e. when large terms need to be copied. The worst case between the 23 procedures : 8 fields on the global stack, 15 on the copystack with representation a, 11 with representation b.

We conclude that PROLOG interpreters are quite different from resolution theorem provers, especially because of their dept first search strategy. Although structure sharing results in very space efficient resolution theorem provers, it is not the only possibility for PROLOG-interpreters. The representation of literals in the proof tree is as close to conventional implementation techniques of Algol-like languages as to structure sharing. For the representation of the binding of variables, structure sharing is not the only alternative to be considered.

6. References

[1] Battani, G. and Meloni, H., Interpreteur du langage de programmation PROLOG. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Universite d'Aix-Marseille, 1973.

[2] Boyer, R.S. and Moore, J.S., The sharing of structure in theorem-proving programs.
Machine Intelligence 7 eds. Meltzer, B. and Michie, D., Edinburgh University Press, 1972, 101-116.

[3] Bruynooghe, M., An interpreter for predicate logic programs - basic principles.
Report CW10, Afdeling Toegepaste Wiskunde en Programmatie, K.U.Leuven, Belgium, oct. 1976.

[4] Bruynooghe M., Naar een betere beheersing van de uitvoering van programma's in de logika der Horn-uitdrukkingen.(In Dutch)
Doctoral Dissertation, Afdeling Toegepaste Wiskunde en Programmatie, K.U.Leuven, Belgium, may 1979.

[5] Colmerauer, A., Kanoui, H., Roussel, P. and Pasero, R., Un systeme de communication homme-machine en francais.
Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Universite d'Aix-Marseille, 1973

[6] Kowalski, R.A., Predicate logic as a programming language.
IFIP 74, North-Holland 1974, 569-574.

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

AN ALTERNATIVE TO STRUCTURE-SHARING IN THE IMPLEMENTATION OF A PROLOG INTERPRETER

C.S.Mellish,
Department of Artificial Intelligence,
University of Edinburgh,
EDINBURGH, UK

1. Introduction

This paper presents an alternative to "structure sharing" (SS) as a technique used in a Prolog interpreter. We firstly summarise some of the basic ideas used in Prolog implementations and show how structure sharing fits into such a scheme. We then present our alternative approach and make some comparisons. Our non structure sharing (NSS) approach, which uses a system of "copying", is used in a practical interpreter for the PDP-11. This interpreter is capable of running substantial Prolog programs, even though the PDP-11's address space is limited to 32K, 16-bit words. We compare the space efficiency of this interpreter with that of a structure sharing version that was constructed later. It turns out that the "copying" approach compares very favourably, although the comparison would be less favourable on a machine capable of holding two addresses in one word. Our comparison shows that the decision whether or not to use structure sharing in a Prolog implementation is not simple and must take into account a number of factors.

2. Some Basics of Prolog Implementation

We will now discuss some of the basic concepts used in some Prolog implementations. We follow here essentially the model presented by Warren [5], as developed from the work of Roussel [4] and Battani and Meloni [1].

2.1. Storing Prolog Programs

A Prolog program consists of a set of clauses and a goal statement. A clause represents a general rule that can be used many times in the satisfaction of the goal or its subgoals. Each time, we may want to consider different values for the variables that occur within it - that is, we need to be able to handle multiple instances of the same rule. When a clause instance is invoked to decompose an existing goal, we need to unify the clause head with the goal concerned and then consider the new subgoals given by the body of the clause. Since the main operation to be carried out with the parts of the clause is the recursive comparison involved in unification, it is appropriate to store them in a way that reflects the tree structure of the logical terms. (Note that in a Prolog compiler, some of the unification steps are "unfolded" in advance, so that this argument does not hold) The internal representations of these structures are usually called skeletons. In PDP-11 Prolog, skeletons have a prefix Polish format, with one machine word for each node of the tree. Since the value of a variable mentioned in a clause will vary from instance to instance, it is appropriate to represent a variable as an offset from an environment pointer to be provided each time.

In general, the lifetime of a clause will be quite long - it will remain until the user explicitly removes it. Clauses can be kept on a heap and the space freed by the removal of a clause garbage collected. For this reason, we will refer to the area where clauses are stored as the heap.

2.2. Basic Runtime Storage

During the running of a Prolog program, the Prolog interpreter must have access to a certain amount of space to store intermediate results of the computation. Some of this will be used for information about the control state of the interpreter - "return" addresses and so on - and the rest will be used to keep track of the values of the variables in the various clause instances. It is convenient to organise this into a set of frames - simple structures associated with the subgoals of the proof. A frame expresses both the control information about the subgoal's invocation and the variable values giving the necessary environment for its achievement. Thus, for instance, the PDP-11 Prolog system uses frames including the address of the place in a clause where this subgoal was invoked, the address of the frame for the goal invoking this as a direct subgoal, the address of the frame corresponding to the last choice made, the address of the clause chosen to satisfy this goal and the values of the variables for this particular clause instance. In many ways, this is like the information recorded on the stack for a conventional programming language like ALGOL. However, because many of the computations in Prolog are not determinate (several clauses may be available for providing possible solutions for a goal), there is the important difference that the space allocated for a frame cannot necessarily be reclaimed when a "procedure exit" takes place. In many cases, it is likely that an alternative solution for the goal may later be required, even though both it and its ancestors have been successfully satisfied once. Thus it is essential to keep a record of information like the last clause used and the environment of the parent (in the form of retained frames) whenever choices are made. Since backtracking in Prolog is strictly chronological, the retained frames can be kept in a particularly simple way. When backtracking occurs, everything done since the activation of the last choice frame must be undone, and the space used by the more recent frames can be reclaimed. It is therefore possible to keep frames in a stack-based system, with new fames "pushed" as they are activated and frames "popped" primarily when backtracking takes place.

In the basic loop of the interpreter, pointers to the currently relevant frames will be readily available, and from these the values of certain variables can be easily found. This information, together with the pointers to clauses (and the constituent skeletons) under consideration, enables the system to keep track of exactly what each clause instance looks like.

2.3. Constructed Terms

Prolog variables do not always have simple values (atoms, integers, or "undefined"), but can also be associated with complex terms. Such an association can be made either when a variable is unified with a variable that already stands for a complex term or when an "undefined" variable is matched directly with a complex pattern mentioned in a clause. In the second case, we can talk in terms of constructing the complex term, because this complex value has now become accessible through the variable and can be manipulated in various ways. The examples in fig 2-1 both involve the construction of a complex term to unify with a variable X.

```
?- p(X), ...          |        ?- p(f(A,B)), .....
                      |
p(f(A,B)) :- .....    |        p(X) :- .....
```

Figure 2-1: Examples of the Construction of Complex Terms

In these, the term "f(A,B)" is not accessible as an object until it is unified

with X; after this, it can be passed to other variables, further instantiated and so on. On the other hand, it is not appropriate to say that a term is constructed when "p" is called in either example in fig 2-2.

```
?- p(f(A,B)), ....          |          ?- q(X), p(X), ....
                            |
p(f(C,D)) :- ....           |          p(f(C,D)) :- .....
                            |          q(f(A,B)) :- .....
```

Figure 2-2: Examples of Other Operations on Complex Terms

In the first of these, the complex term is never available as an object to be manipulated - the specification of the two patterns merely serves as a way of associating A with C, B with D. We thus do not refer to this as a constructing operation. In the second, a term is constructed in the call to "q", and the matching operation in the call to "p" merely causes an association between the "inside" variables. This is more appropriately called accessing, than constructing.

It can be seen from these examples that it cannot be ascertained from individual clauses what terms will be constructed when they are invoked. In the Dec10 Prolog compiler [5], it is possible for the user to specify mode declarations which provide some of this information and enable the compiled code to be more efficient. Such possibilities will not be considered in what follows. Mode declaraions do in any case lessen the flexibility of the user's procedures.

When a complex term is constructed to be the value of a variable, that variable is suddenly associated with a great deal of information - both the form of the term and the values of the variables that appear inside it. It might seem impossible to represent this information in the same amount of space that is used for other variable values (such as atoms). The main focus of this paper is on two different approaches for tackling this representation problem.

2.4. Reclaiming Storage after Deterministic Computations

A computation (the satisfaction of a subgoal) that has no choice points within it is called determinate. The stack frames arising from such a computation can be "popped" from the stack when the subgoal has been established, for backtracking will never need to reconsider one to try another possibility. It is, however, necessary to ensure that no pointers are left from the remaining stack into the area that is reclaimed. Thus one should ensure that the binding together of two uninstantiated variables is always recorded by a pointer from the more recent to the less recent variable cell, and not the other way around. However, as soon as complex values of variables are possible, even this is not adequate. Consider the value given to X in fig 2-3.

```
?- f(X), g(X).

f(foo(A,B)) :- h(Y), j(Y).
```

Figure 2-3: Variables Occurring inside a Complex Term

Imagine that the satisfaction of "f(X)" is completely determinate (there are

no more clauses for "f", and "h" and "j" are also satisfied uniquely). We cannot reclaim the space associated with A and B, as these variables are part of the value of X to be used in "g(X)". However, although the space used by A and B cannot be reclaimed, other information associated with the subgoal (such as the value of Y) can be. If we wish to use the stack mechanism to achieve the reclamation, the only possibility is to have A and B represented somewhere off the main stack. We are thus led to introduce a second storage area, one designed for data that cannot be discarded at the end of a determinate computation. Since the values stored in this area can be discarded on backtracking in the same way as those in the first area, a stack organisation suggests itself again. This second data area is generally called the global stack, whereas the first is the local stack. Basically, the global stack grows in parallel with the local stack, with items being "popped" from both when backtracking occurs. The only difference is that items on the global stack cannot be reclaimed after a determinate computation. Because (in the absence of backtracking) items put on the global stack remain there indefinitely, it may be worthwhile trying to garbage collect it occasionally. This will clear out items that have become inaccessible since they were put there. Garbage collection of the global stack is used in Dec10 Prolog, but we will not discuss it further here.

2.5. Distributing Data between the two Stacks

The question of how to decide which pieces of data should go on each stack and when should now be discussed further. As is suggested by the example, a variable only needs to be allocated on the global stack if it occurs within a complex term constructed in a determinate computation and this complex term is "passed out" to be used elsewhere. Ideally the decision of where to store each variable should be made at runtime, because space on the global stack is not easily reclaimed and should be used sparingly. A reliable decision cannot be made at "compile time" (when the clauses are stored in the heap), because as we have noticed it is not even possible to determine then whether the clauses will be used to "construct" or to "access" complex structures. So the optimal place to make the decision is at runtime. However, when a clause is stored in the heap, the variables within it must be referenced in such a way that the values for any instance can be unambiguously located. This suggests that a decision at "compile time" is unavoidable.

In fact, the two approaches to storage management that we consider in this paper now begin to diverge. A "structure sharing" system, as we will see in section 3, relies on the fact that variables in the global stack have fixed positions relative to a current environment pointer, and so in particular it requires a decision on global/local status at "compile time". The alternative system that we present makes the decision at runtime, but incurs the extra overhead that every variable must be allocated a fixed offset in the local stack, in addition to an entry in the global stack for each time it appears in a constructed term. When a variable occurs in several places, these are linked together with pointers in such a way that all can be reached from the fixed location on the local stack which is referenced indirectly in the clause.

Where a "compile time" decision about the status of variables is made, an approximation is to make global every variable ocurring in a skeleton that could be used to construct a complex term. This can be determined syntactically - such a variable is one that appears within a complex argument of a goal in a clause. This method is used both by the Dec10 Prolog system [6] and in our structure sharing interpreter, although the use of "mode declarations" in Dec10 Prolog enables some of these variables to be made local. Where a "runtime" decision is made, an approximation is to give a global entry to a variable whenever a term containing it is constructed. The "compile time" algorithm falls short of optimality whenever a term that could

theoretically be constructed in fact is not. Both fall short when a term constructed in a determinate computation is not in fact "passed out" to the other parts of the program.

3. Structure Sharing

We saw earlier that in the main loop of the interpreter the state of each clause instance can be found out from a pointer to the clause together with a pointer to an appropriate stack frame. Moreover, this information must be stored in any case for the interpreter to properly organise backtracking (it must know which clause was last chosen) and "procedure exits" (it must continue from the stack frame corresponding to the "parent" goal). The idea of representing complex data by a pair of pointers (skeleton + environment, together called a molecule) is known as structure sharing, and this can also be used to represent terms constructed at runtime. Structure sharing capitalises on the fact that clauses are stored in a way that reflects their syntactic structure. Thus the skeletons in a clause provide all there is to know about the structure of a given instance except the values of the variables.

If it is possible to store a pair of machine addresses in the space allocated for a variable value, then it is straightforward to have molecules as possible values of variables. If a variable becomes instantiated as a complex term, it suffices to bring together a pointer to the skeleton with which the match was made and a pointer to the environment of the corresponding clause instance. "Constructing" is thus rather a simple process, and information about the internal structure of the variable's value can be obtained thereafter from the molecule. Note, however, that a molecule refers to a variable inside a constructed term via the fixed offset given in the skeleton. It is thus necessary to fix the relative locations of variables at "compile time" in order that the values can be found. In particular, since the variables inside constructed terms in general have to appear on the global stack in order to be accessible when the appropriate local frame has been reclaimed, a decision about global/local status must be made at compile time. Another consequence of the fact that local frames may be reclaimed is that it is essential for the environment pointer in a molecule to point to the global variable cells asociated with the goal, rather than the local ones.

As an example, consider the treatment of the clauses defining the concatentation relation between lists (fig 3-1).

```
C1. append(nil,X,X).
C2. append(A.B,C,A.D) :- append(B,C,D).

?- append(a.b.c.nil,nil,R).
```

Figure 3-1: Clauses and Goal Statment for 'append'

Since there is no reason why an 'append' goal should not have an uninstantiated variable as its first or third argument, the pairs <A,B> and <A,D> of variables in C2 may sometimes be involved in constructing operations. Hence A, B and D must be classed as global variables, with space allocated for them on the global stack, when C2 is chosen. On the other hand, C (and X) can get by with space on the local stack. When the goal is invoked and C2 is chosen, A in the new clause instance is unified with "a", and B is unified with the whole list "b.c.nil". This is represented by a molecule, with a pointer to the skeleton in the goal clause and a pointer to the environment

(which is, in fact, irrelevant as the list is ground). Similarly, R in the goal is unified with the list "A.D", this giving rise to another molecule. When the recursion is about to hit the "nil" case, there are 9 variable cells on the global stack (3 instances each of A, B and D) and 4 on the local stack (1 instance of R and 3 of C). Moreover, 5 of these variable values are molecules.

4. An Alternative Approach

Structure sharing saves space by sharing the structure common to multiple instances of the same skeleton. It has been proposed as a more space efficient method than more obvious approaches. We will now examine one such alternative, which is similar to what is used in conventional programming languages (like POP-2 and Algol68) for constructing new records dynamically in a heap.

The basic idea is that variables are allocated space primarily on the local stack, and that the references to them appearing in skeletons are always to these positions in the local stack. When a complex term is constructed, a concrete copy of the appropriate skeleton is created on the global stack, with the values of the variables appropriate to this instance substituted for the variable references. This means that there can be more than one location representing the same variable; these multiple locations must be appropriately linked with pointers. Once a concrete copy has been constructed, the values of its component parts can be simply read off, without the necessity of consulting environment pointers. In particular, variables that are unified with substructures can be dealt with by having pointers to the appropriate parts, for a variable that becomes instantiated as a complex term is represented simply by a pointer to the appropriate concrete copy.

Consider how this would work with the "append" example (fig 3-1). When C2 is chosen for matching with the goal, "A.B" in the new instance is unified with the ground list "a.b.c.nil" in the goal. As a result, A gets the value "a" and B is associated with a concrete copy of the rest of the list. Similarly, R's value turns out to be a pointer to a concrete copy of a "cons cell". The two variables inside this are A and D. A's value can be substituted in directly, whereas D is as yet uninstantiated and so a link must be established between the two locations representing it. In time, D's value is discovered to be a complex term, and so these two locations become indirectly linked to another concrete copy. When the "nil" case is about to be investigated, there are 13 variable cells on the local stack (1 instance of R and 3 each of A, B, C and D) and a total of 14 locations taken up by concrete copies on the global stack (5 for "b.c.nil" and 3 each for 3 individually constructed "cons cells").

5. Comparison

5.1. General Comments

5.1.1. Constructing Complex Terms

The whole point of SS is that there is a very low space overhead in constructing complex terms. As far as the global stack is concerned, the only pieces of information represented for a constructed term are the values of the variables occurring within it. Moreover, each variable is only represented once, even if it appears several times in the term or in multiple constructed

terms. On the other hand, the NSS approach needs to copy ground parts of a complex term and to have a location for every occurrence of a variable in a complex term. So structure sharing definitely seems to require less global stack when a term is constructed. An illustration of this is the construction of the list "b.c.nil" in our "append" example, where structure sharing needs no global space (no variables occur within it) but the other approach needs 5 locations (1 for each "symbol" in the skeleton). It is clear that an NSS system could be optimised to avoid copying ground subterms, but it would still lose by having to copy functor information and pointers for ground subterms of terms which are not themselves ground.

5.1.2. Accessing the Components of Complex Terms

When it comes to accessing the components of already constructed terms, SS does not perform so well. It allocates space on the global stack for the variables referring to the components, because at compile time there is no way of telling that an accessing rather than a constructing operation will be involved. On the other hand, the copying approach only puts items on the global stack when constructing takes place. In the "append" example, none of the "B" variables really need to be allocated on the global stack, because the pattern "A.B" is always used to decompose an already existing object.

5.1.3. Overall Stack Usage

From the last two paragraphs, we can see that the relative merits of SS and NSS as regards global stack usage will depend on the types of programs that we wish to run. On the one hand, we can construct a pathological program where huge structures containing repeated variables are continuously created but never accessed - on this, structure sharing will gain by arbitrary amounts. On the other hand, we can construct a pathological program that constructs a single complex term and repeatedly accesses its subterms - on this, structure sharing will lose by arbitrary amounts. Presumably, "real" programs fall somewhere in between these extremes.

As regards local stack storage, SS is clearly superior, since it allocates space for only some variables on the local stack, whereas NSS allocates space for all. Since local stack space can be reclaimed at the end of a determinate computation, NSS might be expected to do best with determinate programs. The hope of this approach is to reduce the total global stack usage at the expense of the total amount of stacks used. This clearly will not pay off if the local stack space can only rarely be reclaimed.

5.1.4. Speed

We cannot comment here on the relative speed of the two approaches, because such a comparison needs to take into account the machine's instruction set and addressing modes. However, we can note that our NSS system seems to save work looking up the values of variables in environments (for accessing subterms) at the expense of the "once and for all" copying operations (for constructing). What effect this has in practice is a matter for further investigation.

5.2. Representing Molecules in Machines with Small Word Sizes

Our discussion so far has assumed that it is possible to store a molecule (two addresses) within the space allocated for a single variable value. In general, the most economical and simple unit of space to use is the machine word (here taken to be the smallest independently addressable unit of storage

above some minimum size). In a machine with a small word size, we may be able to store things like atom representations and (smallish) integers in single words, but a molecule is really out of the question. This was a problem that we had to confront for the PDP-11, and in fact it was one of the main reasons why we turned to an alternative to structure sharing.

If we wish to stick to a structure sharing approach, how are we to represent molecules on a machine of small word size? One possibility is simply to take a larger unit, such as 2 words, for the value of a variable. Another is to have a molecule as a 2-word item to be separately allocated on the global stack and to which variable cells can point. The second of these will certainly take up less storage at any time if the number of variables allocated is at least twice the number of molecules. This has proved to be the case in all the examples we have tested.

What difference does this molecule overhead make for our "append" example? The whole example involves 5 molecules, and so an extra 10 locations now appear on the global stack. The NSS version of this example now has less on the global stack than the SS one, the reverse of what was previously the case.

It should be noted that this overhead in representing molecules sometimes affects a structure sharing approach even when the components of constructed terms are accessed. When structure sharing has to represent a complex subterm of a constructed term, it needs to produce a molecule, of course. Sometimes the appropriate molecule is already available as the value of a variable occurring in the constructed term. Otherwise a new one must be constructed. (This is what happens with the second instance of B in our "append" example). Such an action incurred no overhead previously, when we considered a molecule as something that could be stored in any variable cell.

5.3. Some Figures

How do all these factors interact and what are the relative merits of structure sharing and its alternatives in practice? When we decided to investigate this question, we had already developed a non structure sharing interpreter for the PDP-11 [3], and so we decided to construct a structure sharing version of it and make some comparisons. The structure sharing interpreter copes with the small word size of the PDP-11 by using the second approach for representing molecules, with a molecule being allocated as a serarate object on the global stack. In spite of these particular details, our figures provide a basis for a comparison between SS and NSS in general.

We took 7 different, already existing, programs and ran them on the two interpreters. Three of the programs (numbers 2, 3 and 4) represent a selection from the examples given in the appendix in Warren's paper [5]. Numbers 2 and 3 ("naive reverse" and "quicksort") intuitively seemed to involve roughly equal amounts of "accessing" and "constructing" (with a bias towards the latter), whereas number 4 ("serialise") was expected to involve a higher proportion of "accessing". For reasons that are not of interest here, we reduced the "quicksort" problem to deal with only the first 30 elements of the list given by Warren. Because our simple interpreters (without the aid of Warren's "mode declarations") regarded these programs as highly non-deterministic, we introduced for contrast a version of program 2 ("naive reverse") with added "cuts" as number 7. Next, we decided to investigate the claim that the real disadvantages of NSS would become apparent when we tried to write a Prolog interpreter in Prolog. We thus took the very simple Prolog interpreter given in [6] and presented it with programs 2 and 3, giving numbers 5 and 6. Finally, we wished to try something a bit different from these rather "toy" examples. We took the natural language parser from Dahl's [2] program, as modified by Pereira and Warren to deal with a new language (English) and a

different domain. This program reads an English sentence, character by character, from the terminal, converting it to a list of atoms, and constructs a logical formula expressing its "meaning". (This can then be used in conjunction with a database of facts about the world to produce an appropriate response) This program, when presented with the question "What files dating from Tuesday does the owner of the file dating from Monday possess?" forms our first example.

With each program, we measured various parameters of the space used. We did not measure runtime or heap usage (so there is no accounting for the space occupied by basic skeletons in the structure sharing system). The parameters (all measured in machine words) that were recorded are as follows:

A - The total amount of local stack in use at the end of the computation

B - The maximum amount of local stack in use at any time during the computation

C - The amount of space occupied by molecules (2 words each) at the end of the computation (SS only)

D - The amount of space occupied in the global stack at the end of the computation. This corresponds to the total amount of space that would be taken up in the long term if the program were invoked as a "subroutine" and all choice points were then discarded (for instance, with a "cut"). It also corresponds more or less to the maximum amount of global stack in use at any time in the computation.

E - The total amount of global stack in use, ignoring the space taken by molecules (D-C)

F - The total amount of space that would be taken up in the long term if the program were invoked as a "subroutine" but all choice points were kept (A+D)

G - The total amount of space needed to run the program (B+D)

The figures themselves are given in fig 5-1

5.4. Conclusions

It is hard to know how to select a representative sample of programs for such an experiment. As it was, the programs were selected in advance as easily accessible programs that covered a range of different situations. There was no postselection. Assuming that they are not too unrepresentative, we can draw the following conclusions:

- Comparing columns C and E, we see that in each case the number of variable cells on the global stack in SS is at least twice the number of molecules (since C is the amount of space used by the 2-word molecules, this result follows from the fact that C<E in all cases). This provides justification for our choice of representation of molecules as separate items on the global stack. It also suggests that Prolog implementations making use of long words to store molecules may not be very efficient in terms of the number of bits used, unless the extra length is used significantly

Program		Local Occ (A)	Local Max (B)	Mols (C)	Global Occ (D)	D-C (E)	A+D (F)	B+D (G)
(1) Natural Lang Prog	SS	620	700	510	1235	725	1855	1935
	NSS	780	907	-	805	-	1585	1712
(2) Naive Reverse	SS	2820	2823	988	2353	1365	5173	5176
	NSS	4185	4192	-	1454	-	5639	5646
(3) Quicksort	SS	907	913	340	763	423	1670	1676
	NSS	1270	1280	-	482	-	1752	1762
(4) Serialise	SS	267	311	338	924	586	1191	1235
	NSS	454	512	-	467	-	921	979
(5) Interp running (2)	SS	4975	4992	2046	4995	2949	9970	9987
	NSS	7892	7913	-	3414	-	11306	11327
(6) Interp running (3)	SS	2715	2732	1188	2757	1569	5472	5489
	NSS	4252	4276	-	1839	-	6091	6115
(7) (2) with "cuts"	SS	0	214	988	2353	1365	2353	2567
	NSS	0	278	-	1454	-	1454	1732

Figure 5-1: Comparison of SS and NSS Systems

for the representation of large integers, say.

- Comparing column D for NSS with column E for SS (and columns A and B for both), we see that if there were no overhead in representing molecules then SS would be better everywhere (except in program (4)). This suggests that SS is the more efficient technique in terms of the number of words used, given a machine of large word size.

- Comparing column D for the two systems, we see that (given the molecule overheads) NSS is in each case better than SS for global stack usage. This suggests that for determinate programs (where local stack is eventually reclaimed) NSS is superior in the amount of storage occupied in the long term.

- Comparing columns A and B for the two systems, we see that NSS always uses more local stack than SS. However, when this is added to global stack usage (giving columns F and G), the systems are fairly similar. Differences now depend on the particular task - (1), (4) and (7) are better for NSS; (2), (3), (5) and (6) are better for SS. But the differences are not great.

- It should be noted that most of these programs include "false choice points" where the interpreter cannot detect that a clause chosen will be the only one that matches the goal (this happens in our "append" example, where only one clause will ever be appropriate at any stage). These would disappear with a more intelligent interpreter that used indexing on the main functors of a predicate's arguments. The great difference that this would make is illustrated in the difference between programs (2) and (7). From our point of view, the important thing is that NSS comes out better in the

program with "cuts" (see columns F and G), whereas it is worse off in the one without. Thus, certain obvious improvements in our interpreters can be expected to make NSS an even more favourable option. Of course, other improvements (such as, perhaps, garbage collection of the global stack) may well work in favour of SS.

6. Some Final Remarks

As we have seen, the comparison between structure sharing and its alternatives is not a simple one, and no quick answer can be given as to which approach is best. It is interesting, however, that a significant factor in the decision is the relationship between the word size and address size of the machine on which the system is implemented.

As we have pointed out, neither of the systems presented is optimal in its use of the local and global stacks. It remains to be seen whether mixed approaches can be devised that have the benefits of both. It is hoped that this paper has made clear what some of the issues are and that it has revealed how much more work still needs to be done in this area.

7. References

[1]
Battani, G. and Meloni, H.
Interpreteur du Langage de Programmation Prolog.
Technical Report, Groupe d'Intelligence Artificielle, Univ of Marseille-Luminy, 1973.

[2]
Dahl, V.
Un Systeme Deductif d'Interrogation de Banques de Donnes en Espagnol.
Technical Report, Groupe d'Intelligence Artificielle, Univ of Marseille-Luminy, 1977.

[3]
Mellish, C. and Cross, M.
The UNIX Prolog System.
Software Report 5, Dept of Artificial Intelligence, Univ of Edinburgh, 1979.

[4]
Roussel, P.
Prolog: Manuel de Reference et d'Utilisation.
Technical Report, Groupe d'Intelligence Artificielle, Univ of Marseille-Luminy, 1975.

[5]
Warren, D.H.D.
Implementing Prolog - Compiling Predicate Logic Programs.
Research Reports 39 and 40, Dept of Artificial Intelligence, Univ of Edinburgh, 1977.

[6]
Warren, D.H.D, Pereira, F.C.N and Pereira, L.M.
User's Guide to DECsystem-10 Prolog.
Occasional Paper 15, Dept of Artificial Intelligence, Univ of Edinburgh, 1979.

8. Acknowledgements

I am grateful to Fernando Pereira, David Warren and Lawrence Byrd for many fruitful discussions, as well as for comments on earlier versions of this paper.

I. Some Technical Details

For those who wish to look more closely at our figures, here is a summary of some of the relevant technical details.

In both interpreters, the amount of administrative information stored in the local stack for a goal depends on whether the clause used is the last one for the predicate. For NSS, the space is 2 words for the last clause case; 5 otherwise. For SS, these numbers are 3 and 5 respectively. The rest of the space on the local stack is taken up by variable cells, at 1 word each. When a "cut" is encountered, all local stack after the "parent" frame is reclaimed and the "last choice point" frame becomes the last choice frame before the parent. When a goal has successfully "exited", its local stack space is reclaimed if the "last choice point" comes further back in the stack. We do not employ "tail recursion optimisation".

In the NSS interpreter, "concrete copies" on the global stack occupy the same number of words as the corresponding skeletons (1 word per "symbol"). Copying of ground skeletons is not optimised. In the SS interpreter, the global stack holds variable cells (1 word each) and molecules (2 words each). In neither case is the global stack garbage collected.

For simplicity, we have made no mention of the "trail" here. In fact, both interpreters store the trail interleaved with the global stack, although its size is not included in our figures. NSS can use slightly more trail than SS. In these examples, trail usage is identical except in program (1), where NSS uses in total 2 more words than SS.

Intermission --- Actors in Prolog

Kenneth M. Kahn
Stockholm University

Prolog as a computer language offers simplicity and a declarative interpretation of programs. Computer languages based upon computational entities called "actors" offer modularity, parallelism, data representation free programming and a simple but powerful computational semantics. Prolog is not well-suited for controlling computation, for defining new data types, or for writing programs that do not depend upon the physical representation of its data. This paper introduces the concept and motivation for actors and then describes a system called "Intermission" which implements actors in Prolog. The thesis presented is that a hybrid of actors and logic programming is a strong alternative to a language based upon either concept alone.

INTRODUCTION AND MOTIVATION

During the last ten years there has been much research on a new kind of computational entity, variously known as "actors", "objects", and "abstract objects". An actor combines both procedure and data into a single object. Actors perform computation via "message passing". Various computer languages have been built upon actors, among them are Smalltalk ([Goldberg 1976] and [Kay 1977]), Act 1 (a descendant of Plasma) ([Hewitt 1977] and [Lieberman draft]) and Director ([Kahn 1976], [Kahn 1978], and [Kahn 1979]).

The advantages of building systems in such languages are increased modularity and increased extensibility. Actors are also very well-suited for describing parallel processing. On highly parallel hardware it is anticipated that actor programs will be simplier and more efficient than the traditional alternatives.

Prolog ([LNEC 1979] and [Warren 1977]) is a programming language that has the unique feature that programs written in it can be viewed either procedurally or declaratively as logical statements. However, as a high-level programming language it has certain deficiencies. It is difficult to write programs that are not dependent upon the representation of the data. A typical sort program, for example, works only upon lists as they are provided by Prolog and a different version is needed for difference lists or other kinds of lists. It is also quite awkward in Prolog to handle "virtual data objects", such as the list of natural numbers, that is computed as needed. In general it is difficult to delay computations until they are needed. The ability to construct new kinds of data structures in Prolog is limited to those that can be represented by terms and list structures. These very general data structures are on occassion extremely inefficient in comparison with more specialized structures such as arrays or bit strings. Other more general data structures are often more convenient than terms or lists whose parts are accessed by their position in the structure. The "packagers" of Act 1 which support named subcomponents and partial descriptions are example of such. As I hope to show, many of these deficiencies of Prolog can be remedied by the inclusion of actors.

This report describes an implementation in Prolog of actors modeled after the Act 1 language. The implementation is called Intermission.

There is another motivation for implementing actors in Prolog besides the allieviation of the above mentioned deficiencies: that is that it may lead to a better or different actor semantics. The logical view of Prolog programs applies to the programs that implement actors in Prolog. Certain unusual features of Prolog carry over to the actors implemented in Prolog. The possibility of reversing the normal "input" and "output"

variables of a Prolog relation, for example, changes the normal semantics of actor computations. The ability to use the same program in many different ways is very attractive and adds new dimensions to actor programs.

WHAT IS AN ACTOR

An actor is a computational entity that combines in a single unit both program and data. Actors therefore subsume both procedures, functions, and all kinds of data structures. Computation is performed only by sending messages. It is not possible to reach inside an actor or change an actor without sending that actor a message requesting such an operation. This guarantees the integrity of the objects of computation. The programs written in an actor language depend only upon the behavior of modules and not upon their physical representation.

An actor consists of two parts: a "script" which decides what should be done with incoming messages and a set of "acquaintances" which are the other actors that the actor knows. The acquaintances play the role of local data for the actor. An actor can only send a message to someone it knows, i.e. either to one of its acquaintances or to someone referred to in the incoming message.

Actors can represent a data type in many different ways and the programs that use them need not know which type it is dealing with. For example, one can define matrices as two-dimensional arrays, or as pairs of indices and values (perhaps stored in a hash table), or as a procedure that computes the values as needed. The first alternative is the traditional way of representing matrices and exploits the way memory is addressed in conventional computers. The second one provides great savings of space and time if the matrix is large and sparse. The third alternative is ideal for special matrices such as identity matrices.

Since programs depend only upon the behavior of the "data" there is no difficulty defining infinite objects such as the list of prime numbers. Lists are actors that accept messages asking for their first and rest components, for printing, for determining equality with other lists, and for adding new elements. Some lists accept other messages such as those asking for its length or to append another list to itself. There is no reason an infinite list cannot be defined to do these things.

In a totally consistent actor system, it is relatively easy to add actors that represent computations yet to be done or that are being done in parallel (perhaps even on another processor). The actor can be passed around, inserted in lists, and the like and only when its value is needed must the computation involved finish. Again the dependence upon the behavior, as opposed to the physical implementation of a data structure, makes this possible.

HOW TO PUT ACTORS INTO PROLOG

If one were adding actors to Prolog to produce a better practical computer language then they would have to be incorporated at a low level of implementation. Perhaps they would have the same status as lists, symbols and numbers have in current implementations of Prolog. However, this report describes an incorporation of actors in Prolog whose purpose is to clarify and explore the issues and ideas involved. As a consequence, the actors are added to Prolog in a very clean, general, and flexible manner that is unfortunately extremely inefficient. The implementation has proved adequate for running simple programs like quick sort or the sieve of Eratosthenes and for implementing six or seven different types of lists.

Actor theory defines the basic computation mechanism of message passing as "uni-directional". In other words, you send a message to an actor and

its up to it to reply or to send messages off to others. If we were to implement actors in this completely general fashion then we might have a Prolog relation called "sent" between four terms as follows:

```
sent(Continuation-1, Message-1, Continuation-2, Message-2)
```

which is interpreted as message-1 is sent to the actor continuation-1 and as a result message-2 is sent to continuation-2. When all the terms are instantiated the following should happen:

```
sent(Continuation-2, Message-2, Continuation-3, Message-3)
```

In other words, each transmission of a message to a continuation should create a new continuation and a new message. The newly created message and continuation then become the participants in the next message transmission. In the full generality, the "result" of sending a message to an actor should be any number of new transmissions. The problem with this setup is that the programs written in this fashion repeat the same text twice (e.g. Continuation-2 in the above example). Actor interpreters avoid this and the Prolog interpreter could be changed similarly.

Instead of making this major change to Prolog's interpreter a more limited version of actor semantics was implemented where all message transmissions "return" a value. This is similar to the approach taken in Smalltalk, Director, and the abstract objects in Lisp Machine Lisp. In Prolog terms that means that the "sent" relation has only three terms as follows:

```
sent(Target, Message, Answer)
```

This scheme was generalized by allowing for any number of "answers" including zero (for print messages, for example). It is the one used in Intermission. For example, let us consider a simple implementation of lists to illustrate the "sent" relation. Of course, since Prolog already has lists this is meant solely as an illustration of the basic ideas. It turns out we will represent actors as Prolog lists, so this clearly will not make Prolog more powerful --- later examples are for that. Actors are represented as lists whose first element is their type which plays the role of the "script" and the rest of the list are the "acquaintances" of the actor.

First we define the message "first" which returns the first element of the actor list, and the message "rest" which returns the rest of the list. (Lower case words are literals; upper case are variables.)

```
sent([list,First,Rest],first,First).
sent([list,First,Rest],rest,Rest).
```

For example, to find the rest of the list (A B) we type the following to Intermission.

```
sent([list,A,[list,B,[emtpy_list]]],rest,R).
```

and the system responds

```
R = [list,B,[emtpy_list]]
```

Next we define a means of making lists by adding new elements in front (i.e. "cons" in Lisp).

```
sent([list,First,Rest],[add_element,New_element],
                       [list,New_element,[list,First,Rest]]).
```

To get started we need the empty list, which we will represent as an actor without any acquiantences.

```
sent([empty_list],[add_element,New_element],[list,New_element,[empty_list]]).
```

Now suppose we want our lists to respond to "length" messages. We could define this as follows.

```
sent([list,First,Rest],length,N) :- /* the list's length is N if */
 sent(Rest,length,M), /* the length of its rest is M */
 sent(M,[+,1],N). /* and N = M + 1 (numbers as actors are described below) */

sent([empty_list],length,0).
```

To extend our lists so that they can respond to messages asking if they are equal to another list, we have the following.

```
sent([list,First,Rest],[equal,Another_list],true) :-
 sent(Another_list,[are_you_a,list],true), /* is the other is a list */
 sent(Another_list,first,Others_first),
 sent(First,[equal,Others_first],true), /* is my first equal to his first */
 sent(Another_list,rest,Others_rest),
 sent(Rest,[equal,Others_rest],true). /* and his rest equal to mine */
```

We need to stop sometime so we define empty lists to equal themselves.

```
sent([empty_list],[equal,[empty_list]],true).
```

We introduced a new message of general usefulness that verifies the type of the actor. We need both lists and the empty list to answer yes to the question "are you a list".

```
sent([empty_list],[are_you_a,list],true).
sent([list,First,Rest],[are_you_a,list],true).
```

There remains a problem at this point. Suppose we have the list (A B C) and we ask it are you equal to (Z B C) then it will ask A if it is equal to Z. But A and Z are Prolog symbols not actors. The solution to this problem that we take is similar to that taken by Act 1 in handling "rock bottom" actors. We represent numbers and symbols specially but they still behave just like full-fledged actors. The representation we choose is Prolog's. For example, to enable Prolog symbols to answer "equal" message we do the following.

```
sent(Symbol,[equal,Symbol],true) :- atomic(Symbol).
```

A LIST OF INTEGERS

The actor lists we just defined have no advantages over Prolog lists and seems more complex and awkward. An example that points to some of the advantages of the actor approach is a list of integers. We can represent a large class of them simply by an actor with three acquaintances: the first element, the last element, and the difference between sucessive elements. We can define "first" and "rest" messages for such lists as follows.

```
sent([nlist,Begin,End,Increment],first,Begin).
sent([nlist,Begin,End,Increment],rest,[nlist,New_begin,End,Increment]) :-
 sent(Begin,[+,Increment],New_begin).
```

The "rest" message can be read as "if a list of numbers is asked for the rest of its elements it answers with a list of numbers just like itself except that the first element is the old first element plus the increment." Notice that the addition is performed by actors; the first number is sent the message "add the value of increment" to yourself.

Numbers are able to take messages like this because they are "rock bottom" actors with message handlers such as the following.

```
sent(Number,[+,Another],Result) :-
 integer(Number), integer(Another), /* if they are both numbers */
 Result is Another+Number. /* then add them */
```

We could go on and define "equal", "length", "are_you_a", and "print" messages for these new kinds of lists but some of it will be a repetition of the previous clauses. Other kinds of lists will be defined and some will have even more in common with the behavior of our actor lists. The solution to this problem in Act 1 and Director is "message delegation". When an actor does not know how to handle a particular message it "delegates" it to someone it thinks can handle it for him. This actor which delegates is called the "client" and it delegates to its "proxy". Delegation is implemented by having the following two clauses of "sent" at the very end.

```
sent(Anyone,Message,Result) :- /* if any actor cannot handle a message */
 sent(Anyone,proxy,Proxy),  /* we ask the actor who his proxy is */
 sent(Proxy,[handle_for,Anyone,Message],Result).
 /* and send the Proxy the message asking it to handle this for the actor */

sent(Anyone,[handle_for,Client,Message],Result) :-
 /* and if a proxy cannot handle the problem passed to him, he passes it
    on along to his proxy */
 sent(Anyone,proxy,Proxy),
 sent(Proxy,[handle_for,Client,Message],Result).
```

This simple scheme greatly increases the power of the actor system by facilatating the sharing of knowledge. The programmer now can place knowledge at as high a level of abstraction as desired. For example, we can define a "print" message for all kinds of lists as follows.

```
sent(list,[handle_for,A_list,print]) :-
 write('('), /* print an open parenthesis */
 sent(A_list,print_elements),
 write(')'). /* print a close parenthesis */

sent(list,[handle_for,A_list,print_elements]) :-
 write(' '),
 sent(A_list,first,First),
 sent(First,print), /* send a "print" message to the first element */
 sent(A_list,rest,Rest),
 sent(Rest,print_elements). /* send a "print elements" message to the rest */
```

Notice that since "print" messages are sent only for their side effect and there is no "result" the "sent" relation has only two arguements here. (This actually causes slight problems with delegation. The two clauses implementing delegation need to be copied with only two arguements to "sent".)

Now if we declare that our number lists have the generic "list" actor as a "proxy" as follows,

```
sent(nlist,proxy,list).
```

then we can print "nlists" without difficulty. For example,

```
sent([nlist,1,15,2],print). /* results in the following be printed */
( 1 3 5 7 9 11 13 15 )
```

This is fine but how should a list like "[nlist,1,1000000,1]" be printed? The delegation mechanism provides only a default behavior, we can override it in this case as follows.

```
sent(nlist,[handle_for,A_list,print_elements]) :-
 /* no need to override "print" since we still want the parentheses */
 sent(A_list,length,Length), /* find the length of the list */
 sent(Length,[>,5],true), /* and its greater then 5 */
 sent(A_list,print_elements_with_dots). /* print it specially as follows */

sent(nlist,[handle_for,A_list,print_elements_with_dots]) :-
 write(' '),
 sent(A_list,first,First),
 sent(First,print), /* print the first element */
 write(' '),
 sent(A_list,2,Second),
 /* We define lists to respond to numbers with their Nth element */
 sent(Second,print), /* print the second element */
 write(' '),
 sent(A_list,3,Third),
 sent(Third,print), /* print the third element */
 write(' ... '), /* print three dots */
 sent(A_list,last,End), /* lists return the last element to "last" messages
 sent(End,print). /* and print the last element */
```

Our change has not affected lists of numbers with less than 6 elements. However if we try to print the first million integers, the list behaves sensibly as follows.

```
sent([nlist,1,1000000,1],print).
( 1 2 3 ... 1000000 )
```

Notice that our print method asks the list for its length. This could be quite expensive considering that the general method for length keeps sending "rest" messages until the list is empty. (One part of our "nlist" actor that has been ommitted here are the clauses that determine if an "nlist" is empty.) We can fix this by adding a method which simply finds the difference between the first and last element and divides by the increment when asked for the length. A similar problem exists with the "last" element message. This one is trivial for "nlist" to handle and quite expensive to let the general method in "list" handle it.

As it turns out, we have defined "nlists" in such a way that they can represent infinite lists. For example, the list of all the positive odd integers is just "[nlist,1,infinity,2]". "Infinity" is just a number actor which has clauses such as the following.

```
sent(infinity,proxy,number). /* infinity delegates to number */
sent(infinity,[+,Anyone],infinity). /* infinity plus anything is infinity */
sent(infinity,[>,Anyone],true). /* infinity is greater than any other */
```

Properly all the clauses of "infinity" should be modified to make sure that the other number which it is being compared with is not itself infinity. This could be done as follows

```
sent(infinity,[>,Anyone],true) :- sent(Anyone,[are_you,finite],true).
```

The list of odd integers can be added to, taken apart, printed, asked its length, and so on. For example,

```
sent([nlist,1,infinity,2],print).
(1 3 5 ... infinity)

sent([nlist,1,infinity,2],length,L).
L=infinity.
```

DELAYED COMPUTATIONS

Sometimes it is easy to describe what each object or process in a computation should be or do but the parts depend upon each other in such complex ways that it is difficult to order the events. One would like to have each process run in parallel and wait when they need some value that has yet to be computed. Actor systems are well-suited for describing parallel processing because of message passing and the internalization and localization of state descriptions. However, since achieving concurrency within Prolog would require major changes to the interpreter the actor primitives for parallelism were not implemented. (See [Clark 1980] for a description of some of these changes to Prolog.)

A primitive for delaying computations until the value is needed has, however, been implemented. It is especially useful for computing with infinite objects. "Delay" could be defined as follows in Prolog.

```
sent(actor,[handle_for,Client,[delay,Message]],
          [delayed_transmission,Client,Message,Value]).
```

If any actor receives a message beginning with "delay" followed by a message it just "returns" an actor that is a delayed transmission whose acquaitences are the original recipient of the message, the delayed message, and a variable representing the to-be-computed result of sending the message to the actor. The next problem is to define delayed transmissions as actors that when they get a message finally do the delayed action and then send the message on along to the result.

```
sent([delayed_transmission,Target,Delayed_message,Value],Message,Result) :-
 sent(Target,Delayed_message,Value), /* compute the delayed computation */
 sent(Value,Message,Result). /* and send Message to the result */
```

The difficulty with this solution is that the actor will recompute its delayed computation every time it gets a message. We would like it to compute it the first time only and from then on have it behave as the result. To avoid this we take advantage of Prolog's ability to compute with partially instantiated structures. The first time the computation is performed the last element of the list representing the delayed computation is instantiated. To take advantage of this we add the following

```
sent([delayed_transmission,Target,Delayed_message,Value],Message,Result) :-
 nonvar(Value), /* If the Value is instantiated, then use it */
 sent(Value,Message,Result). /* and send Message to the Value */
```

One use of this "delay" message is to construct the list of natural numbers as follows.

```
sent(list,[natural_numbers_beginning,N],Result) :-
 sent(N,[+,1],N_plus_one),
 sent(list,[delay,[natural_numbers_beginning,N_plus_one]],Delayed_rest),
 sent(list,[add_element,N,Delayed_rest],Result).
```

The result of sending a "natural numbers beginning 1" message to "list" is the list of natural numbers. In many ways this is a less useful implementation of a list of numbers than the one described earlier since if we ask it for its length or to print the computation will not terminate.

A very old algorithm for computing prime numbers is called the "sieve of Eratosthenes". The idea is simple. You begin with the list of integers beginning with 2. You then repeatedly cross out all numbers that are multiples of the first element of the list. The list consisting of those first elements is the list of primes. With actors and "delay" messages there are no difficulties dealing with these infinite objects and computations. We can define the list of primes as follows.

```
primes(The_primes) :-
 sent(list,[natural_numbers_beginning,2],Numbers),
 sent(Numbers,repeatedly_cross_out_multiplies_of_first,The_primes).

sent(list,[handle_for,List,repeatedly_cross_out_multiples_of_first],Result) :-
 sent(List,first,First),
 sent(List,rest,Rest),
 sent(Rest,[cross_out_those_divisible_by,First],Those_left),
 sent(Those_left,[delay,repeatedly_cross_out_multiples_of_first],Primes),
 sent(list,[add_element,First,Primes],Result).

sent(list,[handle_for,A_list,[cross_out_those_divisible_by,N]],Result) :-
 sent(A_list,first,First),
 sent(A_list,rest,Rest),
 Mod is First mod N,
 cross_out_helper(First,Rest,Mod,N,Result).

cross_out_helper(First,Rest,0,N,Result) :-
 /* mod is 0 meaning that this one is a multiple of N */
 sent(Rest,[delay,[cross_out_those_divisible_by,N]],Result).

cross_out_helper(First,Rest,_,N,Result) :-
 /* is not a multiple of N so keep it and delay the recursion on the rest */
 sent(Rest,[delay,[cross_out_those_divisible_by,N]],Those_left),
 sent(list,[add_element,First,Those_left],Result).

print_primes :-
 primes(P),
 sent(P,print).
print_primes.  /* results in the following being typed */
( 2 3 5 7 11 13 17 /* until we interrupt the program */
```

DATA REPRESENTATION FREE PROGRAMMING

The original instigation of this research was reading two Prolog programs: one for quick sorting an ordinary list and another for difference lists ([Hansson 1979] and [Hansson 1980]). Using Intermission's message passing we can write a quick sort that works for any sort of list as follows.

```
sent(list,[handle_for,A_list,[quick_sort,Relation]],A_list) :-
 /* the sorted version of any empty list is itself */
 sent(A_list,empty,true).

sent(list,[handle_for,A_list,[quick_sort,Relation]],Sorted_list) :-
 sent(A_list,first,First),
 sent(A_list,rest,Rest),
 sent(Rest,[partition_by,[Relation,First]],Less_than_or_equal,Greater_than),
 /* Partion the list into two parts --- those greater and those less than
    the first element of the list */
 sent(Less_than_or_equal,[quick_sort,Relation],First_part_sorted),
 sent(Greater_than,[quick_sort,Relation],Rest_sorted),
 /* just sorted the two smaller lists --- now put them back together again */
 sent(Rest_sorted,[cons,First],New_rest_sorted),
 sent(First_part_sorted,[append,New_rest_sorted],Sorted_list).

sent(list,[handle_for,A_list,[partition_by,Predicate]],A_list,A_list) :-
 sent(A_list,empty,true). /* an empty list partitions into itself */

sent(list,[handle_for,A_list,[partition_by,Predicate]],True_ones,False_ones) :-
 /* partition A List into those that Predicate is true of and those its not */
 sent(A_list,first,First),
 sent(A_list,rest,Rest),
 sent(First,Predicate,true),
 sent(Rest,[partition_by,Predicate],Rest_true_ones,False_ones),
 sent(Rest_true_ones,[cons,First],True_ones).
```

```
ent(list,[handle_for,A_list,[partition_by,Predicate]],True_ones,False_ones) :-
/* this handles the case where the Predicate is false */
sent(A_list,first,First),
sent(A_list,rest,Rest),
sent(First,Predicate,false),
sent(Rest,[partition_by,Predicate],True_ones,Rest_false_ones),
sent(Rest_false_ones,[cons,First],False_ones).
```

otice that this quick sort procedure works for any kind of list and any elation between elements. For example, it works on Intermission's rdinary lists, lists of integers (finite ones only), difference lists, ists of lists (an implementation of lists for which the "append" peration is very inexpensive), and Prolog lists (preceeded by a symbol ndicating that they are Prolog lists so as not to be confused with ther actors). One somewhat silly test which shows off some of the eatures of Intermission is one which a list of all different sorts of ists is sorted by the length of its elements. Some of the elements are nfinite lists. All that was required for this test was to extend lists o answer messages asking if they are longer than another list. This was one as follows.

```
ent(list,[handle_for,A_list,[longer,Another]],Answer) :-
sent(A_list,length,My_length),
sent(Another,length,His_length),
sent(My_length,[>,His_length],Answer).
```

THER WAYS THE FEATURES OF INTERMISSION MIGHT BE PROVIDED
e have shown how by replacing the data types of Prolog with actors we ave increased the expressive power of the language. Certain kinds of ata structures that were difficult to express in Prolog (such as nfinite lists) are not difficult in Intermission. Intermission also rovides more control over the computation as exemplified by the "delay" essage. Intermission programs are by their very nature independent of he representation of the data.

ne question that needs to be answered is whether these advantages of ntermission could not have been achieved easily in Prolog. We need to mphasize the word "easily" in our question since we are dealing with niversal Computers that are Turing equivalent. For example, suppose nstead of sending messages as we do in Intermission we have Prolog elations for dealing with all data structures. Compare the following

```
ent([list,First,Rest],first,First). /* Intermission's way */
irst([list,First,Rest],First). /* Prolog's way */
```

his scheme is admittedly simplier and a less drastic departure from normal rolog but is much more limited than the message passing actor system n Intermission. The scheme gets more awkward when the message is a list tructure. More significant is the difficulty of incorporating elegation and delay in such a scheme. These operations apply to any ind of message which is very difficult to express in this setup.

nother possible answer to the question of whether the advantages of ntermission could be achieved with a less drastic departure from Prolog s yes, the implementation of IC-Prolog is such an alternative. IC- rolog allows the user to annotate their Prolog programs to control the omputation. They have a primitive that is similar to Intermission's elay message, for example. One important difference between the IC- rolog's control features and Intermission's is that Intermission's are ot built into any interpreter but were defined with three or four short lauses in Intermission. This suggests that Intermission's users are in better position to define their own control "primitives" rather than imply accept those provided by the language implementors. The other eatures of Intermission: delegation and data representation ndependence are as difficult to provide in IC-Prolog as in Prolog.

PROBLEMS WITH INTERMISSION

There are many problems with Intermission of course. The syntax is very awkward and verbose. This is not a consequence of the use of actors or message passing but of how Intermission was built upon Prolog. Delegation, for example, requires handlers for "handle for ..." messages. With the proper defaulting this level of detail does not need to appear in any user's program as is the case in most actor languages. The worst problem with the syntax of Intermission compared to Prolog is the use of explicit constructors and selectors instead of pattern matching. This is ironic since in all other actor languages pattern matching is an essential part of the language. In Act 1, for example, the pattern matching is performed by actors and is very powerful and extensible. Intermission can only partially make use of Prolog's pattern matching because it is representation dependent. What is needed in Intermission is a pattern like "[Head,..Tail]" which will match any kind of list while binding "Head" and "Tail" to the result of sending a "first" and "rest" message, respectively, to the list. These syntactic problems with Intermission could be overcome by placing a "front end" parser between Intermission programs and Prolog or by changing the Prolog interpreter.

Another problem with Intermission is that all message passing is bi-directional. For full generality, especially with respect to defining new kinds of control structures, we would prefer a uni-directional basis. This change is difficult to acheive within Prolog and seems to require large changes to the behavior of the Prolog interpreter.

Certain deficiencies with Prolog have been inherited by Intermission. For example, in the presence of backtracking it is difficult to describe default behavior. The delegation mechanism is intended to take over only when the actor in question cannot handle the current message. One cannot describe a pattern for messages that an actor will not accept and instead one is forced to rely upon the search order of Prolog's theorem prover so that the delegation clauses are tried only after all the others have. (Prolog's "cut" primitive could help here. However, its use limits the important feature of Prolog of using a program in different ways and so was never used in Intermission.)

Debugging in Prolog is very difficult compared to an actor or Lisp-like language. The major difficulty is the lack of a distinction between failures and errors. Both cause backtracking, however in the case where the program has a "bug" this leads to bizarre behavior (e.g. message delegation in an inappropriate situation) and often the program will not terminate. It is especially difficult to have the program break at a point where it is clear that an error should be signaled. A serious example of this is the inability to break when an actor receives a message that neither it nor its proxies can handle. If one adds such a clause then it will be triggered erroneously in the process of backtracking. Some of these difficulties would be alleviated by implementing Intermission in QLOG, an implementation of Prolog which provides most of InterLisp's debugging facilities to the user [Komorowski 1980].

The most serious short-coming of Intermission as a practical programming language is that it is terribly slow and inefficient. This is a consequence of the way in which actors will built on top of Prolog instead of being incorporated at a much lower level. Much of the inefficiency is also due to the lack of control over Prolog's search behavior. This aspect could be alleviated by implementing Intermission is a language like IC-Prolog.

There is no reason to believe that the use of actors and message passing necessarily entails computational inefficiencies. Experience with Act 1, Director, and Smalltalk indicate that much (if not all) of the "overhead" of actors and message passing can be compiled out without any

loss of flexibility. Many anticipate actor compilers that will be more efficient than those of conventional languages because the programs being compiled rely only upon the behavior of the objects involved leaving the compiler great freedom in performing optimizations which are perfectly safe. Also there has been much research that suggests that the actor model of computation leads to programs that can fully exploit the parallel processors of the not-too-distant future.

POSSIBLE CONTRIBUTIONS TO THE ACTOR MODEL OF COMPUTATION

This research was performed with two goals: to improve Prolog by adding actors and to improve actors by implementing them in Prolog in such a way so as to preserve some of its unusual features. One beneficial side-effect of this research is that Intermission seems to be a good way of introducing actors and message passing concepts to a community familiar with Logic Programming.

One very appealing feature of Prolog is the ability to use the same program in many ways. For example, the Prolog definition of "append" can be used not only to compute the result of appending two lists together but can also be used as a predicate to verify if the result of appending two lists is a third list, as a generator of pairs of lists that append to a particular list, as a way of finding the difference between two lists, and as a generator of triples of lists such that the first two appended form the third. In Prolog only some of these uses are efficient, however IC-Prolog provides a means of greatly increasing the efficiency of the other uses.

This feature of Prolog has only partially been preserved in Intermission. The problem is that when sending a message to an un-instantiated actor Prolog often never terminates while it creates more and more examples of the wrong kind of actor. The difficulty is primarily one of the inability in Prolog to control its search. It is expected that this difficulty would be remedied by implementing Intermission in IC-Prolog. Intermission in IC-Prolog would also provide the important ability to describe and test the program first and then add commentary to improve its performance.

One of the most important features of Prolog is that the programs have both a declarative logical interpretation and a procedural one. Because of the way Intermission was built upon Prolog programs written in Intermission also have these two interpretations. This is important for several reasons. Sometimes the declarative interpretation is simplier and thus it is easier to write and debug programs in such cases. The task of implementing programs that understand themselves is eased by the ability to reason about the code as logical statements. Verification of programs is made easier by taking a logical interpretation of the code. There is also the possibility that theorem provers could derive programs from specifications ([Hansson 1979] and [Hansson 1980]).

FUTURE RESEARCH

Experience is needed using Intermission for more than "toy" programs. Before this can be done a more efficient and practical implementation needs to be made. The big problem here is how to accomplish this while preserving the features of Prolog, especially the declarative interpretation of programs. Prolog programs can be interpreted declaratively or procedurally. One view of this research is that its an attempt to generalize the "procedural" interpretation of Prolog programs without losing the declarative.

Another avenue of research is to implement a Prolog-like (or IC-Prolog-like) language in an actor language such as Act 1. This would help clarify the relationship between the two languages and would provide a many of Act 1's features to Prolog. The idea here is similar to that behind QLOG which provides Interlisp features to Prolog. One exciting possibility is that this implementation would be the more practical implementation of Intermission because actors would be underlying all of its data types and pattern matching.

ACKNOWLEDGEMENTS
I would like to thank Sten-Ake Tarnlund for his help and encouragement with this research. I am also indebted to Carl Hewitt for most of the ideas about actors incorporated in Intermission and to Henry Lieberman whose actor langauge Act 1 was the model for Intermission.

REFERENCES

[Clark 1980]
Clark K. and McCabe F. "The Control Facilities of IC-Prolog", Department of Computing and Control, Imperial College, London

[Goldberg 1976]
Goldberg, A., Kay A. editors, "Smalltalk-72 Instruction Manual" The Learning Research Group, Xerox Palo Alto Research Center, March 1976

[Hansson 1979]
Hansson A and Tarnlund S, "A Natural Programming Calculus", IJCAI-79, Tokyo, Japan, August 1979

[Hansson 1980]
Hansson, A. "A Formal Development of Programs", Department of Information Processing Computer Science, The Royal Institute of Technology and The University of Stockholm, January 1980

[Hewitt 1977]
Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages", Journal of Artificial Intelligence, Vol 8. No. 3, June 1977, pp. 323-364

[Kahn 1976]
Kahn, K. "An Actor-Based Computer Animation Language", ed. Treu, S., pp. 37-43 Proceedings of the SIGGRAPH/ACM Workshop on User-Oriented Design of Interactive Graphics Systems, October 14-15, 1976

[Kahn 1978]
Kahn K. and Hewitt C. "Dynamic Graphics using Quasi Parallelism", Computer Graphics, Vol. 12, No. 3, August 1978, pp. 357-362

[Kahn 1979]
Kahn, K. "Director Guide", MIT AI Memo 482b, December 1979

[Kay 1977]
Kay, A. "Microelectronics and the Personal Computer", Scientific American, September 1977

[Kornfeld 1979]
Kornfeld, W. "Using Parallel Processing for Problem Solving", MIT EECS Master Thesis, Spring 1979

[Komorowski 1980]
Komorowski, H. J. "Qlog - The Software for Prolog and Logic Programming", Datalogi Linkoeping Research Report, Linkoeping University, April 1980

[Lieberman draft]
Lieberman, H. "A Preview of Act 1", submitted for publication

[LNEC 1979]
"How to Solve it with Prolog" Laboratorio Nacional de Engenharia Civil, Lisbon Portugal, August 1979

[Warren 1977]
Warren, D. "Implementing Prolog -- compiling predicate logic programs", D.A.J. Research Report No. 39, Department of Artificial Intelligence, University of Edinburgh, May 1977

IC-PROLOG - LANGUAGE FEATURES

K. L. Clark and F. G. McCabe, Imperial College of Science and Technology, Department of Computing and Control, 180 Queen's Gate, London, SW7 2BZ

Extended Abstract

IC-PROLOG differs from the other implemented PROLOG's in two major respects. Firstly, it does not provide the extra-logical feature "/", nor the meta-logical facilities to add and delete clauses during a computation. In part compensation, it provides negation as a primitive and allows the programmer to use conditionals. The other major difference is the control facilities. With the use of annotations the programmer can cause the computation to proceed in a pseudo-parallel mode, and to coroutine on the basis of the flow of data through shared variables.

Negation

In IC-PROLOG a program clause is an implication of the form:

$$B \leftarrow L1 \& \ldots \& Lm \ , \ m \geq 0$$

where B is an atom and L1,..,Lm are literals. Thus, a negated atom ~A can appear as a procedure call of a clause. Negated atoms are evaluated by the negation-as-failure proof rule. That is, if all possible proofs of A fail, ~A is assumed to be proven. As explained in Clark [1978], this proof rule can be reconciled with the truth functional semantics of "~" if one assumes that the implications of the PROLOG programs are a shorthand for a set of equivalences and inequalities. The IC-PROLOG implementation of negation is faithful to that semantics. In particular, if the proof of A succeeds, ~A is failed only if no variables in A have been bound. The way

(2)

in which this binding condition is checked without too much overhead is explained in our companion paper on implementation. [Clark and McCabe 1980].

Conditionals

The pair of clauses

P ← C & D

P ← ~C & E

express a conditional definition of P. In IC-PROLOG a pair of clauses like this can be absorbed into the non-clausal implication.

P ← C THEN D ELSE E.

The logical semantics of the implication is that it is equivalent to the above pair of program clauses. However, procedurally it avoids the redundancy of attempting to establish C twice, once for the positive proof, and once for the failure proof. In other PROLOGs this procedural advantage has to be achieved by using "/". The clauses are written

P ← C&/&D

P ← E.

The problem with this solution is that the second clause is no longer a true statement.

Set of Solutions

The collection of the set of solutions of a goal clause as the list binding of a variable is another facility that is normally left to the PROLOG programmer to program using "/" and "add clause". In IC-PROLOG this is packaged and expressed using an abbreviated set notation.

(3)

Suppose that the programmer would like all the bindings for the variables x and y for which there a solutions to the goal ← P(x,y,z). This list denotes the set

$$\{\langle x,y\rangle \mid \exists z\, P(x,y,z)\}.$$

In a language such a SETL [Kennedy and Schwartz 1975] such a set extension could be assigned to a variable by writing.

$$w = \{\langle x,y\rangle \mid \exists z P(x,y,z)\}.$$

In IC-PROLOG, the programmer can write

$$w = [t(x,y) / P(x,y,z)]$$

as a shorthand for this equality statement. Its evaluation will result in w being bound to a list of the form.

$$t(a1,b1).\ t(a2,b2)\ldots.t(ak,bk).Nil$$

where

$$x=a1,\ y=b1,\ \ldots\ldots\ldots,\ x=ak,y=bk$$

are all the solution bindings for the goal ←P(x,y,z).

. An example of the use of this set facility is the clause

Courses-taken (x,w) ← w=[y/Takes(x,y)].

Used to evaluate the goal ← courses-taken(Smith,w) it will result in w being bound to the list of all the courses that Smith takes.

Search Control

(4)

Back-tracking is the basic search strategy with the clauses for a predicate selected in the before-after order in which they are listed (they do not need to be contiguous). The programmer can exercise some limited control over the back-tracking.

Firstly, if a relation R is such that some subset of its arguments are uniquely determined by the other arguments, the programmer can express the fact as an assertion about his program. After the successful evaluation of some call involving R in which these other arguments were given, the trace of the evaluation is succeess popped from the stack. This saves space and avoids unnecessary backtracking.

Secondly, the programmer can request indexing of the clauses for any predicate on any argument positions. For each indexed argument a table is constructed. This contains an entry for each constant and each top-level function symbol that appears in that argument position in the consequent atom of some clause for the predicate. It also contains an entry for a variable occurence in that position. When an atom is selected with an indexed predicate the index tables are used to produce a candidate set of clauses. Clauses not in the candidate set cannot unify with the selected atom and the evaluation only backtracks through the candidate set.

Computation rule control

The rule which determines which literal of some derived goal clause is selected for the next evaluation step is the computation rule. To our knowledge all other PROLOGs use a fixed lelf-right computation rule. This gives the programmer very limited control over the order of execution. All that he can do is suitably order the literals in each clause to specify an order of execution for each use of the clause. IC-PROLOG gives the programmer a

(5)

rich set of control facilities. The programmer can make the execution order conditional upon which variables of the clause are bound by the unification that invokes the clause. He can also relax the strictly sequential execution. He can cause the evaluation of particular calls to be coroutined with the evaluation of the preceding sequence of calls, and he can cause a sequence of calls to be evaluated in pseudo-parallel fashion.

Most of these computation rule control facilities are fully described in [Clark and McCabe 1979]. We shall just give one or two examples.

Selection of control alternatives

To make the order of evaluation of the calls of the clause

grandparent(x,z) ← parent(x,y) & parent(y,z)

conditional upon whether x or z is given we write

[grandparent(n?,z) ← parent(x,y) & parent(y,z),
grandparent(x,z?) ← parent(y,z) & parent(x,y)]

in place of the single clause. Bracketing the clauses together prevents their being considered as back-tracking alternatives. The annotation ? expresses the condition that the variable it annotates must have been bound to a non-variable before that clause copy can be used.

Data-flow coroutining

The classic coroutining example for testing whether or not two trees have the same leaf profile can be expressed by a coroutining annotation. In the clause

(6)

same-leaves (x,y) ← leaves(x,u) & leaves (y,u?)

the ? annotation on the u of leaves (y,u?) causes the evaluation of two leaves calls to be coroutined. The trigger for the transfer is the evaluation step that finds a new leaf on either tree. Thus, when the leaves(x,u) evaluation finds the first leaf, there is a transfer to the leaves (y,u) evaluation. This checks the first leaf and continues until it finds the second leaf of y. Just before it would further instantiate u with the second leaf control transfers back to leaves (x,u).

This is because the "?" annotation specifies that the 'flow of data' through u is from leaves (x,u) to leaves (y,u). The leaves (x,u) evaluation now runs until u is further instantiated with the second leaf of x. At this point we transfer back to leaves (y,u) to check this second leaf. The interaction continues in this way until both successfully terminate, or until there is a mismatch of leaves. In this event we benefit from an early failure.

Pseudo-parallel execution

We can also interleave the evaluations of the two leaves calls by using the parallel annotation. We replace the "&" with a"//" and write the clause

same-leaves(x,y) ← leaves (x,u) // leaves (y,u).

When the clause is used the evaluations of leaves (x,u) and leaves (y,u) will be strictly alternated. In this regime either evaluation may bind the shared variable, and one evaluation may run ahead of the other.

There are other annotations which can be used to delay a coroutining

(7)

transfer (a ":" instead of "&"), and to suspend an evaluation that is being executed in pseudo parallel ("!" annotation on a variable). An example of the coroutining delay is a clause written as

$$P(g(u),f(v)) \leftarrow T(u) \ : \ P(u,v).$$

The ":" will delay any transfer of the partial result represented by f(v) until after the evaluation of T(u) is successfully completed. An example of the parallel suspension is

$$R(x,y) \leftarrow Q(x!)R \ldots .$$

If this clause is used during the evaluation of a call P which is being executed as one of a set of pseudo parallel executions, then the execution of P is temporarily suspended until one of the other executions binds x to a non-variable.

File handling

Finally, a word about the file handling of IC-PROLOG. Logically, the transfer of a sequence of characters from one file to another is a relation over strings. This is how it must be described in IC-PROLOG. The programmer gives a set of clauses for the relation R(x,y) that holds between the input string x and the output string y. He then links the x and y to specific files of characters with the system provided predicates. For example, the goal clause.

$$\leftarrow In(x) \ \& \ R(x,y) \ \& \ Out(y)$$

will bind x to the string of characters that <u>will</u> be typed at the terminal and causes the string binding generated for y to be printed on the terminal. By coroutining the Out(y) evaluation, using the annotated

(8)

goal clause

← In(x) & R(x,y) & Out(y?),

the evaluation of the program becomes interactive. Each new line character appearing in y causes a printout of the latest sequence of characters added to y.

References

Clark, K. L. [1978], Negation as failure, in Logic & Data Base, ed Gallaire & Minker, Plenum Press.

Clark, K. L. & F. G. McCabe [1979], Control facilities of IC-PROLOG, in Expert Systems in Micro Electronic Age, (ed D. Michie), Edinburgh University Press.

Clark, K. L., & F. G. McCabe [1980], IC-PROLOG - aspects of implementation, forthcoming CCD research report, Imperial College.

Kennedy, K & Schwartz J. [1975], An Introduction to the Set Theoretical language SETL, Computer Maths with Applications, Vol. 1, pp 97-119.

KLC/FGM/RM

10/6/1980

Some Aspects on a Logic Machine Prototype

Åke Hansson*, Seif Haridi** and Sten-Åke Tärnlund*

(Extended Abstract)

We shall outline some properties of a logic machine prototype under development. We shall take up the logical system of the machine, and focus on its object language (mainly its programming language) and the computation rules associated with it. Finally we comment briefly on a microprogrammed target machine.

Logical system

The logic machine prototype is a part of a programming calculus in which programs can be developed formally (see Hansson and Tärnlund [1979a], [1979b]) and moreover, run efficiently. The latter possibility has been demonstrated by a succession of PROLOG implementations (e.g., see Roussel [1975] and Warren [1977]), which are based on resolution logic (see Robinson [1965]).

Our logical system consists of several parts:

(i) A calculus that facilitates reasoning and derivations of object language programs. The logical system is a first order predicate logic natural deduction system (see Prawitz [1965]).

(ii) An object language for writing programs. This part will be explained in more detail below.

(iii) A meta language which is intended to provide adequate tools for hypothetical reasoning about programs e.g., inserting and deleting hypotheses and strategies. It is also intended to provide control information for running programs efficiently on the target machine e.g., computation rules.

Addresses: * UPMAIL, Computer Science Department, Uppsala University
Uppsala, Sweden.
** Deparment of Computer Systems, The Royal Institute of Technology, Stockholm, Sweden.

Object language

The object language in which we write our programs as functions or relations on a domain of objects consists of sentences which are composed of constant symbols, variable symbols, function symbols, predicate symbols and the logical connectives. There is one special predicate symbol, = , for identity.

The logical signs by which we build up sentences are:
& (and) v (or) - (not) <- (conditional) <--> (equivalence).

The structure of a sentence is e.g., literals Q(...), -Q(...), simple conditional or biconditional statements of literals P <- Q & -R , -P <--> R v T . Only one literal is allowed in the left part of the principal connective.

In comparison to PROLOG we have introduced two new logical signs, "not" and "equivalence". They give us a more balanced system with respect to negated and non-negated statements and thus a bit more clear treatment of negated statements. Moreover, we have introduced identity which gives us a functional notation. Not only are functions important notions for mathematical reasoning, they also give us a control concept. First, by using functions we get information for running such programs deterministically. This will obviate the use of "cut/slash" in most logic programs. Other uses of "cut" will be catered for in the meta language.

Secondly, the execution mechanism assumes, as a default, that function definitions are used to compute results from arguments. This means that upon an activation of a function variables at the argument positions will be bound to a nonvariant instance and variables at a result position will be uninstantiated. Of course, this default can be revoked by writing down control information in the meta language.

Moreover, as we shall see in the section on computation rules, this input-output information implied by the functional notation, will be used to introduce a computation rule similar to the applicative order in functional programming languages.

The following quick-sort program on simple lists illustrates a few concepts in our language. The result of quick-sorting a list x.y is appending a list x.q(y") , where q(y") is the result of quick-sorting y" , to the result of quick-sorting y' if the result of partitioning y with respect to x is y' and y" where y' is a list of the elements on y that are less than or equal to x and y" is a list of the elements that are greater than x . Moreover, the empty list is sorted.

q(x.y) = append(q(y'),x.q(y")) <- partition(x,y,y',y") (1)
q(0) = 0

We omit the procedure for partition for reasons of space.

We have combined functions and relations in this program, but we can take one further step and substitute the relation by a function i.e., partition(x,y) = (y',y") and thus make use of a Cartesian product as a data structure. In fact, we had been in trouble without the Cartesian product since partition can be viewed as a multi-valued function.

Data structures

Our programming calculus consists of axiomatized data structures e.g., simple lists, lists, d-lists, binary trees and Cartesian products. These data structures are given an efficient implementation. In particular many substitution patterns during computations can be compiled into primitive operations on such data structures e.g., assignment of values to the components, except to d-lists where an "occur" check has to be carried out. In addition to these structures it is also possible to use an arbitrary term as data structure. Such data structures can, of course, be axiomatized in the programming calculus for reasons of clarity, program reasoning and running efficiency.

Computational rules

PROLOG has combined a simple top-down and left-to-right computational rule with a more tricky "cut" mechanism which can obscure the meaning of programs. Moreover , the search strategy which is implied by the textual order of clauses, can be exploited for writing programs that fail to characterize the problem to be solved. In contrast, our compuational rules cannot change the meaning of the program, only the efficiency of the computation.

We have two classes of compuational rules. The first class characterizes sequential processing. Operationally it gives a procedural interpreation to the sentences obeying such computation rules (see Kowalski [1974]).

To evaluate a sentence composed solely of relations the body of the sentence (i.e., the antecedent part, if the principal connective is conditional) is evaluated in a left to right order. This is the rule used in PROLOG-based systems and it is our default rule for such sentences.

To evaluate a sentence composed of functions and relations, the literals of the sentence will be partially ordered according to their input-output relation at compile time and then topologically sorted to produce the evaluation order. We call this rule the applicative order of evaluation. For example, an equivalent but a bit different version of our quick-sort program in (1) illustrates this rule.

```
q(x.y) = w <- partition(x.y,y',y") & q(y') = w' &
              q(y") = w" & append(w',x.w") = w
```
(2)

The computation of partition is finished before the computations of q(y') = w'and q(y")=w" are initiated following the topological ordering, then q(y") = w" and finally append(w',x.w") = w are executed. This sequential computation rule is followed independently of the way we write the ordering between these predicates.

The final sequential computation rule is based on the instantiation pattern of the variables of a sentence. This instantiation pattern implies a dynamic topological ordering on the literals of the sentence based on the input-output relation. The difference between this rule and the applicative order of evaluation is that separate invocations of the same sentence may cause different topological orderings due to different patterns of instantiation. This rule requires a run time check, however, it is useful in programs operating on a data base of assertions. A rule similar to this exists in IC-PROLOG, where the programmer ,however, specifies explicitly the alternative order of evaluation (see Clark and McCabe [1980]).

The second class of computation rules characterizes parallelly distributed processes communicating with channels. It consists of a single computation rule called demand driven computation. Operationally an object language sentence specifies an algorithm by a network of communicating processes through unbounded buffers. Networks are constructed by composition and recursion. Our distributed networks are related to the idea of Kahn and McQueen [1977], to streams of Landin [1965] and data flow computation of Dennis [1973]. In contrast, a behavioural approach, i.e., actor computations for logic programs is exploared by Ken Kahn [1980]. We are running programs on a single processor, so our processes behave like co-routines. For example, the following sentence

```
N(x) = y <- A(x) = z & B(z) = y
```
(3)

specifies a simple network of two processes, A and B communicating through a channel z. When N is activated two process instances for A and B will be created with the channel z used as a communication link. A is called a producer for z and B is a consumer of z. In every such network there is one process (here B) which is the initial demander and drives the whole network. The computation proceeds as follows: B runs until it needs a partial "result" via z, the control is then transferred to A and then returned to B at its point of suspension.

It is to be noted that such networks may evolve dynamically under a computation. When we prefer to think of a computation as a process on infinite data structures (streams), on which no process can give a complete result, demand driven computations can give satisfying partial results. A special data structure (dynamic data structure) is introduced in POP-2 (see Burstall, Collins and Poppelstone [1971]) for this purpose.

We take up an example from Kahn and McQueen [1977] and give a program that computes the prime numbers according to Eratosthenes' sieve on streams in a functional notation that illustrates a demand driven computation.

```
sift(integers)    Our goal is to compute this stream!
integers=increment(1)
increment(n)=n+1.increment(n+1)
sift(p.q)=p.sift(filter(p,q))
filter(p,n.q)=n.filter(p,q)  <- mod(n,p)≠0
filter(p,n.q)=filter(p,q)  <- mod(n,p)=0
```

where the computation rule of sift is demand driven.

Process networks can also be cyclic which make them different from lazy evaluations for LISP (see Morris and Henderson [1976]). Cyclic networks are useful in some programs. To illustrate this we take up an example from Dijkstra [1976] where we shall generate elements of a stream of the form $2^a 3^b 5^c$ where a, b, c ≥ 0 , moreover, the elements shall be generated in increasing order without omission or repetition.

We have:

```
merge(x.y,u.w)=x.merge(y,u.w)  <- x<u
merge(x.y,u.w)=u.merge(x.y,w)  <- u<x
merge(x.y,u.w)=x.merge(y,w)  <- x=u
times(x,y.z)=x*y.times(x,z)
```

We want to compute a stream in a demand driven fashion that we write as follows:

```
y=1.merge(times(2,y),merge(times(3,y),times(5,y)))
```

In addition, to the demand driven computational aspects of this program it is an example where resolution systems fail to compute the stream of elements due to the occur check in the unification algorithm. Unfortunately, without this check the system becomes inconsistent.

In a sentence whose execution is controlled by a demand driven rule one demanding process will start the execution of the body of the sentence. If the network is cycle free any non-producer consumer can be the demanding process. If the network is cyclic, with one cycle or an overlapping number of cycles then any process in such cycles can be the demanding process. If the network is composed of disjoint cycles with one cycle acting as a producer to one or more cycles, then there must be at least one cycle which acts as non-producer consumer. Any process in such a cycle can be the demanding process. The only restriction we have is that there can be only one designated producer for any variable. This is reasonable since a problem would arise in deciding which producer should grant a demand.

A situation may arise where many consumers may demand a partial result from a single producer. This may only occur when the producer of a result has local consumers. In this situation the consumers will be resumed in an inner-most to outer-most order before the procedure is reactivated.

```
S(x)=z <- Po(x)=y & Q1(y)
Po(x)=y <- P1(x)=y & Q2(y)
   .
   .
   .
P1(x)=y <- P2(x)=y & Q3(y)
```

where in all sentences above y is a channel variable, Q1, Q2, Q3 are consumers of the same channel y, Q2, Q3 are local consumers of the producer Po. When Po produces a partial result of y on behalf of P1 and P2 then Q3 will be activated, then Q2 and finally Q1. This is implemented by chaining the consumers Q1, Q2 and Q3.

The target machine

Our logical system is being implemented on a microprogrammed computer V77 with a microprogram control store of 4K x 64 bits. The machine belongs to the class of horizontally microprogrammable machines which permits several microoperations to be executed in parallel. It is our intention that the system will be used as a personal computer. A virtual memeory management system is being built at the lowest level, which permits user jobs to have a large address space of 2 exp 24 bytes.

References

Burstall R. & Collins J. & Poppelstone R. [1971] Programming in POP-2, Edinburgh University Press

Clark K. & McCabe F. [1979] IC-PROLOG reference manual, CCD Research Report, Imperial College, London

Dennis J. [1971] On the Design and Specification of a Common Base Language, in Computers and Automata, Brooklyn Polytechnic Institute

Dijkstra E. [1976] A Discipline of Programming, Prentice Hall, Englewood Cliffs, N.Y

Hansson Å. & Tärnlund S-Å. [1979a] A Natural Programming Calculus, Proc. IJCAI-6, Tokyo

Hansson Å. & Tärnlund S-Å. [1979b] Derivations of Programs in a Natural Programming Calculus, Electrotechnical Laboratory, Tokyo

Kahn G. & McQueen D. [1977] Coroutines and Networks of Parallel Processes, IFIP-77, North-Holland Publ. Company

Kahn K. [1980] Intermission - Actors in PROLOG, Logic Programming Workshop, John von Neumann Computer Science Society, Hungary, 14-16 June

Kowalski R. [1974] Predicate Logic as Programming Language, Proc. IFIP Congress 1974, North-Holland Publishing Company, Amsterdam

Landin P. [1975] The Correspondence between ALGOL 60 and Church's lambda notation: Part 1, Communications of the ACM, vol.8, no.2

Morris J. & Henderson P. [1976] A Lazy Evaluator, Proceedings of the Third ACM Conference on Principles of Programming Languages

Prawitz D. [1965] Natural Deduction, A Proof-Theoretical Study, Almqvist & Wiksell, Stockholm

Robinson J.A. [1965] A Machine-Oriented Logic based on the Resolution Principle, JACM 12, 1, January

Roussel P. [1975] Prolog: Manuel de Reference et d'Utilisation, Groupe d'Intelligence Artificielle, U.E.R. deLuminy, Marseille

Warren D. [1977] Implementing Prolog - Compiling Predicate Logic Programs, Dept. of Artificial Intelligence, No 39, Edinburgh

TWO SOLUTIONS FOR THE NEGATION PROBLEM

Verónica Dahl
Facultad de Ciencias Exactas
1428 Buenos Aires, Argentina

In logic programs, negation has traditionally been defined by default, i.e., by considering false any fact whose truth can not be demonstrated. This convention, while saving numerous explicit representations of facts that, under certain circumstances, can just as well be deduced, might have the undesirable side effect of not always leading to reliable answers. This article examines the conditions under which a logic program with negation by default is likely to produce all the expected answers, and proposes two ways of automatically ensuring that this is always the case.

1- INTRODUCTION

The problem of how to represent negative facts in logic programming has traditionally been solved by stating that those facts whose truth can not be proved are false. This rule is appropriate for many applications: those in which our knowledge about the domain being represented is complete. Whenever this assumption -known as the closed world assumption- is correct, the explicit representation of negative facts becomes redundant, since in a two-valued logic these can simply be established by default.

R. Reiter has discussed this assumption with respect to data bases[12], and shown that it may lead to inconsistencies for non-Horn clause data bases, but not for Horn-clause ones. In [4] we proposed a three-valued version of the closed world assumption, suited for natural language applications. K. Clark has investigated the relationship between negation's implementation by default and its truth-functional semantics [1], by explicitating the closed world assumption into "if and only if" definitions, as opposed to the classical "if" procedures of logic programs.

These discussions, however, are independent from the particular proof strategy chosen and are therefore not concerned with some of the operational problems that negation by default might cause with respect to an actual logic program interpreter.

In this article we examine Prolog's [14] negation by default, we show that dealing with closed worlds does not in itself guarantee it to behave as expected, and we propose

two alternative ways of solving this problem. Both solutions have been developed within natural language consultable data base systems written in Prolog [5 6], but they are general enough to serve in other programs as well, and therefore fit into a larger picture, involving higher level ways of querying logic programs.

Because of space limitations, we shall mostly deal with our subject informally. Some of the related aspects are covered more formally elsewhere [4 7 8]. Our terminology is slightly biased towards data bases, but this is only a matter of convenience (indeed, we share R. Kowalski's view that the distinction between data bases and ordinary programs is not a useful one [10]). Some knowledge of the procedural interpretation of Horn clauses [11 13 15] is assumed.

2- PROLOG'S STRATEGY

Given a query $-G_1 \ldots -G_n$, and a set P of procedures, Prolog selects the first procedure $+A_1\ -A_1 \ldots -A_m$ in P such that A and G_1 match with a most general unifier θ, and derives the new query: $(-A_1 \ldots -A_m\ -G_2 \ldots -G_n)\theta$. This process is repeated on each new query, always selecting the first literal to be resolved upon. Alternative matching procedures in P are tried in the order in which they were stored, with exhaustive backtracking upon failures. Efficiency is therefore influenced both by the order in which the literals appear in a query, and by the order in which the alternative procedures for each predicate are stored.

Notice that, by choosing always the leftmost literal, Prolog systematically produces all those resolvents springing from the first literal in a query, and only turns to the second one when and if it has succeeded in evaluating this first literal (i.e., in replacing it by an empty set of literals).

Execution can be further controlled by the programmer, through Prolog's extra-logical features. The predicates used for control are: "VAR(x)", used to test whether x is a "free"(1) variable or not, and the 0-ary predicate "/", serving to limit the non-determinism that arises from different procedure choices.

---

(1) Prolog's concept of a "free" variable is a dynamic one: at a given point in the resolution process, a variable is "free" if it has not yet been substituted for a function term. From now on, we shall use inverted commas whenever we refer to this meaning.

"/" always evaluates to true and has the side effect of disqualifying certain alternative procedures, so that they will not be considered upon a backtrack. For instance, given the procedures

(1) +P(x) -R(x)....-T(x) -/ -Q(x)....-S(x)
(2) +P(x) .....
(3) +P(x) .....

(N.B. Variables are underlined throughout the paper)

and the query -P(a), Prolog first selects procedure (1). Two cases arise: if the literals preceding "-/" can all be evaluated, Prolog tries to evaluate the remaining literals in the clause, but, upon a backtrack, is inhibited from trying the remaining choices for P,R,...T. If, on the other hand, (1) fails before "-/" is selected, (2) and (3) can be tried next.

3- THE NEGATION PROBLEM

Prolog's negation operator implements the rule "take every unprovable fact as false", as follows:

(1) +NOT(p) -p -/ -FAIL
(2) +NOT(p)

Here "FAIL" is a predicate for which no corresponding procedures are defined, and whose evaluation will therefore provoke a failure. Thus, procedures (1) and (2) above can be read: "if -p can be evaluated, NOT(p) is false. Otherwise, NOT(p) is true.

Let us now consider a logic program comprising procedures (1) and (2) above plus the following:

(3) +ACTOR(A)
(4) +ACTOR(B)
(5) +SINGER(C)
(6) +SINGER(D)

With respect to this (closed world) program, it is easy to verify that the queries -NOT(ACTOR(C)) and -NOT(ACTOR(A)) evaluate to true and false respectively, which is what we would expect. But, faced to the query:

Q_1= -NOT(ACTOR(x))

(i.e., "find an x who is not an actor"), for which one could expect the answers "C" or "D", Prolog would not come up with any answer at all. It would successively generate the resolvents :

Q_2= -ACTOR(x) -/ -FAIL (from Q_1 and (1))
Q_3= -/ -FAIL (from Q_2 and (3))

At this point, the evaluation of -/ would eliminate (2) and (4) as alternative procedure choices, and the remaining query (Q_4= -FAIL) would fail.

The fact is that Prolog's negation operator is only safe to apply on predicates with ground arguments (terms which contain no "free" variables). In other words, it can safely verify the truth of proposed answers, but not generate an answer.

Since ground terms in a given query can not cause the negation problem, we shall call them safe. This is also the case for those variables with no occurrences inside a literal of the form -NOT(P). A variable occurring in the scope of a negation, however, must be considered in its dynamic behaviour.

We shall say that a variable x occurring inside a literal N=-NOT(P) within a query Q is safe in Q whenever it is certain to take a ground value before N is evaluated. A query Q is said to be safe whenever each of the variables it contains is safe.

In Prolog, the programmer can arrange for queries to be safe through a particular ordering of the literals. For instance, let us suppose that the evaluation of -P(x) substitutes a ground term for x. Then it is safe to query:

-P(x) -NOT(Q(x))

because, although the argument of "NOT" is not ground at the beginning, Prolog's strict left-to-right strategy assures that it will take a ground value before -NOT(Q(x)) is selected.

On the other hand, the unsafe but declaratively equivalent query:

-NOT(Q(x)) -P(x)

will always fail (except, of course, in the unlikely case that no procedures for Q exist).

In this example, query reordering is rather simple, but it can become difficult in real applications, where efficiency is often crucial, since it forces the programmer to remember the run-time behaviour of each predicate. This, in turn, varies according to the input/output role of the arguments: a query of the form -P(a,y) will behave differently than another of the form -P(x,b).

Prolog's negation operator, therefore, while saving a lot of time and trouble in the development and execution of a closed world program, introduces extra trouble in querying it and increases the opportunity for error.

With respect to data bases this problem is particularly acute, since the user must not be expected to know how to program. He should instead be provided with a flexible query language -preferably, natural language. But even programs not conventionally regarded as data bases should ideally accept casual users, so the problem we are dealing with is a rather general one.

The two solutions we describe here were implemented in combination with natural language interfaces translating a user's query into a logical formula. We shall next try to show how these systems solve the negation problem by an appropriate evaluation of the formulas generated by the natural language analysers.

4- COROUTINING

Our first solution is based on a very simple idea: the evaluation of a literal of the form -NOT(P) inside a query should be blocked until P becomes ground.

We can achieve this either by reordering the query so that all negations appear in the end, or by altering Prolog's strategy, so that the selection of a literal is made to depend upon its dynamic merit rather than upon the static ordering of the query.

The latter alternative corresponds to computation with coroutines: instead of always completing the evaluation of the first literal, any literal can be selected, resolved upon (i.e., partially evaluated), and its descendants may eventually be kept waiting while attention shifts to other literals with a higher merit. This approach, while allowing in particular to postpone the evaluation of non-ground negated literals, can also be exploited to improve efficiency and to simplify the posing of queries, as we shall see later.

4.1- Implementing coroutines in Prolog

One way of implementing coroutines in Prolog itself is by defining a Prolog predicate -say, PROVE(Q)- capable of selecting any literal in a query Q and of performing

just one derivation step at a time. For commodity, a query $-G_1\ -G_2\ \ldots -G_n$ can be noted as a list $G_1.G_2\ldots G_n.nil$ (where "." is a Prolog-defined binary operator in infix notation), and procedures to be coroutined can be noted through a special predicate, e.g. as

$$+AXIOM(A,A_1.A_2\ldots A_m.nil)$$

instead of $+A\ -A_1\ \ldots -A_m$.

Then the prover can easily find a matching procedure for a selected atom G_k by querying: $-AXIOM(G_k,\underline{l})$, and then replacing G_k by the retrieved list of atoms $\underline{l}$.

Thus, just one derivation step is performed, and the prover regains control over the new query. Any of its atoms might be selected next.

The selection of a predicate to be evaluated must depend both on the run-time state of the variables in the query and on the particular definitions chosen for each predicate, which determine its run-time behaviour. General guidelines for this selection must be therefore set up by the programmer. This can be done through a special predicate of the form DELAY(P,n), stating under which conditions a predicate P is to be delayed n derivation steps.

For instance, an efficient evaluation order for queries such as:

$$-PRICE(\underline{s},\underline{p})\ -SYSTEM(\underline{s})\ -HAS(\underline{s},FORTRAN)$$

(What is the price of a (computer) system having Fortran?) can be obtained simply by defining:

1)+DELAY(PRICE($\underline{x}$,$\underline{y}$),0) -GROUND($\underline{x}$) -/
2)+DELAY(PRICE($\underline{x}$,$\underline{y}$),∞)
3) +DELAY(SYSTEM($\underline{x}$),1)
4) +DELAY(HAS($\underline{x}$,$\underline{y}$),0)

where GROUND is an auxiliary predicate evaluating to true when $\underline{x}$ contains no "free" variables, and ∞ is a sufficiently large integer. These definitions ensure that an appropriate system is produced before attempting to calculate its price, and that Fortran is included before any other component of the system, which is then completed accordingly.

The prover's evaluation of a query Q can now be summarized as follows: Q is successively scanned, looking first for 0-delayed predicates, then for 1-delayed ones, and so on.Each time one such predicate is found, AXIOM is used to perform one derivation

step and the next scan's delay is set to 0 -since this partial evaluation might have altered the run-time delay value of any other predicate. If none is found in a given scan, the searching delay value is incremented by one for the next scan. This process ends successfully when Q has been transformed into "nil".

Notice that, within this coroutining system, the negation problem can be solved simply by blocking non-ground negated literals through the DELAY predicate. Our actual treatment of negation, however, is somewhat subtler: our prover can use negative information to improve efficiency, and also accepts more flexible queries.

4.2- Coroutining logical formulas

Our Prolog implementation of coroutines was designed for a data base system representing the software and hardware catalogues for the French series of computers SOLAR 16 [6]. A user can ask this system questions in French, ranging from simple ones involving retrieval of stored facts, up to requests to assembly and describe computer configurations suiting his particular needs. The analyser uses a modified version of a metamorphose[2] grammar written by A. Colmerauer[(2)], which translates restricted French queries into closed logical formulas, whose syntax is defined recursively by:

a) if p is an atom, p is a formula.
b) if f_1, f_2 and f are formulas, then NO(f), EXISTS($\underline{x}$,f), EVERY($\underline{x}$,f), AND(f_1,f_2), OR(f_1,f_2) and IMPLIES(f_1,f_2) are also formulas.

These formulas can be converted into a list of queries to be coroutined, by successive application of appropriate rewriting rules. This task can be left to the prover itself, by considering formulas as 0-delayed predicates and using AXIOM rules such as:

(1) +AXIOM(EXISTS($\underline{x}$,$\underline{f}$),$\underline{f}$.nil) -/
(2) +AXIOM(EVERY($\underline{x}$,$\underline{f}$),NO(EXISTS($\underline{x}$,NO($\underline{f}$))).nil) -/
(3)+AXIOM(AND($\underline{f}_1$,$\underline{f}_2$),$\underline{f}_1$.$\underline{f}_2$.nil) -/
(4) +AXIOM(NO(NO($\underline{f}$)),$\underline{f}$.nil) -/

which precede the AXIOM representations of all the procedure definitions in the data base.

4.3- Using negated formulas to improve efficiency

In constructive programs such as the SOLAR 16 one, some negative requirements of the user can be exploited to reduce the search space, and should therefore be given priority even when applied on non-ground arguments. For instance, a request for the charac-

---

(2) Personal communication, 1977.

teristics of a system which does not include Fortran should first forbid the addition of Fortran and then complete the system accordingly. This can be done for instance through the rule: +AXIOM(NO(HAS(x,y)),FORBID(x,y).nil) -/ , where FORBID is a 0-delayed data base predicate, which somehow marks in x the mandatory absence of y.

Let us now consider a formula f containing a subformula NO(f') which can not be further transformed. This fact can be registered by renaming NO into NEG once that none of the formula-rewriting AXIOM rules has been found to match (i.e., by adding after these rules a rule of the form: +AXIOM(NO(f),NEG(f).nil). Notice that the subformula f' could be coroutined if we simply replaced it by a recursive call on the demonstrator: -PROVE(f'). But some of the variables in f' may well also appear outside it, which would mean that some of the predicates concerning these variables are being delayed outside f'. Since f is assumed closed, this case arises whenever a variable x is free in f' (its quantifier appears outside the negated subformula).

Therefore, in order to evaluate NEG(f'), the prover first calculates the list of the non-quantified variables inside f'. If it is empty, it evaluates NOT(PROVE(f')), where NOT is Prolog's negation by default. Otherwise it just keeps NEG(f') waiting and continues the current scan of f.

Thus, our prover can handle a subtler treatment of negation than the mere delaying of non-ground negated formulas, but careful coroutining specifications must be designed for each problem domain.

4.4- Some performance considerations

The pre-processing of a query by the demonstrator, although taking some time, increases the overall speed, because in many cases coroutining reduces the search space. On the other hand, those predicates not needing to be coroutined can be handed over to Prolog directly. This can be achieved by: a) defining their delay value as 0 (a single default rule suffices), b) writing non-coroutined procedures in the usual Prolog notation instead of through AXIOM, and c) adding as the last AXIOM rule: +AXIOM(p,nil) -p. Thus, if an atom does not match in any of the preceding rules, it is immediately evaluated by Prolog and eliminated from the prover's list.

The search for a matching procedure, however, can be time-consuming due to our special notation: most Prolog versions have direct access to the set of procedures corresponding to a given predicate, but this is obviously lost when all the predicate definitions are packed up within a single "super"-predicate like AXIOM or DELAY.

Fortunately, more recent research on Prolog implementation has eliminated the overhead caused by searching through all the AXIOM and DELAY rules, by providing quick direct access to subsets of procedures whenever their first argument is either a constant or a function term [16].

5- A SET-ORIENTED QUANTIFIER MECHANISM

Our second solution to the negation problem was devised for a less sophisticated but more general data base system, also programmed in logic [5]. It is a more restricted solution than the previous one in the sense that it has no coroutining facilities and is only applicable to programs that represent finite worlds, but it accepts definitions of fairly arbitrary data bases with less effort from the programmer.

This solution has also been worked out in combination with a natural language interface, which translates Spanish queries into a more evolved, three-valued logical system, that we shall call L3 [8]. Some details regarding this translation can be found in [7], and related work for French is [3]. L3's features, which were designed to meet not only natural language processing but also data base development requirements, are discussed from the latter point of view in [4]. Those characteristics of L3 related to our treatment of negation can be summarized as follows:

- a formula in L3 can take the value true, false or undefined. Owing to this feature, the evaluation of a formula NOT(f) must be subtler than previously described, but still follows the negation-by-default idea. Briefly, NOT(f) is considered false when f evaluates to true, undefined when f evaluates to undefined, and true when f evaluates to false. A formula f evaluates to false when all attempts to prove that it is true or undefined have failed.
- each variable in a formula is typed, i.e., associated with a finite domain or semantic type.
- sets are represented within formulas, either extensionally (through a term of the form: $a_1.a_2...a_n.nil$) or intensionally. In the latter case, a subformula of the form "those($\underline{x}$,f)" is used to represent the set of all those $\underline{x}$'s in $\underline{x}$'s associated domain which satisfy f. The evaluation of such a formula always yields the extensional representation of that set. The "those" formula is the only "quantification" allowed.
- the meaning of natural language quantifiers, including any presuppositions they might induce, is represented within a formula through special sub-formulas such as "card(s)", which stands for the cardinality of the set s. For instance, the sentences:

"Twenty students registered" and "Some students registered" are respectively represented through the formulas:

equal(card(those(x,and(student(x),registered(x)))),20)

greater-than(card(those(x,and(student(x),registered(x)))),1)

where x is typed by the student domain.

The basic idea in this second solution is to use set representations as the main data structure, and to make sure that any set designated in a query is evaluated (i.e., replaced by its extensional representation) before the evaluation of any predicate concerning it takes place. This, in particular, ensures negated predicates to be evaluated only when all their arguments are known. If, in turn, the evaluation of a set can also be made safe, the negation problem is solved.

In our finite, typed world, there is an extra-logical but Prolog-feasible way of evaluating a THOSE (x,f) formula such that the designated set S is obtained without ever having to evaluate f with x "free". It consists in successively evaluating f for each value of x in its associated domain, and concatenating those who satisfy it into a list representing the set S.

With this evaluating sequence, it is obvious that any quantified variable is safe within its quantifier's scope. If, moreover, it has no free occurrences in a given formula f, it is obviously also safe in f, since it is safe in the only subformula in which it occurs. Therefore, by allowing only closed formulas (formulas with no free variable occurrences), we can ensure their safeness. Our system solves the negation problem by generating only closed formulas in L3, and evaluating quantified subformulas as described above.

Of course, the implementation of this solution involves a careful mapping of Prolog's two-valued logic into the semantics of L3, which includes not only three logical values, but also set-handling operations. We have done this in such a way that the user needs only define which tuples make each relation true (in terms of either sets or individuals, or both, as he may prefer), and leave the system to deduce from these definitions which make it false or undefined. We thus obtained a higher level solution than the previous one, in the sense that it both leaves less work to the programmer and has more linguistic power (the third logical value allowing it to detect false natural language presuppositions), but, lacking coroutines, it can only deal efficiently with less complicated worlds.

6- CONCLUDING REMARKS

We have shown how Prolog's negation by default might induce some operational anomalies with respect to non-safe queries, and we have proposed two ways of solving this problem: an appropriate coroutining system and a set-oriented quantifier mechanism.

The first solution is adequate for "constructive" data bases such as the SOLAR 16 one, where different modules can be combined to form particular configurations adapted both to the user's requirements and to the specific construction rules stored in the data base. In return for the power obtained, however, the programmer must learn how to use the coroutining system. This is certainly a lighter task than the one of finding ad-hoc solutions for each problem that would need coroutining, but it would still be desirable to generalize it further, so as to provide higher level language features -or, in data base terminology, greater data independence.

Coroutines should form part of a more powerful interface allowing the sequential simulation of time-independent patterns of thought, in a manner invisible to the programmer. Ideally, he should be able to describe in n.tural language sentences the world he wants represented, and leave all efficiency considerations to that interface, just as, in the consultation level, our solution relieves him from all sequencing concerns- in particular, those relative to negation.

Our second solution works in a less ambitious context and can therefore afford to be more general, allowing an easy and straightforward definition of finite but fairly arbitrary closed world data bases. This has been achieved mainly through a careful study of the syntax and semantics of natural language queries on one hand, and of the Prolog implementation of set primitives on the other.

More general solutions to the negation problem, in our view, must necessarily be concerned with higher-level ways of querying and defining logic programs, and should include a set axiomatization that can be efficiently combined with coroutines. Work is under way in this direction [9], and should allow complex worlds such as the SOLAR 16 one to be defined in terms of a few "primitive" predicates describing sets and suitable ways of combining them, in a manner totally independent from exactly how this is to be done.

References

1. Clark, K. Negation as failure. In: Logic and data bases, Plenum Publ. Co, 1978.
2. Colmerauer, A. Grammaires de métamorphose. In: Natural language communication with computer. Lecture Notes in Computer Sciences, Springer-Verlag, 1978.
3. Colmerauer, A. Un sous-ensemble intéressant du francais. R.A.I.R.O. vol. 13, N° 4, 1979.
4. Dahl, V. Logical design of deductive, natural language consultable data bases. Proc. V International Conference on Very Large Data Bases, Rio de Janeiro, 1979.
5. Dahl, V. Un système déductif d'interrogation de banques de données en espagnol. Thèse de Doctorat de Spécialité en Intelligence Artificielle, Univ.d'Aix-Marseille II, 1977.
6. Dahl, V. and Sambuc, R. Un système de banque de données en logique du premier ordre, en vue de sa consultation en langue naturelle. D.E.A. Report, Univ. d'Aix-Marseille II, 1976.
7. Dahl, V. Quantification in a three-valued logic for natural language question answering systems. Proc. Sixth International Joint Conference on Artificial Intelligence, Tokyo, 1979.
8. Dahl, V. A three-valued logic for natural language computer applications. Proc. Tenth International Symposium on Multiple Valued Logic, Illinois, 1980.
9. Dahl, V. Towards constructive data bases. Univ. of Buenos Aires (forthcoming report).
10. Kowalski, R. A. Logic for data description. In: Logic and data bases, Plenum Publ. Co., 1978.
11. Kowalski, R. A. Logic for problem solving. North-Holland, 1980.
12. Reiter, R. On closed world data bases. In: Logic and data bases, Plenum Publ. Co, 1980.
13. Robinson, J. A. A machine-oriented logic based on the resolution principle. JACM vol 12, 1965.
14. Roussell, Ph. Prolog: manuel de référence et d'utilisation. Univ. d'Aix-Marseille II, 1975.
15. Van Emden M. H. Programming with resolution logic. In: Machine Intelligence 8, 1977.
16. Warren, D. and Pereira, L. Prolog- the language and its implementation compared with Lisp. ACM Symposium on Artificial Intelligence and Programming Languages, 1977.

A CONTROL METALANGUAGE FOR LOGIC PROGRAMMING

Hervé GALLAIRE and Claudine LASSERRE

ABSTRACT:

In this paper we present a revised position on a metalanguage built for expressing control knowledge on the derivation of information in a logic program. The metalanguage is itself a logic language; the user expresses his metarules separately from his program clauses; he can specify both clause selection and literal selection metarules; these metarules can be specific of his problem or they can express general strategies. This approach is examined in the light of other approaches to metaknowledge expression.

Keywords: Metaknowledge expression, control of deduction process, logic programming, problem solving.

Address : ENSAE -CERT Complexe aerospatial -
31055 TOULOUSE Cedex FRANCE

Part of this research was financed by CNRS under ATP N° 4270.

1. INTRODUCTION

In a previous paper [GAL] we defined a metalanguage for controlling the derivation process in a Horn-clause programming language. This control metalanguage (CML in short) was inplemented in top of PROLOG [ROU] and its interpreter coded in PROLOG itself. In this paper we first review and update our position on CML. Next we present a revised version of a CML which is to be implemented and which will be used in connection with intelligent backtracking techniques not presented here. The CML exposed here extends that of [GAL] both in its syntax and in its semantics. A discussion of both aspects is given in section 3 where the interpreter is sketched too.

2. ISSUES IN METAKNOWLEDGE EXPRESSIONS

The following statements recapitulate our positions on CML :

- The types of CML we advocate are logic programming languages whose predicates (i.e the meta-level predicates or metapredicates) are given a fixed interpretation in terms of the behavior of the logic interpreter.

- The CML statements and the world description statements are separated from each other;this is contrary to some other viewpoints, as for instance Clark and McCabe [CLA] , or Pereira [PER]. They both mix knowledge i.e world description and metaknowledge i.e contol primitives. The one advantage of that approach over ours is likely to be in the efficiency of interpretation; but we feel it is a step which clobbers logic programming when used in a systematic manner; its expressive power is clearly more restricted than ours if only for beeing local to a logic statement thus not capable to bear on clauses themselves.
- The kind of metaknowledge we want to express is not the same as that illustrated by Bundy [BUN] or Dincbas [DIN]. Namely, as recalled above, we only talk in terms of interpreter behavior, while their metaknowledge expression is built strictly in terms of world description, through hierarchisation of levels of perceptions of the world (such as objects, assemblies, equations, methods, heuristics, ...). They are led to build their own interpreter as long as no general agreement is reached on a world structuring language. In this respect they can be put in parallel with all the efforts in knowledge representation language definition and implementation such as KRL [BOB].

We are now to briefly present a refined CML which extends the capabilities of [GAL]. This presentation is based on A. Fahmi's thesis [FAH].We shall also discuss this language in terms of the above mentionned alternatives.

3. A REVISED CONTROL METALANGUAGE

We shall discuss it from its syntax and semantics viewpoints successively.

3.1 Syntax of metarules

The syntactic modifications which have been introduced allow to express meaningful actions in terms of the derivation process and to have several ways to indicate which objects (i.e clauses or literals) are involved in the metarules.

The general form of a metarule is :

Action(t1, t2, .., tn) ⟵ Condition1 ∧..... ∧Conditionk

A metarule is written in order to describe an Action on the interpreter behavior whenever the interpreter focuses its attention on an object which is involved in that metarule. Both "t" and Condition parameters may be partially instantiated or variables; Condition is a literal which is either system-defined or user-defined. The set of objects involved in a metarule is obtained from a combination of direct and/or indirect selection through the "t" terms and through the Condition arguments.

- a literal P or a clause P⟵Q is directly designated by any literal P' which is less instantiated than P and which unifies with P. Of course P' being an argument in a literal is logically a term;

- an indirect designation of a clause may be either content-directed or position-directed :

  OPORDER(P(x,y), n1.n2...)⟵ C1 ∧ C2∧..∧Ck
  will select clauses numbered n1, n2, .. in that order for attempting resolution of literals matching P(x,y) with the above mentionned restrictions;any such literal is directly designated and is involved in the metarule;

  OPBEFORE(R(x,a), t1, t2)⟵ CLAUSE(t1, z, P(x,y))∧
  CLAUSE(t2, r, Q(x,y))
  will select for any literal R(x,a) clauses containing P(x,y) before clauses containing Q(x,y) ; here t1 and t2 are used to link both parts of the metarule; z and y are variables naming the clauses; thus in this case this metarule would apply to any clause whose head matches R(x,a); of course it would be possible to restrict its application to some clauses by appropriately instantiating z and r;

- an indirect designation of a literal is given by :

  LITERAL(x, name, list-of-properties)

  where x links this designation to other parts of the metarule, as t1 and t2 above, where name will select the literals for which that metarule was designed, and where list-of-properties is a list of (Pi : Vi). The properties Pi are to be taken in a set of predefined symbols :
  ANCESTOR, FATHER, DEPTH, SOLVED, ... :

BEFORE(t1, t2) ← LITERAL(t1, x, DEPTH: n1) ∧
LITERAL(t2, y, DEPTH: n2) ∧
INF(n1, n2)

will impose a breadth first strategy to the interpreter;further restrictions on x and y would impose such a strategy only to the involved literals

Clearly this syntax is rich enough to express any designation of literals and clauses; whether the set of properties is sufficient or not remains to be seen; in any case the user can define his own conditions built with his problem literals as well as with other system literals, such as the Ancestor predicate and the Var predicate in PROLOG.

3.2 Semantics of metarules

Metapredicates are provided for clause selection and literal selection. By selection we mean either electing a candidate among others or eliminating a candidate.

3.2.1 Clause selection

Let S be the set of clauses that could be used for resolution with the active i.e selected literal. Then a metarule for clause selection can give information on some of the clauses in S (namely those clauses that it designates), expressing precedence between them, excluding some of them, expressing mutual exclusion between them. As this is straightforward, it is not discussed further here. Example :

OPBEFORE(t1, t2, t3) ← CLAUSE(t2, x, n1) ∧
CLAUSE(t3, y, n2) ∧
INF(n1, n2)

allows to express that clauses are to be selected according to their number of literals (shorter clauses first) for any literal t1.

Such a type of control expression is outside the scope of expression of the [PER] type of approach (see 2 above).

3.2.2 Literal selection

These metarules express how to choose in a resolvant clause the next literal to be solved; they provide for the following :

- turning off metarule control during the proof of a literal;this is an essential characteristic in a realistic environment;
- priority of a literal over another;
- restricting the attention of the interpreter to the selected literal and to its descendants until it is proved;
- recovering the space corresponding to the proof of a literal;

- unsolvability of a literal;
- uniqueness of solution of a literal;
- recursion level limitation;
- inhibition of a literal on backtrack;
- necessary (NEED) and sufficient (READY) conditions for a literal to be selected.

The NEED and READY metarules are quite related to the producer-consumer primitives given in IC-PROLOG. NEED expresses that a literal must have some resources available before it can be selected; it corresponds to a lazy consumer in IC-PROLOG terms [CLA]. Similarly READY expresses that if the resources are available, the literal must be selected as soon as possible; this notion can only be made clear by giving its meaning in terms of the interpreter behavior; thus it contrasts with NEED which is more simple to understand and to use. To some extent READY corresponds to an eager consumer in IC-PROLOG, without the inheriting property, which we feel makes it too difficult to visualize the interpreter process. Of course there are default activations for literals. Finally there is a way of expressing a producer requirement using a NEEDBY predicate which says that t1 needs resources which, if lacking, will be produced by t2, thus asking for selection of t2 :

NEED(Filter(x,y,z)) ← INST(x)
NEEDBY(t1, t2) ← C1 ∧ C2 ...

If we try to relate this approach to that examplified by [CLA] or [PER] it is difficult to discuss the expression power issue because their proposals are not intended to be complete; it seems it would be difficult for them to express priority of a literal over another when no producer-consumer relation holds between them; there may be other significant differences in power but they do not seem to be crucial; more important to us is the fact that logic programs themselves have to be modified. But we can take from their approach the idea to compile the literal selection metarules, thus bystepping the metarule selection process itself and getting modified logic clauses as theirs. This would need a modified interpreter which seems to be compatible with the interpreter for metarules.

3.3 Interpretation process

The interpreter to be built is rather straightforward; it is driven by literal expansion which calls upon metarules; only the literal selection process must be defined with care due to the READY type metarules. An automaton can be used to express the state transition in the interpreter, where the state notion is defined from the state of literals in the resolvant at each step of the derivation process. Roughly state transition of literal is done according first to priority metarules , then to READY metarules and finally to NEED metarules.

One should also pay attention to the unification process between literals in the resolvant, in the clauses, and in the metarules; while the first unification can be unrestricted it is not the case for unification with the metarule; further no indirect instantiation produced by the Condition part evaluation should occur. Thus the standard unification module should be modified so as to have several calling sequences.

We have indicated above a possible compilation of literal metarules. In the same spirit, it is very much conceivable to build an interpreter which would compile links between clauses and metarules thus making it much easier and more efficient at deduction time. This would require the unification process just described and nothing more.

4. CONCLUSION

In this paper we have given an overview of a control metalanguage and discussed some of the related issues. To the above discussion we want to add the following :

- this approach can accomodate specific i.e problem-related strategies as well as general strategies; we gave examples of general strategies much in the spirit of [MIN]. This is due to the possibility of tailoring metarules to their goals with much ease;

- as far as specific metaknowledge language approach is concerned (see 2 above) it should be clear that it is quite complementary to ours and that our proposal could be adapted to interpreters such as the ones they build, although many of our constructs would be redundant in their effects to theirs.

What remains to be done is the actual construction of the interpreter which will also incorporate intelligent backtracking features.

REFERENCES

[BOB] D. BOBROW et al: Experience with KRL 0. One cycle of a knowledge representation language- Proc IJCAI 5, 1977 pp 213-222

[BUN] A. BUNDY, L. BYRD, G. LUGER, C. MELLISH, M. PALMER: Solving mechanics using meta-level inference- Proc. IJCAI 6, 1979 pp 1017-1027

[CLA] K.L CLARK, F.G McCABE: The control facilities of IC-PROLOG- Department of computing and control; Imperial College- London 1978

[DIN] M. DINCBAS: A knowledge-based expert system for automatic analysis and synthesis in CAD- To be presented IFIP 1980

[FAH] A. FAHMI: Controle de systemes de deduction automatique fondés sur la logique- These de Docteur-Ingénieur ENSAE-CERT 1979

[GAL] H. GALLAIRE, C. LASSERRE: Controlling knowledge deduction in a declarative approach- Proc. IJCAI 6, 1979 pp s1-s12

[MIN] J. MINKER: Control structure of a pattern directed search system- TR 503, Department of computer science, U. of Maryland, 1977

[PER] L.M. PEREIRA, A. PORTO: Intelligent backtracking and sidetracking in Horn-clause programs- Departamento de informatica, Universidade nova de Lisboa, Portugal 1979

[ROU] P. ROUSSEL: PROLOG- Manuel de reference et d'utilisation- Groupe d'intelligence artificielle- Marseille Luminy 1975

THE METALOG PROBLEM-SOLVING SYSTEM
AN INFORMAL PRESENTATION

Mehmet DINCBAS

ONERA-CERT - Department of Computer Science
BP 4025 - 31055 TOULOUSE CEDEX, FRANCE

1. INTRODUCTION

This paper* presents the main features of the METALOG logic problem-solving system that we have developed using PROLOG [13]. This is a general purpose system which has also been used to construct PEACE, a knowledge-based expert system for electronic circuit design [4], [5].

In the first section we present the two levels of language that can be used to express object-knowledge and meta-knowledge. The second section presents the user-defined control expressed in the control meta-language. In the following section we discuss the strategy for forward search processing. Finally, we present failure and loop processing mechanisms in the last two sections.

2. KNOWLEDGE AND META-KNOWLEDGE EXPRESSION

The METALOG system offers two levels of language to the user :
- the object-level language (or simply object-language) in which he expresses his knowledge (domain-oriented and problem-specific),

* This research is supported by Direction des Recherches, Etudes et Techniques (under contracts DRET/78.1193 and DRET/79.1216).

- the meta-level language (or simply meta-language) in which he expresses his meta-knowledge, i.e. indications and pieces of advice on how to use this knowledge.

The knowledge/meta-knowledge duality in problem-solving is equivalent to the logic/control duality as proposed by Kowalski for the analysis of algorithms [9]. Through meta-knowledge the user can define a heuristic control over the deduction process in the object-level knowledge-base. This user-specified control has already been investigated by Gallaire [8].

In our system, the object-language in which the knowledge-base is coded is the clausal form of first-order predicate calculus, restricted to Horn clauses. The following notation is used to express object-level knowledge :

+HCLAUSE(A(x,y), B(x).C(y).nil) stands for A(x,y) ← B(x), C(y)

+HCLAUSE(A(x),nil) stands for A(x) ←

As it can be seen from this notation, Horn clauses (more exactly, the regular ones) expressing object-level knowledge are arguments of the meta-predicate (i.e. a meta-language predicate whose arguments are object-language clauses or literals) HCLAUSE, the head of the clause being the first argument and its body being the second argument.

The Horn clause :

$$\leftarrow A(x,y),B(y,z)$$

which defines a goal statement is represented as follows :

GOAL(A(x,y).B(y,z).nil)

A proof is found when we obtain GOAL(nil), which stands for a contradiction (←).

Unlike the object-language, the meta-language is PROLOG which exhibits great advantages for expressing meta-knowledge, in particular by allowing us to use clauses and predicates as terms. A meta-rule expressed in this meta-language has the following general form :

$$+ACTION - Cond_1 - \ldots - Cond_n \qquad n \geq 0$$

where ACTION refers to a built-in meta-predicate that handles object-level clauses. The literals $Cond_1,\ldots,Cond_n$ can be either meta-predicates

or simply PROLOG predicates. The system offers some built-in predicates and meta-predicates to use as $Cond_k$ but a user can define (in PROLOG) the predicates he needs.

Through this control meta-language a user can express some semantic information about his particular problem in order to reduce the size of the search space, as well as specify his problem-solving strategy. Whatever the power of a system, we believe that the "system-user association" is the most efficient because it is easier for a user to convey semantic information to the problem-solver than it is for the problem-solver to discover this same semantic information.

Thus, the system contains two kinds of knowledge (provided by the user) :

- object-knowledge coded as the arguments of the meta-predicate HCLAUSE, and
- meta-knowledge coded in a distinct control meta-language.

Let us notice that for efficiency reasons, the system allows PROLOG-predicates to be used in the object-language. In case of deterministic procedures this is a very efficient process. In our expert system PEACE, this process allowed us to use procedures defined in a formal and numerical computation module, at the object-level knowledge-base [4].

3. USER-DEFINED CONTROL

As we mentioned before, in the METALOG system a user expresses his meta-knowledge in a separated control meta-language. He has built-in meta-predicates at his disposal and he can construct meta-rules with them. These meta-rules can be about subproblem selection, procedure selection, failure-preventing and failure-processing, and other purposes.

a) Meta-rules for subproblem (or literal) selection

There are two meta-rules for controlling the selection of literals in a goal statement. The first one is :

ACTIVATE (g,p)

where : g represents the goal statement (i.e. the list of literals to be solved)

p represents the literal to be selected.

Meta-rules of this form can be used to select a particular problem p_i (by naming it explicitly) or to specify a general problem-solving strategy (by enumerating the properties of the literal to be selected). Let us give some examples.

The meta-rule

+ACTIVATE (g, ON(x,y))

indicates that the system must select the literal ON(x,y) (or any instance of it, like ON(B,A)) in the goal statement g (whenever ON(x,y) occurs in g). The meta-rules

+ACTIVATE (g,p) - TOTALINST(p).

+ACTIVATE (g,p) - PARTINST(p).

indicate to the system a general hierarchical strategy which consists in selecting a fully instantiated literal p (TOTALINST is a built-in meta-predicate). If there is no such literal in g, the system has to choose a partially instantiated one (idem for PARTINST). If no such literal exists, the system selects one according to its internal strategy.

In case of conflict, e.g. when several literals are fully instantiated, the system selects the leftmost one (in fact, it selects the leftmost literal verifying the ACTIVATE condition).

The second meta-rule used in selecting a literal is :

FREEZE(g,p).

It is the dual of ACTIVATE and it prevents the system from selecting a frozen literal p. Everything we said about ACTIVATE can be said about FREEZE. Let us give an example of its use :

+FREEZE(g,ON(x,y)) - VAR(x).

It prevents the system from choosing ON(x,y) as literal to be executed if its first argument is not instantiated (VAR is a built-in PROLOG predicate).

This meta-rule allows coroutining in a system with a static selection rule (like PROLOG).

b) Meta-rules for procedure selection

As in the case of problem selection, we have two meta-rules. These are :

CHOOSECLAUSE (p,1)

INHIBCLAUSE (p,1)

where : p represents a procedure name (i.e. the head of a clause) unifying with the selected problem, and
1 represents the body of this procedure (i.e. the list of literals to be solved).

The first meta-rule indicates a procedure to be chosen while the second meta-rule rules out procedures. These meta-rules allow us to have a content-directed procedure invocation [3], [8].

c) Meta-rules for failure-preventing and failure-processing

As we will see below, failure processing is one of the most important features of METALOG system. Although the system has a heuristically guided automatic mechanism for preventing and processing failures, intelligent backtracking and so on, a user can give semantic information about his particular problem (unsolvable subproblems, the backtracking node in the case of failure, etc ...) in order to prevent the system from useless search and thus improve the problem-solving efficiency.

The first meta-rule for this purpose is :

INSOLUBLE(p)

which allows a user to state that problem p (or any instance of p) is unsolvable.

Through a second meta-rule

INSOLUBLEBACK(p,b)

a user can also specify a node b where to backtrack to (a backtracking node). The above meta-rules prevent the system from failures. A third meta-rule

BACKFAIL(p,b)

can be used to indicate to the system a backtracking node b if the proof of p fails.

d) Other meta-rules

The system offers some other meta-rules for different purposes. For example, the meta-rule

FINISH(p)

allows a user to indicate that problem p must be solved entirely when it has been selected.

The meta-rule :

LITNUMBER(n)

can be used to indicate the maximum number of literals in a goal statement ; this will be used to prevent expanding loops (cf. loop processing section).

4. FORWARD SEARCH PROCESSING

Forward search processing specifies what the system has to perform to obtain a new goal statement G_{i+1} from another goal statement G_i, except for failure and loop processing. When a failure (or a loop) is detected and cannot be avoided, its processing is done by a failure processing (or loop processing) mechanism. This will be explained in the next sections.

Given a goal statement

$G_i = P_1.P_2 \ldots P_n.nil$

at time t_i, forward search processing consists of several steps :

- selecting a literal (or problem) P_j in G_i,
- selecting a procedure Q_j to apply to P_j,
- substituting the body of Q_j for P_j in G_i (with the necessary unifications),
- acceptance checking for the candidate goal statement G_{i+1}.

The first three steps are as for LUSH-resolution [6], [10]. The last step is added to our system in order to increase efficiency and to prevent it from failures and loops. This step consists of several tasks that will be explained below.

During forward search the problem-solver uses many kinds of meta-information coming from several sources. Indeed, it takes into account :

- its internal autonomous control,
- user-specified control,
- semantic information "learnt" during the problem-solving process (in particular, information concerning failed problems).

All this control information is managed by a meta-control module.

a) Selection of a problem in a goal statement

The problem-solver gives a higher priority to user's indications concerning the selection of problems (expressed by the meta-rule ACTIVATE).

If no problem has been specified by the user, the system selects one according to the following strategy :

- select a problem that has only one solution (i.e. which unifies with only one head of procedure), if there is such a problem ; otherwise
- select a problem that can be solved in one step (i.e. which unifies with an assertion - a procedure with an empty body -), if there is such a problem ; otherwise
- select the leftmost problem in the goal statement.

After selecting a problem, the system checks the FREEZE meta-rules to see whether the selected literal is frozen. If it is, another problem is selected.

In case of conflict, for instance when several problems have only one solution and no other control information is given, the leftmost such literal in the goal statement is selected.

b) Selection of a procedure

As in the case of problem selection, a higher priority is given to user's meta-rules (expressed by CHOOSECLAUSE). If no such meta-rule is applicable to the selected problem, the system chooses a procedure according to the following strategy :

- select the procedure that has an empty body (i.e. a unit clause) ; otherwise
- select the first procedure in the order in which they have been written.

While selecting a procedure, the system takes notice of INHIBCLAUSE meta-rules in order not to choose an inhibited clause.

c) Obtaining a candidate goal statement

This step consists in replacing in a goal statement the selected literal by the body of a given procedure and applying a matching substitution θ. This is the resolution rule. Let us notice that in our system the matching substitution is directly applied by the PROLOG interpreter.

d) Acceptance checking for a candidate goal statement

Before accepting the resolvent obtained at the preceding step as the new goal statement, a specialized module of the problem-solver performs a lot of tests in order to prevent the system from a "bad" goal statement and thus from useless search.

Let $G_{i+1} = Q_1.Q_2 \ldots Q_n.nil$

be a candidate goal statement ; the first step is the breadth-first unification. This ensures that all Q_j in G_{i+1} have compatible solutions, i.e. there is a matching substitution θ such that any literal $Q_j.\theta$ can be unified with at least one head of procedure. Obviously, if any Q_j is indicated as failure by the user or by the failure processing module, no goal statement can contain it. Thus, the breadth-first unification allows the problem-solver to reject a candidate goal statement as soon as a failure literal appears. With this acceptance checking most of the failures can be avoided.

Let us notice that at the end of the test, if it is successful, we do not keep the matching substitution and we undo this unification (in order to have the candidate goal statement as it was before this test).

The last step of acceptance checking consists of a lot of more or less heuristic tests to detect loops. This will be seen in the section concerning loop processing.

A candidate goal statement can become a goal statement only if it passes the acceptance test.

5. FAILURE PROCESSING

An intelligent failure processing includes several tasks : prevention, detection, analysis, backtracking, learning and so on. In METALOG, prevention and detection of failures are performed in the breadth-first unification step, as we have seen above. In this section we explain how the other tasks are performed within the system.

The first task is to determine the cause of failure. We can have two kinds of failure causes in our system :

- a literal fails, i.e. in all alternatives, it admits a descendent literal with no solution, or

- a goal statement fails, i.e. each literal of the goal statement has a solution but all solutions together are incompatibles (because of the shared variables).

In the latter case, the problem-solver backtracks to the preceding goal statement and tries other alternatives of this node.

In the former case, i.e. when a literal is detected as a failure, an intelligent backtracking mechanism is in charge of finding a node in the derivation tree (a goal statement) where to backtrack to. Several intelligent backtracking methods have been proposed [7], [11], [12]. The originality of our method is its great simplicity. For this method (and also for loop checking and processing), we need to memorize all the goal statements and protect them against any subsequent unification. When a failure P is detected, backtracking simply consists in testing the memorized goal statements, one by one, beginning with the most recent, in order to see whether they contain a variant or an instance of the literal P. The backtracking node is the first one which contains none of them. Here, let us notice that :

- P_1 is a variant of P if P_1 and P are identical except for the names of variables (e.g. ON(x,y) and ON(z,t)) ;
- P_1 is an instance of P if there exists a substitution θ such that $P.\theta = P_1$ (e.g. ON(A,y) and ON(z,z) are instances of ON(x,y)).

The last task about failures is their memorization. This corresponds to a simple learning process because they will be used during the whole derivation process and thus contribute to the pruning of the search space. In METALOG, this memorization is carried out "intelligently", i.e. before memorizing a failure problem P, the system looks for a memorized failure P_1 which is an instance of P. If there are such problems P_i, all of them are deleted and the most general failure P is added. This treatment avoids having redundant failure indications.

6. LOOP PROCESSING

The METALOG system has some heuristic mechanisms for detecting and processing loops. During the problem-solving process two kinds of loops occur :

- logic loop
- expanding loop.

a) A logic loop occurs when we obtain a candidate goal statement G_i which is identical with a previous goal statement G_j (with $j < i$). The logic loop is very easy to detect (this is done while forward searching) and to process. Logic loops often occur in systems with a static strategy for selecting literals and procedures, like PROLOG.

b) An expanding loop may occur in many different ways. The two most frequent ones are :

- resolvent-expanding loop : recognized as a regular increase of the number of literals in the resolvent (i.e. goal statement)
- term-expanding loop : recognized as a regular increase of the size of terms in the arguments of a literal in the resolvent.

These expanding loops can be detected by the system, possibly assisted by the user who can indicate by meta-rules (like LITNUMBER as already seen) some heuristic detection rules.

When an expanding loop is detected, the system has some heuristic methods to process it. This processing is based on checking the "looping" goal statement to see whether there is a literal with no solution ; to do this, the system proceeds as follows :

- check fully instantiated literals ;
- check partially instantiated literals ; and
- check the other literals.

If a failure literal is detected in the "looping" goal statement, the intelligent backtracking mechanism is used to backtrack to the appropriate node.

7. CONCLUSION

In this paper we informally presented the METALOG system which is written entirely in PROLOG. We believe that the system can be successfully used to construct expert systems (as we did for PEACE) as well as for other applications like plan-formation problems and intelligent data-base systems.

ACKNOWLEDGMENTS

I would like to thank H. GALLAIRE for all discussions I had with him and his advice during the development of the system.

REFERENCES

[1] A. BUNDY et al., Solving mechanics problems using meta-level inference, Proc. IJCAI 79, Tokyo 1979, p. 1017-1027.

[2] R. DAVIS and B.G. BUCHANAN, Meta-level knowledge : overview and applications, Proc. IJCAI-77, August 1977, p. 920-927.

[3] R. DAVIS, Generalized procedure calling and content-directed invocation, Proc. Symposium on AI and Programming Languages, August 1977, p. 45-54.

[4] M. DINCBAS, Etude d'un système expert pour la CAO : présentation de PEACE, Rapport final du contrat DRET. Document CERT/DERI n° 3/3122, Nov. 1979.

[5] M. DINCBAS, A knowledge-based expert system for automatic analysis and synthesis in CAD, IFIP Congress 80, Tokyo 1980.

[6] M.H. Van EMDEN, Programming with resolution logic, Machine Intelligence 8, p. 266-299, 1977.

[7] A. FAHMI, Contrôle de systèmes de déduction automatique fondés sur la logique. Thèse de Docteur-Ingénieur ENSAE, Nov. 1979.

[8] H. GALLAIRE and C. LASSERRE, Controlling knowledge deduction in a declarative approach, Proc. IJCAI-79, Tokyo, 1979.

[9] R. KOWALSKI, Algorithm = logic + control, Comm. ACM, Vol 22, n° 7, 1979, p. 424-436.

[10] R. KOWALSKI, Logic for problem solving, North Holland, 1979.

[11] J.C. LATOMBE, Failure processing in a system for designing complex assemblies, Proc. IJCAI-79, Tokyo, 1979.

[12] L.M. PEREIRA, Backtracking intelligently in AND/OR trees, Dept. Informatica, Universidade Nova de Lisboa, Lisbon Portugal, June 1979.

[13] P. ROUSSEL, PROLOG : Manuel de référence et d'utilisation, Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975.

EVALUATING FUNCTIONS DEFINED IN FIRST ORDER LOGIC

Luigia Aiello

Istituto di Elaborazione dell'Informazione, Consiglio Nazionale delle Ricerche
Via S. Maria 56, I-56100 Pisa, Italy.

Computer Science Department, Stanford University
Stanford, California 94305, USA.

ABSTRACT

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.

1. INTRODUCTION

This paper reports on an extension that has been recently made to the FOL system, namely a compiling algorithm from FOL into LISP, which allows for a direct evaluation in LISP of functions and predicates defined in First Order Logic. The first motivation that has prompted us to devise this extension was the hope of substantially increasing the efficiency of the FOL evaluator; the examples shown is Section 4 confirm this expectation.

There is also another reason for this extension: the algorithm that translates systems of mutually recursive function and predicate definitions from FOL into LISP programs allows for a more direct use of First Order Logic as a programming language. The user of FOL is in fact relieved of the burden of directly coding pieces of programs in LISP because the system itself builds code (i.e. semantic attachments) starting from a specification given in the syntax of first order logic.

This automatic transformation of FOL axioms (or, in general, facts) into semantic attachments has also the advantage of guaranteeing the consistency between the syntactic and semantic specifications of an FOL domain of discourse (i.e L/S structure), or at least, to limit the user's freedom of introducing inconsistencies.

The paper is not self-contained; we refer to the literature [Wey77,78,79] for an extensive description of FOL, and limit the comparisons with related work to a minimum.

In the following, we provide the reader with enough insights in order to understand the evaluation mechanism of FOL, the compiling algorithm we have designed and give some hints on further work to be done. In fact, FOL is an experimental system and, as it is often the case with such systems, it grows through the experience of its users. In the paper, mostly in the concluding section, we suggest some improvements of our compiling

algorithm and of the FOL evaluation mechanism to further increase the efficiency of the evaluations and to further guarantee the user that the syntactic specification of an L/S structure agrees with the semantic specification.

The paper is organized as follows. Section 2 contains a short description of FOL which essentially emphasizes the notions of L/S structure, semantic attachment and gives hints on the behaviour of the FOL evaluator. Section 3 describes the compiling algorithm which translates systems of mutually recursive function and predicate definitions into LISP programs. Section 4 shows, by means of examples run in FOL (both with and without the compiling algorithm), how important this extension has been, not only in terms of reducing the cpu time of evaluations but also in making some evaluations possible that otherwise were not. Section 5 presents a discussion of semantic attachment. Finally, Section 6 presents some ideas on further improvements and few concluding remarks.

2. SHORT DESCRIPTION OF FOL

FOL is a *conversational system* capable of *reasoning* with a user in the language of First Order Logic. It has been designed by Richard Weyhrauch at the Stanford Artificial Intelligence Laboratory and is implemented in LISP on a KL10. The aims of such a system are many [Wey79]. FOL is primarily intended to be a tool for developing a mechanized formal theory of reasoning, with applications in knowledge representation in both Artificial Intelligence and Mathematical Theory of Computation.

We cannot give here a detailed description of FOL and of its many features. We only sketch some of them. The reader is referred to [Wey77,78,79] for more insights and to [AW80,Fil79,Tal80] for some examples of applications.

FOL can be viewed as an interactive theorem proving system. The interaction takes place in a user-determined first order language with equality, enriched with a partially ordered sort structure. The presence of sort information is very useful: interesting conclusions can often be drawn on its basis alone. FOL deals with the *full predicate calculus*, i.e. no restriction is imposed on the well formed formulas – wffs – being manipulated. In addition, FOL allows conditional terms and wffs, which, while being a conservative extension of ordinary predicate calculus, are very useful in defining functions and predicates.

FOL implements the first order logic with the deductive apparatus of Prawitz's *Natural Deduction* [Pra65] enriched in many ways: besides the already mentioned sort structure and the relative sort checking facilites, many decision procedures have been added to FOL.

There are at least three main features of FOL which are not commonly available in most theorem provers and that contribute to make it flexible and epistemologically adequate for knowledge representation and mechanized formal reasoning. 1) The possibility of distinguishing between syntactic and semantic knowledge and using both kinds of knowledge in the same derivation. 2) The possibility of dealing at the same time with many theories (actually it is more appropriate to call them L/S structures). 3) The possibility of representing meta-theoretic knowledge and using it in performing deductions.

The first point will be expanded in Section 2.1. We do not enter into details about the last two points (even though they are very important) because they are out of the scope of the present paper. We only say that in FOL many L/S structures may coexist and - to some extent - exchange information between one another. One privileged L/S structure, named META, is used to represent meta-theoretic knowledge, i.e. to speak about any L/S structure. The presence of a command named REFLECT allows for the transfer of knowledge from the meta-theory level into the theory level during a deduction.

2.1. L/S structures - Semantic Attachments

As pointed out in [Wey77,78,79], one of the most innovative aspects of FOL as a reasoning system is the possibility of representing and using both syntactic and semantic information for the same context of discourse. This idea is embodied in the notion of *L/S structure*.

An L/S structure is a triple $<L, S, F>$, where L is a language, S is a simulation structure and F is a set of facts.

More precisely, L is a *sorted language*, i.e. a finite set of variable symbols and a finite set of constant symbols (all of them with an associated sort), a finite set of function symbols (with associated arity, domain and codomain), a finite set of predicate symbols (with associated arity and domain).

A *simulation structure* S is a computable partial specification of a *model*. It consists of an association between symbols (not necessarily all) of L and *objects* of some *real world*. In the case of FOL the world is LISP and the objects are LISP data structures. A simulation structure is specified by providing a denotaion map for *some* of the constant symbols of L and *some* information about the meaning of *some* of the function and predicate symbols of L. Note that FOL does not require that S contains an *algorithm* for computing the functions and predicates associated with the symbols of L. The information about a function (predicate) can be limited to the specification of how it maps only a few elements of the domain. Note that all predicates and functions of the simulation structure are total, even if their behaviour is specified only on some elements of the domain. The FOL evaluator can deal with this kind of partialness: if it tries to compute the value of a function on some arguments for which no information is provided, it returns "I don't know" as an answer.

A simulation structure is built by associating symbols of the language with LISP entities via the so called *semantic attachment*.

The set of facts F of an L/S structure is a finite set of well formed formulas of L together with a justification, i.e., the specification of whether they are axioms, assumptions or they are the result of some deduction (in this case the system keeps track of the dependencies and of the inference rule used).

Note that L/S structures are very different in spirit from the logicians' notion of theory-model pairs. We have already pointed out the differences between the notion of a model and that of a simulation structure; in addition, it is worth stressing that the set of facts in a L/S structure is *finite* and not deductively closed.

In FOL, each application of a deduction rule (it can be one of Prawitz's inference rules or a sophisticated decision procedure) is a mapping from an L/S structure into an L/S structure, the second one being obtained from the first one by adding a new fact, namely the conclusion of the deduction.

Despite these differences, with an abuse of language, sometimes L/S structures are called theories, and simulation structures are called models (more precisely, a value in a simulation structure is called a model value).

2.2. A simple example

The idea exploited in many proofs carried out in FOL is that the syntactic knowledge (language, set of axioms, set of theorems) about a theory is used in deriving proofs along with the semantic knowledge about the intended model for that theory. Use of semantic information and use of syntactic information are intermixed. When a deduction is performed, semantic attachments are used to compute values in the intended model, such values are then used at the syntax level to carry on the deduction symbolically.

To better illustrate this point we present an L/S structure for natural numbers. In this example we define the language of natural numbers, assert some axioms and build a simulation structure for them. Note that, in the FOL system, numerals are automatically declared as individual constants and are attached to the expected numbers. Thus the following axiomatization includes by default the numerals NATNUM and their attachments to the LISP natural numbers.

```
DECLARE INDVAR n m p q ∈ NATNUM;
DECLARE OPCONST suc(NATNUM)=NATNUM;
DECLARE OPCONST + * (NATNUM,NATNUM)=NATNUM;

AXIOM  Q:
  ONEONE:∀ n m.(suc(n)=suc(m)⊃n=m);
  SUCC1: ∀ n.¬ (0=suc(n));
  SUCC2: ∀ n.(¬0=n⊃∃ m.(n=suc(m)));
  PLUS:  ∀ n.n+0=n
         ∀ n m.n+suc(m)=suc(n+m);
  TIMES: ∀ n.n*0=0
         ∀ n m.n*suc(m)=(n*m)+m ;;

ATTACH suc ⇔ (LAMBDA (X) (ADD1 X));
ATTACH +   ⇔ (LAMBDA (X Y) (PLUS X Y));
ATTACH *   ⇔ (LAMBDA (X Y) (TIMES X Y));
```

The first group of declarations creates the language. The second group specifies some facts: Robinson's axioms Q without the equality axioms [Rob50]. The third group makes the semantic attachments of function symbols to the LISP code for computing them. Actually, FOL requires some more information about details that are irrelevant here, for example the parser must be informed that the symbols + and * are infix operators and have a certain precedence.

Suppose that we extend the previous L/S structure by adding the following facts:

```
1 = suc(0)
2 = suc(1)
3 = suc(2)
```

If in this context we want to prove that for all p the following equality holds:

```
p*0+(1+2) = 3
```

we can use syntactic information only (i.e. repeatedly rewrite p*0+(1+2) by means of the axioms, by rewriting + in terms of suc, using also some of the other facts of the L/S structure). This process can however be improved by noticing that, after p*0 has been rewritten as 0 (by means of TIMES1, i.e. the first axiom for times), the term 0+(1+2) can be directly computed via semantic knowledge yielding 3. Note that this entire process is performed by FOL only giving it a single command, namely

```
EVAL p*0+(1+2)=3 BY {TIMES1}
```

where the names in braces are the simplification (or rewriting) rules, i.e. the set of facts we want to be used in rewriting the formula. This set is sometimes called *simpset*.

While the above equality *could* be evaluated via syntactic knowledge alone, of course it could not be proved via semantic computation alone, due to the fact that the term p*0 appearing on the left hand side of the equality is not a ground term. Hence, syntactic (*symbolic*) evaluation is an essential part of the FOL evaluator.

From the observation that semantic evaluations can only be performed on ground terms, one may conclude that there are not so many interesting cases in which it actually can be done. This impression is incorrect, in particular in much of the reasoning done at the meta-theory level almost all the manipulations are done on ground terms (terms that are built out of constants, i.e. names in META for objects at the theory level).

Before ending this section it is worth giving some more hints on the workings of the FOL evaluator. We will not explain the behaviour of the evaluator in every possible situation, since it would be too long and, moreover, the flow of the control in all of them is very much alike. In order to simplify the presentation we illustrate the behaviour of the FOL EVAL by giving more details about the evaluation presented above, namely:

```
EVAL term1=term2 BY simpset;
```

in the case where $term_1$ is p*0+(1+2), $term_2$ is 3 and simpset is {TIMES1}.

The evaluator looks for all the semantic attachments that are provided in the current L/S structure and uses them, on both terms in a leftmost outermost manner. In the case of p*0+(1+2), namely +(*(p,0),+(1,2)) the evaluator realizes that the function symbol + has an attachment, then it tries to evaluate the arguments *(p,0) and +(1,2). In the first case, it is provided with an attachment for the function symbol *, but it has no attachment for the symbol p (nor p can be syntactically simplified). Hence the semantic evaluation on *(p,0) fails. At this point the evaluator checks whether in the simpset there is some fact that matches with (some subterm of) the term at hand. In our case, the axiom TIMES1 matches the term *(p,0), which is then rewritten (via symbolic evaluation) as

0. As for the term +(1,2), since both the symbol + and the numerals 1 and 2 have an attachment, control is passed to the LISP EVALuator which uses these attachments to compute a model value, namely the number three, whose name in the current L/S structure is the numeral 3. The FOL evaluator is now left with the term +(0,3), which by using semantic information, is evaluated to the term 3 (i.e. the numeral 3, which is the name for the LISP number three, that is the model value computed by applying the LISP evaluator on the semantic attachment of the symbol + and the model values attached to the symbols 0 and 3). At this point the evaluation of the term at the left hand side of the equality of the original wff ends. The evaluation of the term at the right hand side of the equality is straightforward: the result is the term 3. Since the evaluation of both the term at the left hand side and the one at the right hand side of the original wff yield the same term, the wff has been proved, hence the current L/S structure is extended by adding the fact that p*0+(1+2)=3.

In this short trace of the behaviour of EVAL we have omitted many details; we hope however to have provided the reader with an idea of the flow of the control during the evaluation and, more important, of the order in which the use of syntactic and semantic knowledge is intermixed.

3. COMPILING SYSTEMS OF MUTUALLY RECURSIVE FUNCTIONS

In order to further exploit the ability of the FOL evaluator to directly invoke the LISP interpreter whenever it is provided with code for some of the function or predicate symbols occurring in the term/wff being evaluated (and a model value for all the arguments of such an occurrence of the function or predicate symbol), a *compiler* has been designed and implemented which translates systems of mutually recursive definitions into LISP code.

The compilation of systems of function (predicate) definitions from FOL into LISP allows FOL to transform syntactic information into semantic information. In other words the compiling algorithm allows FOL to automatically build parts of a model for a theory, starting from a syntactic description.

The FOL-LISP compiler is invoked by giving FOL the command

```
COMPILE name-list;
```

each `name` in `name-list` is checked to see whether or not it is the name of a *fact* (i.e. an axiom or a theorem) in the current L/S structure and if this fact is a function or predicate definition (note that we use the word "definition" in a broader sense than logicians do). In this case the LISP code for the *definiens* is computed and attached to the *definiendum* as a semantic attachment, i.e. COMPILE has the same effect as the previously seen command ATTACH.

Using a sligthly modified version of the notation of [CMC79] the compiler treats systems of mutually recursive definitions of the following form.

$$\forall x_{i_1} \ldots x_{i_r} . f_i(x_{i_1}, \ldots, x_{i_r}) = \sigma_i[\Phi_i, \Psi_i, \bar{c}_i, x_{i_1}, \ldots, x_{i_r}]$$

$$\forall y_{j_1} \ldots y_{j_s} . (P_j(y_{j_1}, \ldots, y_{j_s}) \equiv \tau_j[\Phi_j, \Psi_j, \bar{c}_j, y_{j_1}, \ldots, y_{j_s}])$$

where f_i-s are function symbols and P_j are predicate symbols. The σ_i are terms in Φ_i-s, Ψ_i-s, $\vec{c}_i$ and x_i-s; the τ_j-s are wffs in the Φ_j-s, Ψ_j-s, $\vec{c}_j$-s and y_j-s. By $\vec{c}$ we denote a tuple of constant symbols. By Φ (resp. Ψ) we denote a tuple of function (resp. predicate) symbols. Φ (resp. Ψ) may contain some of the f_i (resp. P_j), but it is not necessarily limited to them, i.e. other function and predicate symbols besides the definiendums can appear in each definiens.

The compilation algorithm first performs a well formedness check, then a compilability check.

Well formedness check – Each `name` in `name-list` is checked to see if it actually corresponds to a fact in the current L/S structure, and in this case if this fact is a definition, i.e. if it has one of the two following forms:

$$\forall x_{i_1} \ldots x_{i_r} . f_i(x_{i_1}, \ldots, x_{i_r}) = \ldots$$
$$\forall y_{j_1} \ldots y_{j_s} . (P_j(y_{j_1}, \ldots, y_{j_s}) \equiv \ldots$$

Note that no further check is performed on the well formedness or the well sortedness of the definitions, since all these checks have already been performed by the FOL parser when these facts have been input in the current L/S structure.

Compilability check – It consists in verifying that : a) each definition is a closed wff, i.e. no free variable occurs in it; b) all the individual constants and the function (predicate) symbols appearing in the definiens have a model value, i.e. individual constants are attached to some model value, while function and predicate symbols have some code attached to them (either by the compilation presently being done – which allows for recursion or mutual recursion – or by a previous attachment or compilation); c) the definiens can contain logical constants, conditionals and logical connectives but no quantifiers.

Before explaining why these restrictions have been imposed to a system of definitions in order to be compilable, we give more explanations on the workings of the compiling algorithm.

3.1. The compiling algorithm

Let the definiens be a term `TRM` or a wff `WFF`. `TRM` (resp. `WFF`) is traversed only once to check its compilability and compile it. In case of failure, i.e. as soon as a subpart of `TRM` (resp. `WFF`) is not compilable, the process is stopped.

The compilability check and code generation is done in a recursive way, as follows:
– if `WFF` is ¬`WFF1`, it is compiled as `(NOT c-WFF1)` where `c-WFF1` is the code generated by the compilation of `WFF1`;
– if `WFF` is `TRM1=TRM2`, it is compiled as `(EQUAL c-WFF1 c-WFF2)`;
– if `WFF`is `WFF1`∧`TRM2` or `WFF1`∨`WFF2`, it is compiled as `(AND c-WFF1 c-WFF2)` or `(OR c-WFF1 c-WFF2)`, respectively;
– if `WFF` is `WFF1`≡`WFF2` or `WFF1`⊃`WFF2`, it is compiled in terms of ∧, ∨, ¬;
– if `WFF` (resp. `TRM`) is a conditional wff (resp. term) its `test`, `then` and `else` parts are compiled and the resulting `(COND (c-test c-then)(T c-else))` returned, where `c-test` is the code for `test`, etc.;

– if TRM is a variable, then it must not be a free var, the code is the variable itself;
– if TRM is the symbol TRUE then T is the code for it;
– if TRM is the symbol FALSE then NIL is the code for it;
– if TRM is a constant symbol, its model value (i.e. a LISP constant) is computed and returned, if any, otherwise a failure occurs;
– if TRM is an application of a function (predicate) symbol to some terms, then the function (predicate) symbol and the terms are compiled and their LISP application is returned. The function (predicate) symbol is first searched in the list of symbols defined in the system being compiled in this "block" compilation, (i.e. we look first for recursive or mutually recursive definitions). In this case the name of the LISP function attached to that symbol by the present compilation is returned as its model value. Otherwise, an attachment for that symbol is searched. If it if found, this code is returned, otherwise a failure occurs.

A few more remarks are in order. FOL allows for some polymorphism: if a function (predicate) symbol has different attachments for different sorts, then the compiling algorithm selects the approriate code for it, according to the sort of the terms that are arguments for that function (predicate) symbol in the term being compiled.

Finally, note that we allow for the separate compilation of function (predicate) definitions within a given L/S structure, the only requirement being that they are compiled in an appropriate order and that systems of actually mutually recursive functions (predicates) are compiled together.

3.2. An example of compilation

To provide an example of compilation, we present here the mutually recursive definitions of two predicates, without providing details on the language and sorts appearing there. They have been taken from the meta-theoretic description of arithmetic expressions (remember that META is just an ordinary L/S structure). They are used to check whether or not an individual variable or function symbol occurs in a list (of individual variables and function symbols) and given two such symbols which one occurs first:

```
AXIOM MEMBER:
∀ vf lex.(MEMBER(vf,lex)≡
          if EMPTYLEX(lex) then FALSE
             else if vf=hd(lex) then TRUE
                     else MEMBER(vf,tl(lex)));;
AXIOM BEFORE:
∀ vf1 vf2 lex.(BEFORE(vf1,vf2,lex)≡
          if EMPTYLEX(lex) then FALSE
             else if vf1=hd(lex) then MEMBER(vf2,tl(lex))
                     else BEFORE(vf1,vf2,tl(lex)));;
```

If we type to FOL the command for compiling them, i.e.

```
COMPILE MEMBER, BEFORE;
```

it answers:

```
MEMBER compiled as
(DE C-MEMBER (vf lex)
   (COND ((NULL lex) NIL)
         (T (COND ((EQUAL vf (C-hd lex)) T)
                  (T (C-MEMBER vf (C-tl lex)))))))
BEFORE compiled as
(DE C-BEFORE (vf1 vf2 lex)
  (COND ((NULL lex) NIL)
        (T (COND ((EQUAL vf1 (C-hd lex))
                  (C-MEMBER vf2 (C-tl lex)))
                 (T (C-BEFORE vf1 vf2 (C-tl lex)))))))
```

The compiling algorithm uniformly generates names for the LISP functions it creates (C-MEMBER is the code for MEMBER, etc.) and, as stated above, it uses code generated in previous attachments and compilations. In the above examples EMPTYLEX had been previously attached to the LISP predicate NULL, hd (head) and tl (tail) to C-hd and C-tl, respectively.

3.3. Restrictions on the compilability

The above description and examples show that the code produced by the compiling algorithm is pretty straightforward: it is a transliteration from FOL syntax into LISP syntax, hence we will not go into further details about it, nor we will bother about its correctness. We turn instead to explaining the restrictions imposed on systems of mutually recursive function (predicate) definitions in order to be compilable. They have been listed above, as points a), b) and c) of the compilability check.

a) *Closedness of the definition* – We want each definition to be a closed wff. In first order logic with natural deduction we cannot infer $\forall x.P(x)$ from $P(x)$. Hence we cannot compile free variables possibly appearing in the definition as LISP bound variables, which would correspond to compile the universal closure of the given wff. If, conversely, we compile free variables occurring in the definition as free variables in the LISP code, they would result in unbound variables for the LISP EVAL (or, even worse, they might be dynamically captured by some bound variable with the same name).

b) *Existence of attachments* – A consideration similar to the above one justifies the choice we have made to allow the compilation only when all the constant, function and predicate symbols occurring in the definiens have a model value. If this is not the case for some of them, the LISP EVAL, when trying to evaluate that constant or to apply that function (or predicate), would result in an error. The FOL evaluator, as it presently is, after invoking the LISP evaluator expects it to return a model value, it does not know how to recover from an error occurring at the LISP level.

c) *Quantifications in the definiens* – The choice of not allowing quantifiers in the definiens has been made because it was not needed for the present applications of the compiling algorithm. They can be allowed, even though only in bounded quantifications. Clearly we cannot allow for an unbounded quantification in the definiens, (the computa-

tion at LISP level has to return a value), but bounded quantification can be introduced, by compiling, for instance, $\forall x . P(x)$ with the code that repeatedly binds x to all the data objects of the (finite) domain associated with its sort, checking whether or not P holds for all of them (analogously for the bounded existential quantification).

3.4. Soundness of the compilation

While the correctness of the compiling algorithm should not constitute a problem, a legitimate question is the following: Is the compilation process sound? To say it in other words, the question to be asked is: Who guarantees that running the FOL evaluator syntactically on a system of definitions gives the same result as running the LISP evaluator on their (compiled) semantic attachments?

The answer is that the two evaluations are weakly equivalent (i.e. if both terminate, they produce the same result). This is because the FOL evaluator uses a leftmost outermost strategy of function invocation (which corresponds to call-by-name) while the mechanism used by the LISP evaluator is call-by-value. Hence, compiling a function can introduce some nonterminating computations that would not happen if the same function were evaluated symbolically.

This however does not constitute a serious problem and it will be overcome in the next version of FOL. In fact, it will be implemented in a dialect of LISP which is pure applicative, statically scoped and whose evaluator implements call-by-need (note that, in this case, call-by-need is strongly equivalent to call-by-name).

4. AN EXAMPLE OF USE OF COMPILED CODE

The first sizable application of the compiling algorithm from FOL recursive function definitions into LISP code has been done in order to develop a meta-theoretic description of arithmetic expressions.

This application is presented in [AW80]. The FOL theory (and meta-theory) of arithmetic expressions is a completely general one. Arithmetic expressions are allowed to contain variables ranging over INTEGERs, the usual operators (+,*, prefix and infix -) and uninterpreted function symbols. No restriction is imposed on them, nor on their arguments. To be more precise: function symbols may take objects of any sort as their arguments; the only restriction is that they must be hereditarily well sorted and return integers as values.

Question answering involving arithmetic expressions often requires two expressions to be checked for the identity of their values for all the interpretations of the variable symbols occurring in them. This cannot be done by a single rewriting, using the associative and distributive laws (etc.) holding for elements of an integral domain, because of the presence of the commutative laws for plus and times. In fact, the commutative laws may cause a rewriting system to loop.

To solve this problem, an algorithm has been devised in META which describes manipulations on arithmetic expressions that transform them into a canonical form with respect

to commutativity (in addition to associativity, distributivity, evaluation of ground sub-expressions, elimination of zeroes and ones, etc.). This has the property that arithmetic expressions are equal if their canonical forms are the same. Such algorithm relies on an intermixed symbolic evaluation of the expression at the theory level and of a reordering done at the meta-theory level. The specification of the algorithm in META is a rather long piece of FOL code and running it in a completely symbolic way and running it by compiling LISP code for all the definitions (about 30) substantially changes its behaviour.

Consider the following examples (where f has been declared as OPCONST mapping pairs of INTEGERs into an INTEGER):

```
(a)     x*(-y)-(-y)*x
(b)     x*3*(4-y)-(-y+4)*3*x
(c)     x*0*f(4-y,y)
(d)     x*y*f(y,z)-y*x*f(y,z)
(e)     x*3*(4-y)+(z-u)*f((x+y)*w,w*u)
          -12*x-(u-z)*f(w*x+w*y+0,u*1*w)+3*x*y
```

To make the examples more straightforward we have chosen only expressions that are to be proved to be identically zero. The times reported in the following are KL10 cpu time. For each expression, in the first column we report the cpu time for the simplification done using the compiler, in the second one the cpu time for the simplification done completely symbolically.

(a)	5''	1'20''	
(b)	3''	stack overflow	
(c)	2''	11''	
(d)	12''	5' 5''	(*)
(e)	3'11''	stack overflow	

(*) Note that during the last garbage collection before the result was produced only 310 LISP cells were recovered!.

We think that these figures are self-explaining. The reader should not be mislead by them and conclude that the FOL evaluator is particularly inefficient. The point we are making here is only that computations on model values are more efficient than symbolic manipulations at the syntax level, hence that the use of semantic attachments in FOL has to be encouraged.

5. IN DEFENSE OF SEMANTIC ATTACHMENT

The semantic attachment in FOL has been criticised as being error prone. The main objection is that the user of FOL has the freedom of introducing two separate descriptions (a syntactic one and a semantic one) within a single L/S structure, using two separate languages and the system does not provide means to check whether these two separate descriptions actually match. For instance, going back to the example of Section 2.2, in FOL as it presently is, there are no means of checking that the LISP function attached

to the function symbol + satisfies the properties stated by the axioms PLUS.

Conversely, we consider the introduction and use of semantic attachments in FOL a very sound idea and not only because it increases the efficiency of evaluations but (more important) because it is epistemologically valuable to have a clear distinction between the syntactic part and the semantic part of a description of a context of discourse.

In addition, it has not to be underestimated that the notion of simulation structure in FOL allows for partial descriptions. Their need is certainly not evident from the examples provided in this paper, where each function (predicate) symbol introduced had a clear, well known, model value (i.e. a nice recursive function). But if you think of FOL as a system that allows you to represent knowledge about a domain of discourse as soon as you gather it by means of some observation, then the possibility of feeding it with partial information about model values seems the only viable way of setting up and updating a domain of discourse between FOL and the user.

Let us go back to the above objection, that the syntactic and the semantic specification of an L/S structure are made in two different, and in a sense incomparable languages. Note that the presence of the compiling algorithm described in this paper has reduced this problem to a minimun. In fact now the user, even though he intends to use some information at the semantic level, can input it in the form of FOL axioms (or prove it from previously specified axioms), and then transform it into semantic knowledge by means of a compilation.

There are many advantages in setting up an L/S structure by giving FOL syntactic information and then allowing the compiler to transform it into a simulation structure.

First, the FOL user can specify his knowledge in the syntax of first order predicate logic and be guaranteed that the LISP code of the simulation structure generated from it actually models the syntax.

Second, the user can ask FOL itself to prove any property he wants of the notions he is introducing, thus increasing his confidence on the consistency of the specification he is setting up.

Note that in this last point we have never mentioned the word correctness. In this case, in fact, we think that there is no point in speaking about correctness: the only thing that matters is to provide the user with the ability of verifying that the domain of discourse he is setting up with FOL corresponds to his intentions.

Before ending this section we also notice that, once the theory of LISP in FOL has been completely developed [Tal80], an algorithm can be devised for transforming LISP recursive function definitions into FOL formulas, using for instance the minimization schema proposed by J. McCarthy. This, besides allowing for a theory of recursive functions in first order logic, allows for a further check to be done on L/S structures. Namely, it can be checked that the semantic attachments provided in given L/S structure (in particular, those that are not the result of a compilation) are consistent with the syntactic part of the L/S structure itself.

6. FURTHER WORK - CONCLUDING REMARKS

We have given some indication concerning the workings of the FOL evaluator and its use of syntactic and semantic knowledge in performing evaluations. We have then illustrated a compiling algorithm for systems of recursive function (predicate) definitions from FOL into LISP, which greatly improves the performance of the evaluator itself. Here we describe some possible further improvements.

As pointed out in Section 2.2, the strategy that is presently adopted by the FOL evaluator is that, whenever it has some semantic information about the term (or wff) being evaluated, it uses that information first; if nothing can be done at the semantic level, it tries with a syntactic simplification. This strategy was the only reasonable one when the semantic attachments were very few and, conversely, the sets of simplification rules tended to be very large. The addition to FOL of the compiling algorithm described in this paper has changed this balance: facts at the semantic level are many more than those at the syntax level. This implies that a rethinking of the FOL evaluator has to be done in order to choose which information has to be used first. It is our opinion that syntactic information has to be used first. In fact, in a L/S structure where all the function (predicate) definitions have been transformed into semantic attachments, the only knowledge that is left at the syntactic level is in the form of *properties* of these functions and predicates. The use of such properties before trying to compute the function (predicate) may result in avoiding some computation.

To clarify this point we give an example. Suppose that, in the theory of Section 4 you declare a binary function `f` which maps pairs of `NATNUM`s into `NATNUM`, and add (via compilation, or "by hand") an attachment to it. Suppose that in this L/S structure you prove the theorem:

$$\forall\ x.f(x,x)=0$$

From now on, when evaluating `f(trm,trm)` (where `trm` stands for any term), you certainly want to use the above theorem and immediately rewrite `f(trm,trm)` as `0`, instead of running the LISP evaluator on the code for `f` applied to the model value of `trm`, and re-discover that the result is `0`.

One further improvement worth considering is to allow the FOL evaluator to handle some kind of LISP errors, in order to enhance its possibility of performing mixed computations. In Section 3.3 we have said that one of the requirements for a definition to be compilable is that all the function (predicate) symbols occurring in the definiens must have an attachment, to avoid that the LISP EVAL encounters an undefined function. The FOL evaluator can be designed to handle this kind of errors, pass back the control to its symbolic evaluation part and check the simplification set for some fact regarding this function (predicate) symbol, and, if possible, perform a rewriting.

Finally, a subject for further investigation consists in improving the compiler in order to increase the efficiency of the code it generates. In fact, as it presently is, the transformation of first order function (predicate) definitions into code is a transliteration, no optimization at all is performed. This is a subject worth investigating: partial evaluation and other

efficiency increasing transformations (for example, in the style suggested by [BD77] or by [AAP78]), can be performed on the code produced by the FOL-LISP compiler.

Due to the lack of space we cannot present a detailed comparison between our work and other proposals circulated in the recent literature, mostly in automatic synthesis of programs. We only mention that the use of (compiled) meta-functions in FOL is very much in the same spirit as [BM79].

ACKNOWLEDGEMENTS

C. Talcott is acknowledged for useful discussions, J.S. Moore for a careful reading of the manuscript that improved it. R.W. Weyhrauch deserves special thanks: all the conversations we have had about FOL have been interesting, stimulating and enlightening.

REFERENCES

[AAP78] Aiello, L., Attardi, G. and Prini, G. Towards a More Declarative Programming Style, *Formal Description of Programming Concepts*, North Holland (1978).

[AW80] Aiello, L. and Weyhrauch R.W. Using Meta-theoretic Reasoning to do Algebra, *Proc. of the 5-th Automated Deduction Conf.*, Les Arcs (1980).

[BM79] Boyer, R.S. and Moore, J.S. Metafunctions: Proving them correct and using them efficiently as new proof procedures, Comp. Sci. Lab., SRI Int. (1979).

[BD77] Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs, *JACM*, 24,1, 44-67 (1977).

[CMC79] Cartwright, R. and McCarthy, J. First Order Programming Logic, *Proc. of the 6-th ACM Symp. on Principles of Programming Languages*, San Antonio (1979).

[Fil79] Filman, R.E. Observation and Inference applied in a Formal Representation System, *Proc. of the 4-th Workskop on Automated Deduction*, Austin, Texas (1979)

[Pra65] Prawitz, D. Natural Deduction – a Proof-Theoretical Study, Almqvist & Wiksell, Stockholm (1965).

[Rob50] Robinson, R. M. An Essentially Undecidable Axiom System, *Proc. Int. Cong. Math.*, Cambridge, Mass. (1950).

[Tal80] Talcott, C. FOLISP: a System for Reasoning about LISP Programs, Stanford A.I. Lab. Memo, in preparation (1980).

[Wey77] Weyhrauch, R.W. FOL: A Proof Checker for First-order Logic, Stanford A.I. Lab. Memo AIM-235.1 (1977).

[Wey78] Weyhrauch, R.W. The Uses of Logic in Artificial Intelligence, Lecture Notes of the Summer School on the Foundations of Artificial Intelligence and Computer Science (FAICS '78), Pisa (1978).

[Wey79] Weyhrauch, R.W. Prolegomena to a Mechanized Theory of Formal Reasoning, Stanford A.I. Labo. Memo AIM-315 (1979); to appear in *Artificial Intelligence Journal* (1980)

Runnable Specification As a Design Tool

Ruth E. Davis
Electrical Engineering and Computer Science Department
University of Santa Clara
Santa Clara, CA 95053

There are at least four phases in the development of "correct" software.

1) Understanding the problem. The program designer may work with intended users of the system to develop an intuitive understanding of the problem and possible approaches to its solution.

2) Formal specification. Once the designer knows intuitively how to solve the problem, the solution must be specified unambiguously.

3) Programming. An implementation of the specification is programmed.

4) Verification. The implementation developed in step three is shown to satisfy the formal specification of step two.

There is a certain amount of testing and debugging that goes on at each of these stages until one is satisfied with the current step and moves on to the next. Several verification techniques have been developed to assist in accomplishing step four. However, even after a proof is completed we cannot claim to have a "correct" program, only one that satisfies the given specification.

Rather than write a program, being guided by the specification, and then prove it satisfies the specifications, one can automatically generate a program from the specifications that is guaranteed to preserve the semantics, thus obviatiing the need for the verification step entirely [1].

The truly creative (and most difficult) step in the development of a program is the construction of an acceptable formal specification from an intuitive understanding of the problem. Thus, our efforts should be placed on developing design tools to help with the construction and testing of the specification.

How does one "debug" a specification? We cannot hope to formally prove that a specification is "correct" with respect to our intuition, but we can at least test it to see that it conforms to our intuition in specific cases.

Guttag and Horning [2] present an algebraic specification technique as a design tool. As an example they describe part of the specification of a high-level interface to a flexible display and discuss the analysis of the specification. A salient feature of their approach is the ability to "ask questions" of the specification, derive answers, and change the design if the answers are unacceptable. In this way they hope to test and debug the specification.

I suggest that Horn clauses provide a much better specification language than do algebraic axioms. The two languages are closely related; it is a simple matter to translate between them. The ease of writing a specification in one language versus the other is undoubtedly a matter of personal taste and depends largely on which language one is more familiar with. The same may be said of the readability of a specification. Horn clauses, as well as algebraic axioms, can be analyzed for answers to specific questions and modified accordingly.

The major distinction between the two methods is the manner in which questions can be handled. With the Guttag-Horning approach, an informal question is posed and submitted to an "expert" who reformulates the question, often generalizing it. The questioner must then be convinced that the formal statement developed by the expert does indeed reflect the original question, and an answer to the formal question will provide an answer to the informal one. Then an attempt is made to derive an answer from the axioms.

The same approach may be taken with Horn clauses, but it is not necessary. Since Horn clauses are executable, if the questioner wants to know what happens in a particular case, it is possible to simply "try it and see". The expert will still be needed to develop the specification and to determine what modifications should be made to the specification to change an unacceptable answer, but the "what happens if ...?" questions no longer need be formalized. For example, given the specification detailed in the appendix, and definitions for the primitives (machine dependent) that interface the underlying logic with the commands controlling the appearance of the screen, it is possible to execute logic programs that manipulate the display. Ideally, a "front-end" command language should be provided by the designer(s) that enables users/testers of the design to make their requests of the system without having to write them as Horn clauses.

Once one is satisfied that a Horn clause specification is a reasonable embodiment of one's intuition, the task of refining the specification into an efficient program can proceed. The ability to run a specification makes the problem of testing and debugging it much more tractable.

As an example, I have written the Horn clause specification of the display specified with algebraic axioms by Guttag and Horning. The fundamental assumption is that a user will want to be able to display several distinct blocks of information on the screen at once. The top level concept is that of a *view*. A *view* is a spatial arrangement of *pictures*; a *picture* is a block of displayable information. A picture consists of a boundary, a contents, and a coordinate transformation to be applied in viewing its contents. Examples of pictures are the entire display (with implicit boundary), and the interior of a fixed rectangle on the display; examples of contents are text, figures, and views.

The Guttag-Horning specification of picture is as follows:

Operators:

MakePicture: Contents X *[Coordinate → TruthValues]* X *[Coordinate → Coordinate]* → *Picture*

Picture.Appearance: Picture X *Coordinate → Illumination*

Picture.In: Picture X *Coordinate → TruthValues*

Axioms:

Picture.Appearance(MakePicture(cont, bound, trans), coord) = *Contents.Appearance(cont, trans(coord))*

Picture.In(MakePicture(cont, bound, trans), coord) = *bound(coord)*

The operators are listed first, giving their functionality, then the axioms defining them are given. *MakePicture* is not defined further since it is simply the constructor function for the type *Picture*. The first axiom tells us that the appearance at a given coordinate in a *picture* is determined by the appearance at a coordinate (the result of applying the transformation to the original coordinate) in the *contents* of the *picture*. The second axiom indicates that a coordinate is in a *picture* if it is within the boundary of the *picture* as defined by the function *bound*.

The specification of type *Picture* using Horn clauses is given below. The Horn clause

specification clearly indicates the distinction between constructor functions, such as *make-picture*, and the predicates indicating relationships among their arguments. The type constraints, indicating functionality of the predicates, are given only for the clause(s) defining the type being specified. Type-checking can be included explicitly in each clause, however, we assume the required type is made obvious by consistent naming of variables and choose to leave it out of the rest of the specification for the sake of readability.

Picture(make-picture(cont, bound, trans)) ← Contents(cont), Boundary(bound), Translation(trans)

Picture-Appearance(make-picture(cont, bound, trans), coord, illum) ← Compute-position(coord,trans,coord'), Contents-appearance(cont, coord')

Picture-In(make-picture(cont, bound, trans), coord, tv) ← Lies-in(coord, bound, tv)

In the Guttag-Horning axiomatic specification of the display, a *boundary* is a function from *Coordinate* to *TruthValues* and a *translation* is a function from *Coordinate* to *Coordinate*. Horn clause syntax does not allow functions as arguments, thus I've treated *trans* and *bound* as objects, *Compute-position* is a predicate that accomplishes the translation from *coord* to *coord'* indicated by the Guttag-Horning *trans*, similarly, *Lies-in(coord, bound, tv)* results in *tv* being bound to *true* if and only if the Guttag-Horning *bound(coord)* is *true*, and to *false* if and only if Guttag-Horning *bound(coord)* is *false*. I would not need the predicates *Compute-position* and *Lies-in* if I had an evaluation predicate which accepts a function and its arguments and applies the function to the arguments, such as the LISP "apply". I have decided to remain within first-order logic and the strict limitations of Horn clauses. Others have concerned themselves with the problem of moving to second-order, as shown in the "demonstrate" predicate used by Ken Bowen and Alan Robinson.

The specification of type *View*, given algebraically, is as follows:

Operators:

View.Empty: → View

AddPicture: View X Coordinate X PictureId X Picture → View

View.Appearance: View X Coordinate → Illumination

View.In: View X Coordinate → TruthValues

FindPictures: View X Coordinate → IdList

DeletePicture: View X PictureId → View

Axioms:

View.Appearance(AddPicture(v, coord', id, p), coord) =
if *Picture.In(p, Minus(coord, coord'))*
then *Picture.Appearance(p, Minus(coord, coord'))*
else *View.Appearance(v, coord)*

View.Appearance(View.Empty, coord) intentionally left unspecified

View.In(View.Empty, coord) = False

View.In(AddPicture(v, coord', id, p), coord) =
Picture.In(p, Minus(coord, coord')) ∨ View.In(v, coord)

FindPictures(View.Empty, coord) = IdList.Empty

FindPictures(AddPicture(v, coord', id, p), coord) =
if *Picture.In(p, Minus(coord, coord'))*
then *IdList.Insert(id, FindPictures(v, coord))*
else *FindPictures(v, coord)*

DeletePicture(View.Empty, id) = View.Empty

DeletePicture(AddPicture(v, coord, id', p), id) =
if *PictureId.Equal(id, id')*
then *v*
else *AddPicture(DeletePicture(v, id), coord, id', p)*

Guttag and Horning use the convention of prefixing a function name by the type it is operating on and a dot. In this way they can use the same name for similar functions being defined over several different types. They chose to use a 0-ary function *View.Empty* to indicate the empty view, we use a constant *mt-view*. *AddPicture* is the constructor function for type *View*. *Appearance* and *In* are determined by the components (pictures) making up a view. *FindPictures* is a function that constructs a list of *id*'s of pictures containing a given coordinate. *DeletePicture* deletes a picture, specified by its *id*, from a view.

Again, using Horn clauses, we indicate the types of arguments only in the specification of *View*, and assume the desired types are made apparent by naming of variables.

View(mt-view) ←

View(addpicture(v, c, id, p)) ← *View(v), Coordinate(c), PictureId(id), Picture(p)*

View-Appearance(mt-view, coord, x) ←

As in the algebraic specification, we leave unspecified the appearance of the *mt-view* at any coordinate. Since we have no if-then-else, the axiom describing *View.Appearance* corresponds to two Horn clauses, one for each alternative.

View-Appearance(addpicture(v, coord', id, p), coord, illum) ←
Picture-In(p, minus(coord, coord'), true),
Picture-Appearance(p, minus(coord, coord'), illum)

View-Appearance(addpicture(v, coord', id, p), coord, illum) ←
Picture-In(p, minus(coord, coord'), false),
View-Appearance(v, coord, illum)

View-In(mt-view, coord, false) ←

Horn clauses are not allowed alternative conditions. Thus the second axiom for *View.In* is handled by the following three Horn clauses, one for each alternative making the conclusion *true*, and a third to enable us to derive the fact that a coordinate is not in a *view*.

View-In(addpicture(v, coord', id, p), coord, true) ←
Picture-In(p, minus(coord, coord'), true)

View-In(addpicture(v, coord', id, p), coord, true) ← *View-In(v, coord, true)*

View-In(addpicture(v, coord', id, p), coord, false) ←
Picture-In(p, minus(coord, coord'), false),
View-In(v, coord, false)

FindPictures(mt-view, coord, mt-idlist) ←

FindPictures(addpicture(v, coord', id, p), coord, idlist-insert(id idl)) ←
Picture-In(p, minus(coord, coord'), true),
FindPictures(v, coord, idl)

FindPictures(addpicture(v, coord', id, p), coord, idl) ←
Picture-In(p, minus(coord, coord'), false),
FindPictures(v, coord, idl)

DeletePicture(mt-view, id, mt-view) ←

DeletePicture(addpicture(v, coord, id, p), id, v) ←

DeletePicture(addpicture(v, coord, id', p), id, addpicture(v', coord, id', p)) ←
PictureId-equal(id, id', false), DeletePicture(v, id, v')

FindPictures and *DeletePicture* present no surprises. Again, an if-then-else in an axiom results in two clauses in the Horn clause specification. A complete Horn clause specification of the display is given in the appendix.

In analyzing the specification using the algebraic axioms one needs an expert to go between the questioner and the specification. For example, and informal question asked of Guttag and Horning was: "Is it the case that pictures are not transparent or even translucent? I.e., if two pictures overlap, does the bottom one have no effect on what one sees through the top one?". The question was formalized as:

"Is it true that
$(\forall c\,\ell\,w\,id\,v1\,v2)[Picture.In(w, Minus(c\,\ell')) \rightarrow$
$[View.Appearance(AddPicture(v1\,\ell'\,id\,w)\,\ell) = View.Appearance(AddPicture(v2\,\ell'\,id\,w)\,\ell)]]$?"

The formal question is answered affirmatively, following directly from the first alternative in the first axiom of type *View*.

If we so desired, we could formalize the question to be put to our Horn clause specification and derive the same answer, using the second clause in the definition of *View-Appearance*, but there is no need. Since we can run the Horn clause specification, all the user need do is construct overlapping pictures and look at the result. This is sufficient to answer questions about specific cases. If one is interested in proving general properties, then we must fall back to a formalization of the question and formal derivation of an answer from the specification.

Using Horn clauses as a design tool we enjoy all the benefits of the algebraic approach, and gain the advantage that testing is more easily accomplished. An expert may still be required to develop the design specification and to modify it if necessary, but the analysis of the design can be carried out by people who may be experts in the problem domain but not in the specification language.

References

[1] Davis, R. E., "Generation of Correct Programs from Logic Specifications", Ph.D. Thesis, Board of Information Sciences, University of California, Santa Cruz, 1979.

[2] Guttag, J., and J. Horning, "Formal Specification As a Design Tool", Proceedings of the ACM Symposium on Principles of Programming Languages, 1980.

[3] Kowalski, R., *Logic for Problem Solving*, Elsevier North Holland, Inc., 1979.

Appendix

TYPE Picture

Picture(make-picture(cont, bound, trans)) ←
Contents(cont), Boundary(bound), Translation(trans)

Picture-Appearance(make-picture(cont, bound, trans), coord, illum) ←
Compute-position(coord, trans, coord'), Contents-appearance(cont, coord'),
Contents(cont), Boundary(bound), Translation(trans), Coordinate(coord),
Coordinate(coord'), Illumination(illum)

Picture-In(make-picture(cont, bound, trans), coord, tv) ←
Lies-in(coord, bound, tv), Contents(cont), Boundary(bound),
Translation(trans), Coordinate(coord), Truth-value(tv)

END TYPE Picture

TYPE Contents

Contents(mt-contents) ←

Contents(add-component(cont, comp, coord)) ← Contents(cont), Component(comp), Coordinate(coord)

Contents-Appearance(mt-contents, coord, x) ←

The appearance of an empty contents at a coordinate is intentionally left unspecified as yet.

Contents-Appearance(add-component(cont, comp, coord'), coord, illum) ←
Component-In(comp, minus(coord, coord'), true), Contents-In(cont, coord, true),
Component-Appearance(comp, minus(coord, coord'), illum1),
Contents-Appearance(cont, coord, illum2), Combine(illum1, illum2, illum)

Contents-Appearance(add-component(cont, comp, coord'), coord, illum) ←
Component-In(comp, minus(coord, coord'), true), Contents-In(cont, coord, false),
Component-Appearance(comp, minus(coord, coord'), illum)

Contents-Appearance(add-component(cont, comp, coord'), coord, illum) ←
Component-In(comp, minus(coord, coord'), false), Contents-Appearance(cont, coord, illum)

Contents-In(mt-contents, coord, false) ←

Contents-In(add-component(cont, comp, coord'), coord, true) ←
Component-In(comp, minus(coord, coord'), true)

Contents-In(add-component(cont, comp, coord'), coord, true) ← Contents-In(cont, coord, true)

Contents-In(add-component(cont, comp, coord'), coord, false) ←
Component-In(comp, minus(coord, coord'), false), Contents-In(cont, coord, false)

END TYPE Contents

TYPE Component
Simply the union of *View*, *Text*, and *Figure*.

Component(make-vcomp(view)) ←

Component(make-tcomp(text)) ←

Component(make-fcomp(figure)) ←

Component-Appearance(make-vcomp(view), coord, illum) ←
View-Appearance(view, coord, illum)

Component-Appearance(make-tcomp(text), coord, illum) ←
Text-Appearance(text, coord, illum)

Component-Appearance(make-fcomp(figure), coord, illum) ←
Figure-Appearance(figure, coord, illum)

Component-In(make-vcomp(view), coord, tv) ←
View-In(view, coord, tv)

Component-In(make-tcomp(text), coord, tv) ←
Text-In(text, coord, tv)

Component-In(make-fcomp(figure), coord, tv) ←
Figure-In(figure, coord, tv)

END TYPE Component

TYPE Text

Text(mt-text) ←

Text(text-insert(par, txt)) ← Paragraph(par), Text(txt)

macro: *down(d)* is *minus(coord, times(d, UnitVectorDown)*

Text-Appearance(mt-text, coord, x) ←

Text-Appearance(text-insert(par, txt), coord, illum) ←
Paragraph-In(par, coord, true),
Paragraph-Appearance(p, coord, illum)

Text-Appearance(text-insert(par, txt), coord, illum) ←
Paragraph-In(par, coord, false),
Paragraph-Height(par, d),
Text-Appearance(txt, down(d), illum)

Text-In(mt-text, coord, false) ←

Text-In(text-insert(par, txt), coord, true) ←
Paragraph-In(par, coord, true)

Text-In(text-insert(par, txt), coord, true) ←
Paragraph-Height(par, d),
Text-In(txt, down(d), true)

Text-In(text-insert(par, txt), coord, false) ←
Paragraph-In(par, coord, false),
Paragraph-Height(par, d),
Text-In(txt, down(d), false)

END TYPE Text

TYPE View

View(mt-view) ←

View(add-picture(view, coord, picture-id, picture)) ←

View-Appearance(mt-view, coord, x) ←

Again, we leave unspecified the appearance of the *mt-view* at any coordinate

View-Appearance(add-picture(v, coord', id, p), coord, illum) ←
Picture-In(p, minus(coord, coord'), true),
Picture-Appearance(p, minus(coord, coord'), illum)

View-Appearance(add-picture(v, coord', id, p), coord, illum) ←
Picture-In(p, minus(coord, coord'), false),
View-Appearance(v, coord, illum)

View-In(mt-view, coord, false) ←

View-In(add-picture(v, coord', id, p), coord, true) ←
Picture-In(p, minus(coord, coord'), true)

View-In(add-picture(v, coord', id, p), coord, true) ←
View-In(v, coord, true)

View-In(add-picture(v, coord', id, p), coord, false) ←
Picture-In(p, minus(coord, coord'), false),
View-In(v, coord, false)

FindPictures(mt-view, coord, mt-idlist) ←

FindPictures(add-picture(v, coord', id, p),
coord, idlist-insert(id idl)) ←
Picture-In(p, minus(coord, coord'), true),
FindPictures(v, coord, idl)

FindPictures(add-picture(v, coord', id, p), coord, idl) ←
Picture-In(p, minus(coord, coord'), false),
FindPictures(v, coord, idl)

DeletePicture(mt-view, id, mt-view) ←

DeletePicture(add-picture(v, coord, id, p), id, v) ←

DeletePicture(add-picture(v, coord, id', p),
id, add-picture(v', coord, id', p)) ←
PictureId-equal(id, id', false),
DeletePicture(v, id, v')

END TYPE View

TYPE Idlist

Idlist(mt-idlist) ←

Idlist(idlist-insert(id, idl)) ←
PictureId(id), Idlist(idl)

END TYPE Idlist

TYPE Paragraph
macro: Down(d) is Minus(coord, Times(d, UnitVectorDown)

Paragraph(make-paragraph(parlooks, eng-string)) ←

Par-Firstline(make-paragraph(look, s), line) ←
Parlooks-width(look, w),
EngString-Firstline(s, w, line)

Par-Balance(make-paragraph(look, s),
make-paragraph(look, s')) ←
Parlooks-width(look,w),
EngString-Balance(s, w)

Par-Null(make-paragraph(look, s), tv) ←
String-Null(s, tv)

Par-Space(make-paragraph(look, s), dist) ←
ParLooks-space(look, dist)

Par-Height(p, dist) ←
Par-Null(p, true), Par-Space(p, dist)

Par-Height(p, dist1 + dist2) ← Par-Null(p,false),
Par-Firstline(p, line), Line-Height(line, dist1),
Par-Balance(p, p'), Par-Height(p', dist2)

Par-In(p, coord, true) ← Par-Null(p, false),
Par-Firstline(p, line), Par-Space(p, dist1),
Line-Ascent(line, dist2),
Line-In(line, Down(dist1 + dist2), true)

Par-In(p, coord, true) ← Par-Null(p, false),
Par-Balance(p, p'), Par-Firstline(p, line),
Line-Height(line, dist), Par-In(p', Down(dist), true)

Par-In(p, coord, false) ← Par-Null(p, true)

Par-In(p, coord, false) ← Par-Null(p, false),
Par-Firstline(p, line), Par-Space(p, dist1),
Line-Ascent(line, dist2), Line-In(line,
Down(dist1 + dist2), false), Par-Balance(p, p'),
Line-Height(line, dist), Par-In(p', Down(dist), false)

Par-Appearance(p, coord, illum) ←
Par-Firstline(p, line), Par-Space(p, dist1),
Line-Ascent(line, dist2),
Line-In(line, Down(dist1 + dist2), true),
Line-Appearance(line, Down(dist1 + dist2), illum)

Par-Appearance(p, coord, illum) ←
Par-Firstline(p, line), Par-Space(p, dist1),
Line-Ascent(line, dist2),
Line-In(line, Down(dist1 + dist2), false),
Par-Balance(p, p'), Line-Height(line, dist),
Par-Appearance(p', Down(dist), illum)

END TYPE PARAGRAPH

TYPE LINE

Line(mt-line) ←

Line(line-insert(c, l)) ← Character(c), Line(l)

Line-Appearance(mt-line, coord, illum)
intentionally left unspecified
macro: *Right(d)* is *Minus(coord, Times(d, UnitVectorRight)*

Line-Appearance(line-insert(c, ln), coord, illum) ←
Character-In(c, coord, true),
Character-Appearance(c, coord, illum)

Line-Appearance(line-insert(c, ln), coord, illum) ←
Character-In(c, coord, false), Character-width(c, w),
Line-Appearance(ln, Right(w), illum)

Line-In(mt-line, coord, false) ←

Line-In(line-insert(c, ln), coord, true) ←
Character-In(c, coord, true)

Line-In(line-insert(c, ln), coord, true) ←
Character-width(c, w),
Line-In(ln, Right(w), true)

Line-Height(ln, d1+d2) ←
Line-Ascent(ln, d1), Line-Descent(ln, d2)

Line-Ascent(mt-line, 0) ←

Line-Ascent(line-insert(c, ln), d) ←
Character-Ascent(c, d1), Line-Ascent(ln, d2),
Maximum(d1, d2, d)

Line-Descent(mt-line, 0) ←

Line-Descent(line-insert(c, ln), d) ←
Character-Descent(c, d1), Line-Descent(ln, d2),
Maximum(d1, d2, d)

Maximum(x, y, x) ← LessThan(x, y, false)

Maximum(x, y, y) ← LessThan(y, x, false)

I have assumed the existence of a *LessThan* predicate that returns *true* or *false*

END TYPE LINE

TYPE EnglishString

EngString(mt-string) ←

EngString(string-insert(char, string) ←
Character(char), EngString(string)

EngString-Firstline(mt-string, dist, mt-line) ←

```
EngString-Firstline(string-insert(c, s), d, c) ←
    SplitHere(s, c, d, true)

EngString-Firstline(string-insert(c, s), d,
                    line-insert(c, line)) ←
    SplitHere(s, c, d, false), Character-width(c, w),
    EngString-Firstline(s, d-w, line)

EngString-Balance(mt-string, d, mt-string)←

EngString-Balance(string-insert(c, s), d, s) ←
    SplitHere(s, c, d, true)

EngString-Balance(string-insert(c, s), d, str) ←
    Character-width(c, w), EngString-Balance(s, d-w, str)

SplitHere(mt-string, c, d,true) ←

SplitHere(string-insert(c', s), c, d, true) ←
    Character-Equal(c', quoteCR, true)

SplitHere(string-insert(c', s), c, d, true) ←
    Character-Equal(c', quoteSpace, false),
    Character-width(c, w), Character-width(c', w'),
    LessThan(d, w+w', true)

SplitHere(string-insert(c', s), c, d, true) ←
    LexicalBreak(c, c', true), Character-width(c, w),
    WordFits(s, c', d-w, false)

SplitHere(string-insert(c', s), c, d, false) ←
    Character-Equal(c', quoteCR, false),
    Character-Equal(c', quoteSpace, tv1),
    Character-width(c, w), Character-width(c', w'),
    LessThan(d, w+w', tv2), Not(tv2, tv2'),
    Or(tv1, tv2', true),
    LexicalBreak(c, c', tv3), Not(tv3, tv3'),
    Character-width(c, w), WordFits(s, c', d-w, tv4),
    Or(tv3', tv4, true)

Not(true, false) ←

Not(false, true) ←

Or(true, tv, true) ←

Or(tv, true, true) ←

Or(false, false, false) ←

LexicalBreak(current, next, true) ←
    Character-Equal(current, quoteSpace, tru
    Character-Equal(next, quoteSpace, fals

LexicalBreak(current, next, true) ←
    Character-Equal(current, quoteHyphen, tr
    Character-Equal(next, quoteHyphen, fals
    Character-Equal(next, quoteSpace, fals

LexicalBreak(current, next, false) ←
    Character-Equal(next, quoteSpace, true

LexicalBreak(current, next, false) ←
    Character-Equal(current, quoteSpace, fals
    Character-Equal(current, quoteHyphen, fal

LexicalBreak(current, next, false) ←
    Character-Equal(current, quoteSpace, fals
    Character-Equal(next, quoteHyphen, true

WordFits(mt-string, c, d, true) ←

WordFits(string-insert(c', s), c, d, true) ←
    Character-Equal(c, quoteCR, true)

WordFits(string-insert(c', s), c, d, true) ←
    Character-Equal(c, quoteSpace, true)

WordFits(string-insert(c', s), c, d, true) ←
    LexicalBreak(c, c'true), Character-width(c, w
    LessThan(d, w, false)

WordFits(string-insert(c', s), c, d, true) ←
    Character-width(c, w), Character-width(c', s
    LessThan(d, w+w', false),
    WordFits(s, c', d-w, true)

WordFits(string-insert(c', s), c, d, false) ←
    Character-Equal(c, quoteCR, false),
    Character-Equal(c, quoteSpace, false),
    LexicalBreak(c, c'tv1), Not(tv1, tv1'),
    Character-width(c, w), LessThan(d, w, tv2),
    Or(tv1', tv2), Character-width(c', w'),
    LessThan(d, w+w', tv3),
    WordFits(s, c', d-w, tv4), Not(tv4, tv4'),
    Or(tv3, tv4', true)

String-Null(mt-string, true) ←

String-Null(string-insert(c, s), false) ←

END TYPE EnglishString
```

TYPE Character

Character(make-char(code, fig, ascent, descent, width)) ←
CharacterCode(code), Figure(fig), Distance(ascent), Distance(descent), Distance(width)

Character-width(make-char(code, fig, ascent, descent, width), width) ←

Char-equal(make-char(code, fig, asc, des, w), make-char(code, fig', asc', des', w')) ←

Character-Ascent(make-char(code, fig, ascent, descent, width), ascent) ←

Character-Descent(make-char(code, fig, ascent, descent, width), descent) ←

Char-Appearance(make-char(cd, f, a, d, w), coord, illum) ←
Figure-Appearance(f, coord, illum)

Char-In(make-char(cd, f, a, d, w), coord, true) ←
Figure-In(f, coord, true),
Increasing(a, project(coord, unitvectordown), d, true),
Increasing(0, project(coord, unitvectorright), w, true)

Char-In(make-char(cd, f, a, d, w), coord, false) ←
Figure-In(f, coord, false)

Char-In(make-char(cd, f, a, d, w), coord, false) ←
Increasing(a, project(coord, unitvectordown), d, false)

Char-In(make-char(cd, f, a, d, w), coord, false) ←
Increasing(0, project(coord, unitvectorright), w, false)

END TYPE Character

TYPE Figure

Type *Figure* will necessarily include specifications of the *Appearance* and *In* predicates for the type. This type is left unspecified as it may be dependent upon the target system. Clearly, a more flexible specification of *Figure* is possible for a bit mapped display than is possible given a character mapped display.

END TYPE Figure

TYPE Coordinate

This type is not yet specified. It needs at least the functions *minus*, *times*, and *project*, and the constants *unitvectorright* and *unitvectordown*. A 2-dimensional vector space would do, and one might consider this a primitive type of the system one is using.

END TYPE Coordinate

TYPE Distance

Again, this type should be available already on the target system. One simply needs to define the mapping from the predicate form used in the Horn clauses to the functions available.

END TYPE Distance

TYPE Font

Font(mt-font) ←

Font(addcharacter(font, char)) ←

Lookup(mt-font, code, x) ←
another unspecified pathological case

Lookup(addcharacter(fnt, make-char(cd, f, a, d, w)), cd, make-char(cd, f, a, d, w)) ←

Lookup(addcharacter(fnt, make-char(cd, f, a, d, w)), code, c) ←
CharacterCode-Equal(cd, code, false),
Lookup(fnt, code, c)

END TYPE Font

PRIMITIVES

The primitive predicates that relate the logic to a particular system include the types *Boundary*, *Translation*, *Coordinate*, *Illumination*, and the predicates, such as *Compute-Position*, *Lies-In*, and *Minus*, that operate on them. For example, an *illumination* may be one of two values (white/black), one of several shades of grey, or a more complex combination of hue and intensity, depending on the capabilities of the system one is designing.

COMPUTING ALGORITHMS AND PROVING PROPERTIES BY COMPUTING TERMS

by Enrico Pagello (+) and Silvio Valentini (=)

(+) LADSEB-CNR, Corso Stati Uniti 4, Padova, Italy
(=) Istituto di Matematica, Università di Siena, Siena, Ita

Abstract

We want to do program transformations within the framework of th PROLOG programming language. This may be done in an interesting way making an appeal to the metatheory by using techniques like passing terms as predicates and so on. The metatheory knowledge helps us also to proof termination of some example program.
Keywords: Horn-clauses, metatheory, correctness proof, program termination, PROLOG programming language.

1. Introduction

Program transformations are an uptodate problem in any syntactical frame choosen as a programming language; based on the procedural interpretation of the Horn-clauses /3/, the programming language PROLOG /7/ offers an interesting frame for manipulating techniques: making an appeal to the metatheory may be the way to do this.

In this paper we will try to discuss such a possibility by presenting a set of examples which fully exploit the PROLOG programming style:by passing terms as predicates and viceversa we transform clear specifications via correctness-preserving transformations into pure Horn-Clause programs.

As clearly stated in /2/ , the need for correlating a suitable specification set to a Horn-Clause program prevails to the same extent as it does to a procedural program; the problem is now to be able to use computationally both the specifications and a suitable set of sentences specifying theoretical and metatheoretical (like e.g. mathematical induction) properties of the choosen problem domain.

Furthermore in /4/ was well founded that algorithms can be considered made by two parts: a LOGIC component which specifies the problem domain part of the algorithm, i.e. the algorithm's meaning, and a CONTROL component which is responsible for the strategy choosen to solve the problem, i.e. the algorithm behaviour.

With this paper we intend to study how to alter the logic component of algorithms to embed the ability of controlling the computation steps in the logic specification of the problem; by appealing to the metatheory this embedding is easier and deterministic algorithms are mostly obtained from non deterministic ones.

In section 2 we will introduce a set of Horn-Clauses for computing over natural numbers; in section 3 we will explain our use of the metatheory in a PROLOG-like programming style; in section 4 we will give more examples either about the computation of terms within predicates and about proving properties by using induction via sets of Horn-Clauses.

We think that our approach, even if quite simple, is a neat and clear suggestion to do everything (proving and improving programs) within the same level, i.e. PROLOG programs; its capability should become evident if applied to a richer domain, like the one of binary trees, for example, and if combined with clever strategies. These must be able to choose on what variable to try the induction and how to explore the search space.

2. A basic Horn-Clause set for Number Theory

Let us refer to the usual natural number domain, that must be intended as the set closed under the one-to-one function successor and containing the "0" element; this compels us to use the language, over the Predicate Calculus, with the "0" constant and only the successor term "s(x)".

We reserve the right to introduce new names for relations over the domain, i.e. computable predicates, by giving the Horn Clause programs for our new axioms. We will try to use the induction property at the theory level by creating suitable Horn-clause programs.

First we grow our theory by introducing new axioms /6/ for equality which add to our language new predicates EQ and DIFF:

let be Ax1: $x=y \rightarrow (x=z \rightarrow y=z)$
Ax2: $x=y \rightarrow s(x)=s(y)$
Ax3: $0 \neq s(x)$
Ax4: $x \neq y \rightarrow s(x) \neq s(y)$

We naturally obtain the following Horn-Clause program:

EQ: 1. EQ(y,z)←EQ(x,y),EQ(x,z)
2. EQ(s(x),s(y))←EQ(x,y)

DIFF: 3. DIFF(0,s(x))←
4. DIFF(s(x),0)←
5. DIFF(s(x),s(y))←DIFF(x,y)

where lines 1, 2, 3&4, 5 follow directly from axioms 1, 2, 3, 4. Then we turn to the axioms for plus:

let be Ax5: x+0=x
Ax6: x+s(y)=s(x+y)

By using the following equivalence

$$x+y=z \stackrel{def}{\equiv} PLUS(x,y,z)$$

we write the following program

PLUS: 6. PLUS(x,0,x)←
7. PLUS(x,s(y),s(x ▲ y))←

where the term ⚠ now is not defined in our theory.

There are two main ways to manipulate the specification set for obtaining a Horn-Clause program: the first one is in the following, while the second will be in the next section.

By introducing the output variable within the predicate PLUS, we obtain the equivalence:

$$x+s(y)=s(z) \text{ iff } x+y=z$$

which gives the following program:

PLUS: 6. PLUS(x,0,x)←
7. PLUS(x,s(y),s(z))←PLUS(x,y,z)

Let us try to find a theorem about relating the predicate EQ to PLUS; the same previous idea applied to Ax5 states the equivalence

$$x+0=z \quad \text{iff} \quad z=x$$

which gives the following program:

EQPLUS: 8. EQ(x,z)←PLUS(x,0,z)

We are ready now to prove theorems about our theory:

TH1: 0=0

Theorem 1, under the Horn-Clause form

←EQ(0,0)

trivially follows from clauses 1 ÷ 8 ; so the following clause may be added:

9. EQ(0,0)←

3. Computing and proving by metatheory

Let us introduce now a point of view analogous to that one developed in /8/ about syntax and semantics attachments in the evaluation processes.

If we consider again the Axioms 5&6, we may write immediately:

$$x+s(y)=z \quad \text{iff} \quad z=s(x+y)$$

Let us introduce a new predicate SUCC(x,y) to obtain the program:

PLUS: 6. PLUS(x,0,x)←
7! PLUS(x,s(y),z)←SUCC(plus(x,y),z)

Two problems arise now about how to grow our theory: first we need to define SUCC from correct axioms about the term "s(x)"; second the new term "plus(x,y)" does not exist in our theory, so we need to appeal to the metatheory to compute it; we call upon a new predicate H (for "HOLD") for the metatheoretical knowledge.

By definition of successor term we have:

$$SUCC(x,y) \stackrel{def}{=} y=s(x)$$

from which:

```
SUCC:     10. SUCC(0,s(0))←
          11. SUCC(s(x),s(s(x)))←
```

where 10 and 11 have been choosen to avoid non-determinism with 7'.

The metatheoretical knowledge own by the predicate H gives the following program:

```
HOLD_PLUS:12. SUCC(plus(x,y),z)←SUCC(t,z),H(plus(x,y),t)
          13. H(plus(x,y),z)←PLUS(x,y,z)
```

Here the clauses 6, 7', 10 and 11 describe the syntax of sum and successor functions; with the clause 12 we claim that the function "plus(x,y)" may be computed through its semantics; in the clause 13 the predicate H evaluates the semantics of the term "plus(x,y)" through its syntactic description.

Please note that the circularity of the program is justified by the completeness theorem for predicate logic; the metatheoretical predicate HOLD allows reduction from higher order functional calculus to first order predicate calculus.

Let us now exploit the full capability of our predicate H: we want to try to prove the simple theorem:

TH2: $\forall x.\, x=x$

We pass it , as a term, to the predicate H which tries to compute it making appeal to its metatheoretical knowledge:

```
←H(for-all(x,eq(x,x)))
```

H calls for a program H' which has knowledge of resolution and induction properties:

```
14. H(x)←H'(not(x))
```

where we suppose for the moment to have an effective Horn-Clause program which defines H' /4/ which is able to carry out resolution.

First, H' tries to solve directly by resolution and fails as following:

```
←H'(not(for-all(x,eq(x,x))))
←H'(eq(a,a))
←Ajout(←EQ(a,a))
  ...
no answer in time
```

where Ajout is the PROLOG usual predicate /7/.

Second, H' tries by induction over x and succeeds as following (only significant steps are illustrated) :

```
←H'(not(for-all(x,eq(x,x))))
←H'(not(and(eq(0,0),for-all(x,impl(eq(x,x),eq(s(x),s(x)>
  ...
←H'(or(not(eq(0,0)),est(x,not(impl(eq(x,x),eq(s(x),s(x)>
  ...
←H'(or(not(eq(0,0)),and(eq(a,a),not(eq(s(a),s(a)>
  ...
←Ajout( EQ(a,a)←EQ(0,0)),Ajout(←EQ(0,0),EQ(s(a),s(a)))
      from which
←EQ(0,0),EQ(s(a),s(a))
←EQ(s(a),s(a))
←EQ(a,a)
←EQ(0,0)
  □
```

Let us discuss which problems seems to arise now for constructing such a H' program. For some theorems the result of the application of H is a simple Horn-Clause program easily refutable, as in the previous case. But in most cases, or we cannot find a Horn-Clause set as a result to be passed to the Ajout predicate, or, even if this happens, it becomes very hard to explore the search space for refutation.

Anyhow, we think that such an approach can be usefully exploited because of its circularity: the fact that properties we want to prove about algorithms are all expressible from the same PROLOG program, recalls the circularity of LISP between programs and data.

PROLOG offers many tricks for helping resolution of non-Horn-Clauses; in case predicates involved in a program are recursive, it is possible to use the NON-with-failure trick of PROLOG, as we will illustrate in the next section. Furthermore we think that few and simple ways to compute non-Horn-Clauses may succeed in significant computations /1/,/4/ and that simple strategies may be introduced via metatheory as suggested in /8/.

We want also to suggest to add procedures which tempt to detect if predicates in the programs are recursive: for example detecting loops of the form: "P(x)←P(f(x)),..." ; such a technique is necessary for ex. for correctly terminating the proof of TH2, because the clause "←EQ(a,a)" may resolve with clause 1 entering in a loop.

4. More Examples

Let us extend our theory by introducing programs for computing the product whose axioms are

Ax7: x*0=0
Ax8: x*s(y)=x+(x*y)

where we use the equivalence

x*s(y)=z iff x+(x*y)=z

to obtain the following program

```
PROD:    15. PROD(x,0,0)←
         16. PROD(x,s(y),z)←PLUS(x,prod(x,y),z)
         17. PLUS(x,prod(x,y),z)←PLUS(x,t,z),H(prod(x,y),t)
         18. H(prod(x,y),t)←PROD(x,y,t)
```

where we have justified the introduction of the new term "prod(x,y)" by teaching the predicate H.

Because we consider only recursive functions over natural numbers and because the Horn-Clause syntax is computationally complete, it does exist a universal program for computing terms of higher order whichever form they assume; we think that this allows easily to manipulate programs, every time we are able to describe their semantics.

To clarify the role of the metatheory let us consider the following specification for the recursive schema:

FUN_SCHEMA: f(x,y) ≡ if x=0 then 1 else f(x-1,f(x,y))

If we translate it into a Horn-Clause program, by using the output variables in the simple standard way, we obtain the following program:

```
FVAL:    19'. F(0,y,1)←
         20'. F(s(x),y,z)←F(x,t,z),F(s(x),y,t)
```

where whichever literal we choose to activate, we compute the same result corresponding to a "call by value" rule for the functional schema, i.e. "if x=0 then 1 else ω" .

Instead, if we transform the given specification by carefully using our predicate HOLD, we may compute at the semantics level by the program:

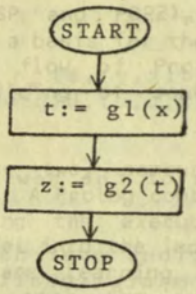
```
FNAME:
         19. F(0,y,1)←
         20. F(s(x),y,z)←F(x,f(s(x),y),z)
         21. F(x,f(w,y),z)←F(x,t,z),H(f(w,y),t)
         22. H(f(w,y),t)←F(w,y,t)
```

which transforms,with a complete search strategy, the original functional specification into a program computing the least fixpoint of "fun-schema", i.e. "if $x \geq 0$ then 1 else ω".

To further clarify the role of metatheory let us consider the problem of translating a given structured flowchart program over the Manna's statements /5/.

Let us consider the first basic flowchart schema which computes the function:

COMP_SCHEMA: p1(x) ≡ g2(g1(x))

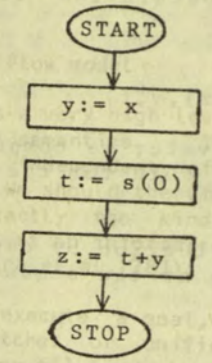


By using metatheory in a trivial application, we obtain the following Horn-Clause program:

COMP: 23. P1(x,z)←G2(t,z),H(g1(x),t)
24. H(g1(x),y)←G1(x,y)

Let us remember for completeness how assignements work by illustrating the following example:

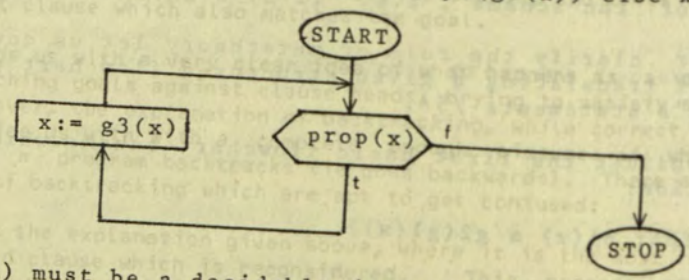
ASSIGN_SCHEMA: p2(x) ≡ i(x)+s(0)



ASSIGN: 25. P2(x,z)←Q(x,x,z)
26. Q(x,y,z)←EQ(t,s(0)),R(x,y,t,z)
27. R(x,y,t,z)←PLUS(y,t,z)

Let us consider the second basic flowchart schema:

LOOP_SCHEMA: p3(x) ≡ if prop(x) then p3(g3(x)) else x



where prop(x) must be a decidable (recursive) predicate to satisfy the Manna's statements /5/.

The use of metatheory suggest the following tranlation:

```
LOOP:    28. P3(x,z)← H(prop(x),T),P3(t,z),H(g3(x),t)
         29. H(g3(x),y)← G3(x,y)
         30. P3(x,x)←H(prop(x),F)
```

Now H may easily solve the problem of determining the truth value of prop(x) using the NON trick of PROLOG:

```
         31. H(prop(x),T)←PROP(x)
         32. H(prop(x),F)←NON(PROP(x))
         33. NON(*X) ← *X,/,FAIL
         34. NON(*X)
```

where the predicate H in this case has knowledge of the way to find the truth value of prop(x) by using the extratheoretical predicate "slash" of PROLOG.

We need however to proof the correctness of 33 & 34 clauses; indeed:

if $Ax \vdash PROP(x)$ then $Ax \models PROP(x)$

if $Ax \not\vdash PROP(x)$ then $\neg (Ax \underset{m}{\cup} \neg PROP(x) \vdash \bot) \equiv Ax \models \neg PROP(x)$

because of the decidability of PROP(x).

Because our transformations of axioms and flowchart specifications are naturally correct, we can use axioms or functional schemata as independent specifications for our Horn-Clauses Programs. Furthermore we suggest that the clauses obtained by these transfor_ mations of flowchart programs into Horn-Clauses may be used to build the inductive assertions usefull to proof the correctness of the program, in a symmetric way.

However it remains the problem of verifying the termination of programs; we may use again whenever possible the predicate H'.

First we express termination for PLUS as:

TH3: $\forall x \forall y \exists z . PLUS(x,y,z)$

where, by partial correctness of our transformations, if such z exists, it is unique.

The proof of TH3 follows now from the usual appeal to metatheory knoledge. Infact we might have the derivation:

```
        ←H(for-all(x,for-all(y,est(z,plus(x,y,z)))))
          ...
        ←Ajout(PLUS(x,b,f(x)) ← PLUS(a,0,z)),Ajout(← PLUS(a,0,z),
                                                    PLUS(c,s(b),w)
          and so
        ←PLUS(a,0,z), PLUS(c,s(b),w)
        ←PLUS(c,s(b),w)
          ...
        ←PLUS(c,b,w')
        ←PLUS(a,0,z)
          □
```

Our program about PLUS then, obtained by correct transformations over Peano Axioms, is totally correct because TH3 asserts its termination.

From TH3 we may add to our axioms the following clause:

35. PLUS(x,y,g(x,y))←

Now we can proove a termination theorem for PROD:

TH4: ∀ x ∀ y ∃z.PROD(x,y,z)

proof: ←H(for-all(x,for-all(y,est(z,prod(x,y,z)))))
...
←Ajout(PROD(x,b,f(x)) ← PROD(a,0,z)),
Ajout(← PROD(a,0,z),
PROD(c,s(b),w))

and so

←PROD(a,0,z),PROD(c,s(b),w)
←PROD(c,s(b),w)
←PLUS(c,prod(c,b),w)
...
←PROD(a,0,w),PLUS(c,f(c),z)
←PLUS(c,f(c),z)
□

TH4 asserts that our program, obtained by correct transformations over Peano Axioms, is terminating over all inputs; therefore the program is totally correct

References

/1/ Eder G., "A PROLOG interpreter for non-Horn-Clauses" Dept. of A.I., Research Report n. 26, University of Edinburgh, 1976

/2/ Hogger C.J., "Derivation of Logic Programs" Draft, Imperial College, University of London, August 1979

/3/ Kowalski R., "Predicate Logic as a Programming Language" Proceedings of IFIP 74, Stockolm

/4/ Kowalski R., "Logic for Problem Solving" North Holland, 1979

/5/ Manna Z., "Mathematical Theory of Computation" Mc Graw Hill 1974

/6/ Mendelson E., "Introduction to Mathematical Logic" Van Nostrand, 1964

/7/ Roussel P., "PROLOG, manuel de reference e d'utilization" Technical report, Group d'I. A. , Universitè de Marseille a Luminy, 1975

/8/ Weyrauch R., "Prolegomena to a Theory of Formal Reasoning" Memo AIM-315, A.I. Lab., Stanford Un., Stanford 1978

UNDERSTANDING THE CONTROL FLOW OF PROLOG PROGRAMS

by
Lawrence Byrd
Department of Artificial Intelligence
University of Edinburgh
Scotland

1. Introduction

This paper is an informal discussion of some of the practical problems involved in teaching and using current Prolog systems. My concern is with the actual execution of Prolog programs - how is it possible to understand and follow such executions, and what sorts of system facilities would assist this task? I shall motivate these questions, and then go on to describe general model of Prolog control flow. This model has been used as the basis for implementations of practical debugging packages for Prolog systems on the DEC-10 and the PDP-11.

I will be assuming familiarity with the language Prolog [Roussel 75], and knowledge of the DEC-10 Prolog implementation [Pereira et al. 78, Warren 77] would also be useful. I shall also be restricting my discussion to Prolog programs executed with the standard, left-right, depth-first, control strategy. It would be interesting to try and extend what follows to take into account more general strategies.

Prolog programs can be read declaratively as collections of predicate calculus clauses. It is possible, indeed desirable, to utilise the advantages following from this when trying to understand Prolog programs, and when trying to teach Prolog programming (see for example [Kowalski 80] for such a general Logic Programming approach). In this paper, however, I shall be discussing Prolog in procedural terms, and assuming that it is necessary to teach this, and to understand many programs in this way.

In order to do this we require a clear model of the execution of Prolog programs which can be used in teaching Prolog programming, and which also provides enough sophistication to form a basis for practical tracing and debugging tools. Thus, I see the specification of the requirements of such a model being guided by related work in two different areas:

1. The teaching of programming. Recent work in this field has emphasised the importance of specifying an underlying "notional machine", in order to give novices a framework for understanding the various operations of the programming language involved (see [du Boulay 80a], [du Boulay 80b]). A model explaining the control flow of Prolog programs should provide such a "notional machine", and this model should be properly integrated into the Prolog system(s) used, so that novices can follow their programs in terms of the operations given by the model.

2. The development of interactive programming environments. The past decade has seen the development of more and more sophisticated tools for the practical development of large programs. Within the Artificial Intelligence community, complex programming systems (such as INTERLISP, [Teitleman 75]) are now almost taken for granted. Interactive debugging facilities form an important part of these systems. Many of us would like to have similar facilities available in the Prolog systems we use. In functional languages

(eg LISP and POP2), the concepts of function entry and return, provide a basis for the debugging mechanisms. A model of the control flow of Prolog programs must provide a similar basis for the building of sophisticated debugging tools within Prolog systems.

I see these two, somewhat different, requirements being related in the following way. A Prolog control flow model must provide a simple, teachable way of viewing the execution of a Prolog program. It must be possible to build this model into the implementations of our Prolog interpreters, so that novices who are learning the language can sit down and exhaustively follow their programs, with the system explicitly going through the operations involved. This will provide them with a complete trace of the program's execution. Such tracing will obviously be helpful to more advanced users who wish to debug their programs. However, exhaustive tracing is unlikely to be acceptable to such users when they are debugging large programs. Satisfying their needs involves selectively restricting the amount of information presented, and also allowing access to additional information and control options. Thus, I hope to accommodate both requirements by seeing one as merely being a sophisticated enhancement of the other. I would like to believe that this simple relationship can be maintained, even when the facilities have to be extended to meet further demands.

2. Control Flow model

Prolog is a very high level language which, none the less, has a very simple procedural semantics. The procedural operations of the language are completely independent of any lower level operations (such as those of the machine). We should therefore expect the procedural semantics of Prolog to provide exactly the kind of model I was asking for in the previous section. Let us look at an informal definition of Prolog's procedural semantics (taken from [Pereira et al. 78]):

> To execute a goal, the system searches for the first clause whose head matches or unifies with the goal. The unification process [Robinson 65] finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then activated by executing in turn, from left to right, each of the goals (if any) in its body. If at any time the system fails to find a match for a goal, it backtracks, ie. it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.

This provides us with a very clear idea of what happens as a program goes forwards, matching goals against clause heads, trying to satisfy subgoals, and so on. However, the explanation of backtracking, while correct and precise, does <u>not</u> provide us with with a complete enough picture of what actually happens when a program backtracks (ie goes backwards). There are two usual explanations of backtracking which are apt to get confused:

1. There is the explanation given above, where it is the most recently activated clause which is reconsidered. This process continues backwards until a choice point is found.

2. More usually, backtracking is described as being the remaking of the decision made at the chronologically most recent choice point. This is closer to the "implementors" view and is certainly

reflected in the information provided by previous DEC-10 tracing packages.

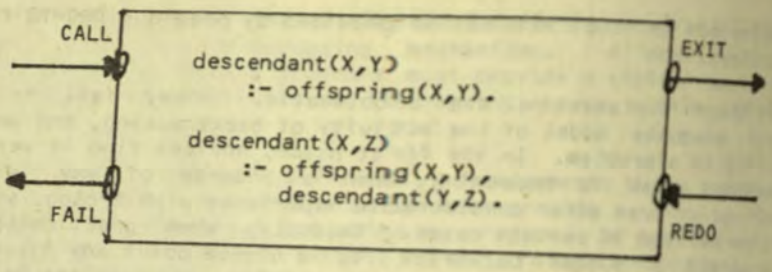
Both of these explanations are problematic. They fail to provide a completely adaquate model of the activity of backtracking, and understanding backtracking is a problem. In the first place, novices find it very difficult to understand what is happening when a program of any size starts backtracking. Even after considerable experience with Prolog, students will claim to be baffled in certain cases. Secondly, when practically debugging large programs a sudden backtrack to a choice point any distance away is highly confusing ("where am I now?"). In neither case does the knowledge that it is the most recent choice that is being redone, provide us with any solution to our difficulties. The models are inadaquate because they do not help us understand the ramifications of the processes they describe. This kind of difficulty was in fact an important argument used against the language PLANNER by Sussman and Steele [Sussman 72]; they argued that programmers simply couldn't understand what was going on!

The first explanation (although very close to what I shall introduce), suffers because it is not always obvious what the last clause activated actually was (we will have conceptually "returned" up the calling chain). More importantly, no way is provided of finding the (reverse) sequence of activated clauses. This is not to say it can't be done, but rather that the so called "explanation" provides us with no clues about how to go about it. Thus, using this model, it will be difficult to understand which procedures will actually be retried on backtracking.

The second explanation is quite simple (and therefore often used), but is even worse in terms of providing an adaquate model for understanding backtracking. The problem here stems from the fact that the backward step (of backtracking) is conceptually very much 'larger' than the forward step (of finding a matching clause). In terms of the actual text of the program, the forward step is reasonably local (we only need to look at some small set of relevant clauses), while the backward step is much more global (the place in the text corresponding to the most recent choice point may be any distance away from the place where the failure occurred). This irregularity causes confusions and difficulties.

I shall now decribe a control flow model where the steps involved are regular, both when going forwards and backwards, and which I believe solves many of the above problems.

The model is based on the idea that we should be able to follow the execution by moving, in some simple way, round the actual text of the program. The program text can be divided into distinct procedures; each procedure consists of some (usually) small sequence of constituent clauses. (Note: a procedure is a set of clauses all of which share a common predicate for their head). Imagine that we place all the clauses for a procedure in a box, and consider the possible control movements in and out of this box. There are four different types of control flow that may occur, I shall consider these to be types of port. Thus, control flow can be seen as movements in and out of procedure boxes, via the ports of these boxes. Let us look at an example Prolog procedure (taken from [Pereira et al. 78]):



I have drawn an explicit box around this procedure, and have also drawn arrows showing the possible control flow movements in and out of this box. The ports have been labelled CALL, EXIT, REDO and FAIL; let us look at each of these in turn:

Call — This arrow represents initial invocation of the procedure. When a goal of the form 'descendant(X,Y)' is required to be satisfied, control passes through the Call port of the descendant box with the intention of matching a component clause and then satisfying any subgoals in the body of that clause. Notice that this is independent of whether such a match is possible; i.e. the box is called, and then such matters are worried about. Textually we can imagine moving our finger to the code for descendant when meeting a call to descendant in some other part of the code.

Exit — This arrow represents a successful return from the procedure. This occurs when the initial goal has been unified with one of the component clauses and any subgoals have been satisfied. Control now passes out of the Exit port of the descendant box. Textually we stop following the code for descendant and go back to the place we came from.

Redo — This arrow indicates that a subsequent goal has failed and that the system is backtracking in an attempt to find alternatives to previous solutions. Control passes through the Redo port of the descendant box. An attempt will now be made to resatisfy one of the component subgoals in the body of the clause that last succeeded; or, if that fails, to completely rematch the original goal with an alternative clause and then try to satisfy any subgoals in the body of this new clause. Textually we follow the code backwards up the way we came looking for new ways of succeeding, possibly dropping down on to another clause and following that if necessary.

Fail — This arrow represents a failure of the initial goal, which might occur if there were no clauses, or if no clause matched, or if subgoals are never satisfied, or if any solution produced is always rejected by later processing. Control now passes out of the Fail port of the descendant box and the systems continues to backtrack. Textually we move our finger back to the code which called this procedure and keep moving backwards up the text looking for choice points.

Using this model, we can follow the execution of a program by following the movement through all the ports traversed during the execution. The control flow within boxes is, of course, followed in terms of the ports for the procedure boxes of the subgoals. Since, in a given program there may be several calls to the same procedure, it is in general necessary to distinguish different invocation boxes. That is to say, we must imagine having a new box for every fresh invocation (ie on every call). This will make sense of the idea of REDOing, (re)EXITing or FAILing the <u>same</u> box as before. Let's follow a simple example where boxes are distinguished by unique invocation numbers

(in parentheses). (I shall be referring back to the alphabetic letters, so ignore them for now):

Given the program

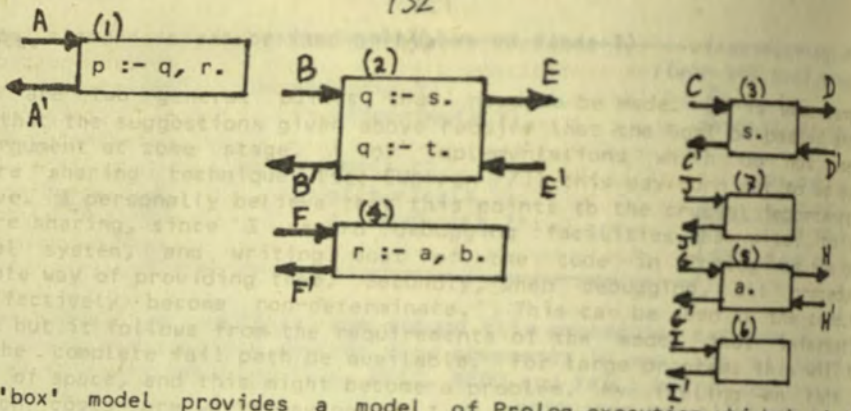
```
p :- q, r.
q :- s.
q :- t.
r :- a, b.
s.
a.

?- p.
```

we get the following execution

(1) Call : p		A
(2) Call : q		B
(3) Call : s		C
(3) Exit : s		D
(2) Exit : q	forwards	E
(4) Call : r		F
(5) Call : a		G
(5) Exit : a		H
(6) Call : b		I
(6) Fail : b		I'
(5) Redo : a		H'
(5) Fail : a		G'
(4) Fail : r	backwards	F'
(2) Redo : q		E'
(3) Redo : s		D'
(3) Fail : s		C'
(7) Call : t	forwards	J
(7) Fail : t		J'
(2) Fail : q	backwards	B'
(1) Fail : p		A'

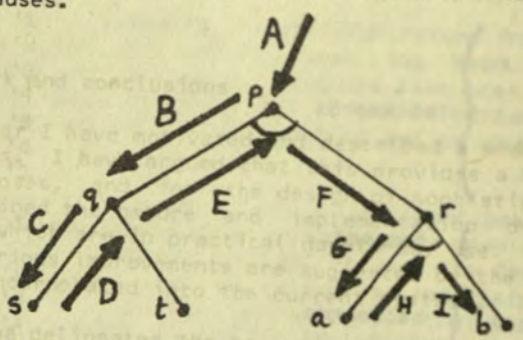
Notice that for any invocation there can only be only one Call and Fail, although there may be an arbitary number of Redos and corresponding Exits (greater than or equal to zero). It is the initial Call which introduces the invocation, and it is here that we first see the new invocation number. Also notice how the backward moving portions of this trace are direct mirror images of previous forward moving portions; where Exit => Redo and Call => Fail. We find a choice point by going back along the path we came until we explicitly find it. If this path is followed in the text, as it can be, then we will be sure about where it is, and we will understand why this was the choice point - we will have noticed that there were no other possibilities after this one. The essential feature of this model is precisely that the intermediate boxes are explicitly given, thus allowing these facts to be seen.

My use of the word backwards reflects the fact that, in the text, we will be actually moving our finger backwards along the goals previously executed. This can be seen graphically by considering the following diagram, where I have explicitly shown the boxes and their interrelationship. I have labelled the arrows at the ports with the letters from the corresponding lines in the above trace.



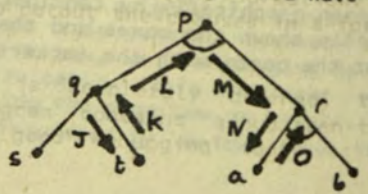
This 'box' model provides a model of Prolog execution which is simple, capturing the essence of the procedural semantics, and understandable, allowing one to follow both activation and backtracking in a consistent way. The control flow can be directly followed in the program text as a sequence of simple regular steps.

I shall now relate this model to the and/or tree representation of Prolog programs. I wish to show that the trace of ports (such as the one above) in the box model corresponds to a simple traversal of the and/or tree representing the program. The following diagram is the and/or tree for the simple program used earlier. I have labelled the nodes with the heads of the particular clauses.



The alphabetically labelled arrows show the traversal that would be used by Prolog when attempting to satisfy the goal 'p'. These labels, which were used in the earlier trace, and on the previous diagram, show how each movement through a port, corresponds to one of the arrows in the tree traversal. Arrival at a node from above corresponds to a Call, and leaving a node (back along the same arc) corresponds to an Exit. After the arrow I, we are forced to backtrack since there is no way of satisfying the goal 'b'. The backward movement through Redo and Fail ports corresponds to a performing all the earlier and/or tree traversals in reverse until we get back to a choice point. These are marked as I',H',G' etc. in the trace. I shall henceforth use the term "reverse traversal" to describe this. Reversing a Call traversal corresponds to a Fail, and reversing an Exit traversal corresponds to a Redo. Again we can see the accuracy of describing backtracking as moving backwards through the program. When the choice point is found then all forward traversals from that point will be new (fresh) traversals. If the goal 't' had been satisfied in the example, then the following would have occurred:

(7) Exit : t K
(2) Exit : q L
(8) Call : r M
(9) Call : a N
(9) Exit : a O



This last point is important when considering the more general case where (possibly different) instantiations will be occurring. At each port of a box the goal involved in the call will be in some particular state of instantiation. At the Call port it will be in its "initial" state, but when the Exit port is reached the goal may be further instantiated due to the actions of the clause used to satisfy the goal. If we Redo this box then further Exits may, of course, have different instantiations. The instantiation state at Redos will always be identical with that of the previous Exit from that box (note: they will NOT in general be identical with the original Call), and similarly, Fails will be identical with Calls. This is natural since Call and Fail are effectively at exactly the same place on the box (similarly Exit and Redo), the only difference is the direction of movement! These facts are reflected in the and/or tree traversal where each traversal arrow has a unique corresponding instantiation state for its particular goal. (Reverse traversals thus have the same instantiation, since they are the same arrows, and remember; reversal corresponds to the mapping Call => Fail, Exit => Redo).

It is convenient at this point to define three concepts in terms of the and/or tree representation. I shall make use of these concepts in what follows.

The ancestor list.

For any port the ancestor list is the sequence of nodes between the corresponding traversal arrow and the top of the tree. Eg, for I (the call of 'b') this would be [r,p].

The complete fail path.

For any port the complete fail path is the sequence of reverse traversals required to reach the last node on the traversal representing a choice point. Eg, for I this would be I',H',G',F',E',D',C'.

The shortest fail path.

For any port the shortest fail path is the (ordered) subset of the complete fail path, which represents the shortest (graph-theoretic) route to the choice point. Eg, for I this would be I',F',E'.

3. Practical Systems

The previous section has outlined a control flow model which is intended to meet my original requirements. I have found it useful for my own thinking and also for the purpose of teaching others exactly how Prolog programs execute. I believe that it provides a suitable "notional machine" for teaching novices about Prolog programming. However, more work would need to be done to verify this. In this section I shall describe the kinds of facility needed in order to meet my second requirement, that of satisfying the debugging needs of sophisticated programmers with large programs. I shall base this description on actually implemented ideas currently used in recent DEC-10 interpreters at Edinburgh.

The user can control the amount of information he receives by both general actions, commands to the interpreter, and specific actions, options available at ports. At the general level debugging can be completely switched on or off. The user can specify which types of port he would like to be prompted at

for specific options; eg, all types, or just Call and Redo ports etc. (This is called leashing). The user can also selectively decide that he would like to always see movement through ports of boxes of particular procedures. This is called setting a spypoint and is similar to the idea of function breaks in POP2 and LISP. Spypoints have turned out to be a major enhancement to the debugging package, and much use is made of them. (This was to be expected!). Information is available about what spypoints are set at any given time and they can also be removed. It is also possible to just start exhaustive tracing of goals right from the beginning.

When control passes through a port a message of the following form is printed:

```
(76) 8 Call : foo([a,b,c],f,_102)
```

This gives the unique invocation number for the box, the recursion depth, the type of port, and the current instantiation state of the goal. The goal is printed using the (new) evaluable predicate 'print', defined more-or-less as follows:

```
print(X) :- portray(X), !.
print(X) :- write(X).
```

'portray' is intended to be user defined, and so the use of 'print' throughout the debugging package provides a handle for various kinds of pretty printing. This has turned out to be extremely useful, especially for masking out arguments known to be very large terms. If the port is leashed (see earlier), then the user is prompted and he then has available a variety of options. These can be divided into:

- Information. The goal can be (re) printed, written in infix or prefix formats, and chosen amounts of the ancestor list can be printed.

- Environmental. The user can break to a new (sub) execution, debugging can be switched off, the whole execution can be aborted or the system left completely.

- Control. Most important of all, the user is provided with ways of controlling what he will see. It is possible to creep to the very next port or leap to the next spypoint. A sequence of creeps will thus produce an exhaustive trace, while leaps will just show selected spypoints along the way. When going into a box (Call and Redo), it is possible to skip so that nothing is seen until the execution returns (to either the Exit or Fail port of that box). This provides a way of masking out irrelevant internal detail, such as deep recursions. At any port it is possible to force a jump to the Call port (retry), or the Fail port (fail) of the box. This is useful in conjunction with other options; eg, arrive at Call, skip to see what happens, if it comes back at Fail or if the Exit instantiation is wrong then retry back to the Call port, and start creeping into the internal execution to see how it happened. This has also been generalised so that retrying can be done to any available earlier invocation boxes (identified by their number). When backtracking it is often not desirable to follow the complete fail path back to some choice point. At Fail and Redo ports it is possible to specify that just the shortest fail path be printed; this will be a sequence of Fails followed by a sequence of Redos.

The control options provide powerful ways of selecting the amount of detail seen, and of moving oneself around an execution if one wishes to reinvestigate

things. The sophisticated user does not have to put up with enormous exhaustive traces; the general and specific actions allow him to be very selective over what is traced.

I have gone over the details rather quickly, but the point I wish to emphasise is that all these features are built as enhancements to the underlying control flow model. It is always possible to follow the full sequence of operations given by the model. Looked at from the other direction, it is much easier to introduce novices to the debugging tools, since they follow in a natural way from the taught model of execution.

4. Implementation

The ideas I have been discussing have been implemented on the DEC-10 as improvements to the DEC-10 interpreter. Slightly more restricted facilities have also been added to the UNIX PDP-11 interpreter developed at Edinburgh ([Mellish 78]). These debugging packages are implemented in Prolog and could therefore, in principle, be moved to other systems. I shall just outline the general principles here. Since the DEC-10 interpreter is itself written in Prolog it is a simple matter to insert something of the following form into the interpretation cycle:

```
break(Goal)
      :- trace(call,Goal),
         ( call(Goal)  ;  trace(fail,Goal), fail ),
         ( trace(exit,Goal)  ;  trace(redo,Goal), fail ).
```

The 'trace' goals will implement the messages and read responses from the user, they implement the four ports of the box around the 'call'. 'break' is therefore a simple skeleton of how the box model can be implemented. The UNIX interpreter is not written in Prolog, but I was able to add spypoints by the trick of asserting an extra clause at the top of the database for each procedure one wanted to spy on. So for a procedure 'example' this would go as follows:

```
example(X,Y,Z)
      :- flipflop,
         !,
         break(example(X,Y,Z)).
```

The procedure 'flipflop' has to be provided and it has the interesting property that it alternatively succeeds and fails! The first time 'example' is called flipflop will be true and so 'break' will be called. When 'break' itself calls 'example' (the Goal), then on this second time flipflop will fail so control drops down onto the real clauses for 'example'. This is a simple way (albeit a hack!) of placing an extra environment around procedures. Notice that given the nature of 'break', only one version of flipflop will be required regardless of how many extra clauses we need to add for different procedures.

Since 'trace' is written in Prolog (on both systems), it is easy to build in the required information and environmental features. The control features are slightly more difficult and there it is necessary to improve 'break'. Both systems make use of extra magic flags (basically assignable locations for integers), which hold status information and which are used to force control around 'break' in various ugly ways (eg, for doing retry etc). For this purpose, both systems provide (secret!) evaluable predicates for manipulating these flags. However, on other systems, use could be made of the database (with probable loss in efficiency), if flags of this kind could not be made

available.

There are two general points that need to be made. It is important to notice that the suggestions given above require that the Goal be passed around as an argument at some stage. For implementations which do not use a structure sharing technique (see [Warren 77]) this may turn out to be quite expensive. I personally believe that this points to the crucial importance of structure sharing, since I regard debugging facilities as vital to any practical system, and writing most of the code in Prolog as the only reasonable way of providing them. Secondly, when debugging, all procedures will effectively become non-determinate. This can be seen in the code for 'break', but it follows from the requirements of the model that information about the complete fail path be available. For large programs this will tie up a lot of space, and this might become a problem. My feeling on this is that such costs are worth paying and I am working on trying to minimise the cost of the current DEC-10 implementation. Since most practical programs make use of cut ("!" - see [Pereira et al. 78]), and this will also cut out some of the debugging information, quite good savings can result without losing the information that the user considers important. The extreme solution is to restrict which ports are actually seen. For example, only showing the Call and Exit ports would not introduce this non-determinacy (and would, incidently, provide the same information as the original DEC-10 tracing). I do not regard such extremes as satisfactory. I feel that there is plenty of work yet to be done, trying to understand what information the user actually requires, and minimising the overheads of providing it.

5. Further work and conclusions

In this paper I have motivated and described a model of the control flow of Prolog programs. I have argued that this provides a suitable basis for both teaching purposes, and for the design of sophisticated debugging tools. I have also outlined the nature and implementation of systems incorporating these ideas which are in practical day-to-day use. I shall now take a brief look at how various improvements are suggested by the model. Given time, this could all be incorporated into the current systems.

The 'box' idea delineates the procedure as the prime focus of attention. The model does not deal with the attempts to match particular clauses. I think this is basically correct, since for programs of any size there will be a large number of irrelevant attempts to match clause heads. However, it does not seem unreasonable, especially for teaching purposes, for some indication to be made of exactly which clause matched. In fact, being able to have the actual clause printed out would be generally useful. It would be nice if one could see both the original clause, and the same clause, but with the current substitutions performed on it. Having this printed out would provide a much better idea of what the variable interconnections were. (It would also make it easier for novices to grasp how Prolog variables actually work). Even more generally, why should't I be able to see the clauses (both original and current) for every goal on the ancestor list, or in the complete fail path? (At the moment I can get no information at all about the fail path!) I am arguing that the three concepts defined earlier (ancestor list, complete fail path, shortest fail path), should be treated seriously as objects available to the user, about which he can gain interesting information. I would also emphasise the importance of trying to output the clauses in a form as close to the original source as possible.

The procedural semantics of Prolog is completely defined, there are no illegal operations. If your program contains a bug then this will be a logical bug, hence the importance of good debugging tools. However, the

Prolog system may provide evaluable predicates which are only defined over some subset of all possible arguments. They will usually produce an error when used incorrectly. When such an error occurs the user would like to know where it occurred. The obvious choice of action is to start the tracing mechanism and wind back to the last Call. Ie, the user should be told of the error, and then he should automatically find himself at the Call port of the procedure where the error occurred. He would then be in a position to discover why it had occurred. (By looking back at the information suggested above, or by retrying from some earlier position etc.) In general, then, ALL errors should cause activation of the tracing mechanisms. Since the control flow model effectively defines the user's virtual machine, errors can be seen as causing a halt at some operation (ie at some port). As a last point, there are many cases where the procedure box is actually empty. In other words, we have an undefined procedure (either we have left something out, or misspelled a goal somewhere). This is almost always a bug. I am convinced that this case should be treated as an error, so that the user is informed and, again, he should find himself at the Call port concerned. This problem is a major source of wasted time on current systems. (The only examples I have seen where failing due to no clauses is significant are clearly database applications which should use a specific "user database" (Cf 'record' etc. on the DEC-10 system)).

Such improvements would greatly enhance the ergonomics of the Prolog systems we use. They would increase the attractiveness of Prolog as a programming language and research tool. However, we cannot always take ideas from other languages straight off the shelf. It is important that such developments are based on models which reflect the actual nature of Prolog. My aim has been to develop such a model for control flow, and to show how certain important features can be constructed in terms of this model.

Acknowledgements

I would like to thank David Warren, Fernando Pereira, Alan Bundy and Chris Mellish for their help and encouragement. I am supported by a British Science Research Council grant, number GR/A/57954.

REFERENCES

[du Boulay 80a]
du Boulay, B. and O'Shea, T.
Teaching Novices Programming.
Research Paper No. 132, Dept. of Artificial Intelligence, Edinburgh., 1980.
To appear in 'Computing Skills and Adaptive Systems', M. Coombs (Ed.), Academic Press.

[du Boulay 80b]
du Boulay, B., O'Shea, T. and Monk, J.
The black box inside the glass box: presenting computing concepts to novices..
Research Paper No. 133, Dept. of Artificial Intelligence, Edinburgh., 1980.

[Kowalski 80]
Kowalski, R. A.
Logic for Problem Solving.
North Holland, 1980.

[Mellish 78]
Mellish C.
The UNIX Prolog System.
Dept of Artificial Intelligence, Univ of Edinburgh, 1978.
[Informal note].

[Pereira et al. 78]
Pereira L M, Pereira F and Warren D H D.
User's Guide to DECsystem-10 Prolog.
Dept. of Artificial Intelligence, Edinburgh., University of Edinburgh, 1978.

[Robinson 65]
Robinson J A.
A machine-oriented logic based on the resolution principle.
JACM 12(1):227-234, December, 1965.

[Roussel 75]
Roussel P.
Prolog : Manuel de Reference et d'Utilisation.
Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1975.

[Sussman 72]
Sussman, G. J. and McDermott, D. V.
Why Conniving is better than Planning.
AI-Memo 255A, MIT AI Laboratory, 1972.

[Teitleman 75]
Teitleman, W.
INTERLISP reference manual
Xerox Palo Alto Research Center, 1975.

[Warren 77]
Warren D H D.
Implementing Prolog - compiling predicate logic programs.
Dept of Artificial Intelligence, Univ of Edinburgh, 1977.
Research Reports 39 & 40.

LOGIC REPRESENTATION OF A CONCURRENT ALGORITHM

C. J. Hogger
Imperial College, University of London, 1980

ABSTRACT

A concurrent algorithm is represented using the 'logic programming' formalism and some general principles are extracted from the formulation. The treatment is shown to be semantically pure and consistent with current approaches to logic program development and verification.

1. INTRODUCTION

Studies in logic programming have usually assumed program execution to be the responsibility of a single processor. This assumption makes it easy to explain conventional PROLOG-like programs using ideas prevalent in other programming formalisms, in particular the idea of interpreting procedure calls as tasks to be completed one at a time by the processor. More recently, significant advances have been made in diversifying the means of specifying control, that is to say, in providing program annotation schemes which indicate control preferences supplementing the usual default strategy. A notable scheme of this kind has been developed by Clark and McCabe (3) and provides an elegant and powerful coroutining facility in their IC-PROLOG system. This allows one to write logically lucid programs whose behaviour is explicitly prescribed in terms of the nature of the data flow through specially annotated variables; the annotations enable call evaluations, by a single processor, to be temporally interleaved, allowing a finer grain of interaction between them than if the annotations were absent. Formerly, comparable behaviour could only have been achieved by executing programs of greater logical intricacy.

Despite the benefits obtained from such elaborations of the control mechanism, there remain numerous simple problems which could be solved yet more efficiently if multi-processor hardware were available. Indeed, Clark and McCabe (3) find an example of this in their discussion of coroutined programs for the eight-queens problem, and Bruynooghe and Clark (2) have already considered program annotations for specifying concurrent call evaluations. The present paper arose from the author's attempt to formulate in logic the essence of a classic problem chosen by Owicki and Gries (7) to demonstrate verification of concurrent ALGOL-like programs. The ideas presented here are somewhat tentative and are chiefly intended to stimulate interest in this and similar problems, rather than to constitute a comprehensive proposal for implementing concurrency. The reader is expected to be familiar with logic programming; this has been comprehensively described by its originator Robert Kowalski (6).

2. ASSUMPTIONS AND NOTATION

Assume that several processors P1,...,Pn are available for solving a collection of calls. Then a goal $\leftarrow A,B$ can be executed using P1 and P2 to solve the two respective subgoals, proceeding concurrently.

For a simple model, imagine that P1 and P2 operate as distinct interpreters sharing access to a single set of procedure definitions. A desire to execute the goal in this way can be expressed by writing it as $\leftarrow A//B$ where // is interpreted logically as conjunction but operationally as a prescription for concurrent execution of A and B. This use of the // symbol is therefore referred to as *conjunctive* concurrency.

Further annotations can be devised to declare which processors are to deal with which calls, if this is prescribable, or else processor allocation can be decided dynamically by the implementation. The details of such arrangements are not relevant to what follows and so are not discussed further.

Although concurrent executions are easiest to conceive and control when they do not manipulate shared data, no limitations are imposed here upon the argument structures of calls conjoinable by //. Thus to express the problem of deciding whether a given element E belongs to given sets A and B, construct a goal whose concurrent subgoals can be executed independently :

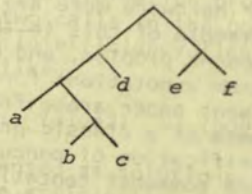
$$\leftarrow m(E,A) \;//\; m(E,B) \;.$$

Similarly, to find an element u common to A and B, construct the goal :

$$\leftarrow m(u,A) \;//\; m(u,B)$$

When either execution instantiates u, the simplest control arrangement (which also usually gives the most sensible behaviour) immediately transmits the binding to the other execution , so that the executions can react instantly to each other's progress. Transmission of bindings complicates the backtracking of concurrent executions. If execution of A in a goal $\leftarrow A//B$ transmits a binding to B and then has to backtrack, repealing that binding, then normally we would also need to interrupt B's execution and backtrack it to the point where it inherited the binding. If realistic exceptions to this rule were found then we would need more elaborate arrangements governing data-flow through shared variables.

We can also permit expression of concurrency within procedure bodies. An example of this is seen in the problem of comparing the frontiers of two binary trees. Derivation of a program for this problem is shown by Hogger (5), in which a tree with frontier (a,b,c,d,e,f) like



is represented by a term $t(t(t(\lambda(a),t(\lambda(b),\lambda(c))),\lambda(d)),t(\lambda(e),\lambda(f)))$. Then a concurrent program which seeks to show that two trees $T1$ and $T2$ represented by such terms have the same frontier is :

$\leftarrow same(T1,T2)$

$same(x,x) \leftarrow$

$same(x,y) \leftarrow split(u',x',x)//split(u',y',y),\ same(x',y')$

$split(u',x',t(\lambda(u'),x')) \leftarrow$

$split(u',x',t(t(x1,x2),x3)) \leftarrow split(u',x',t(x1,t(x2,x3)))$

Observe that the executions which decompose the frontiers of x and y spend most of their time restructuring the trees until they become ready to compare the frontiers' first labels. This restructuring is best performed concurrently as indicated in the second procedure.

3. INFLUENCE OF LOGIC ON CONCURRENCY

In general it is preferable to arrange that as little detail as possible regarding run-time control is programmed into the logic of procedure definitions, for various methodological reasons which are well documented in the logic programming literature. Nevertheless there must exist practical limits upon the complexity of extralogical control annotations. Consider, for instance, how algorithmically intricate such annotations would have to be in order to instruct an interpreter to elicit Dantzig's 'Simplex' algorithm from naive procedures expressing the meaning of linear optimization; they would assume the character of computer programs in their own right and would be more difficult to compose and verify than the procedures they were intended to control. Therefore in practice it is only realistic to expect 'cleverness' in implemented algorithms to derive mainly from the logical content of their procedures. This applies particularly to concurrent algorithms because their efficacy often depends upon carefully contrived communication between the various executions, whilst the control of this communication depends upon subtle, problem-specific logical relationships holding over the evolving data structures which those executions mutually compute and consult. So it is useful to find out what kind of logical constructions can help the programmer to arrange for efficient cooperation between concurrent executions. Some insight into this comes from investigation of a simple problem which is now examined in detail.

4. A CONCURRENT ALGORITHM

An algorithm is required to decide whether a given element E belongs to either of two given sets A and B, and is to achieve this by searching A and B concurrently. One simple way of doing this is to use a program like

$$\leftarrow \overset{\vee}{//} belongs(E,A,B)$$

$$belongs(u,x,y) \leftarrow u \in x$$
$$belongs(u,x,y) \leftarrow u \in y$$

where the new annotation $\overset{\vee}{//}$ labels a call in order to indicate that all procedures responding to it are to be tried concurrently (so that the derived subgoals $\leftarrow E \in A$ and $\leftarrow E \in B$ would be assigned to distinct processors). Successful termination of any one of the ensuing executions would signal successful execution of the annotated call and simultaneously abandon any other unfinished executions instigated by it. Such an arrangement could be called *disjunctive* concurrency, since it is tantamount to solving the goal $\leftarrow (E \in A \vee E \in B)$ by investigating its disjuncts concurrently. However, although such an annotation might have a useful role to play in the concurrent, quasi-breadth-first exploration of alternatives, it brings us no nearer to an understanding of how to arrange for concurrent executions to access and react to detailed knowledge about each other's progress. We shall therefore develop an alternative solution which does.

Instead of using a predicate like *belongs*, which needs to refer to both sets, introduce a predicate $m(u,x,a)$ which deals with just one set x.

The specification of m is as follows :

$$m(u,x,a) \leftrightarrow (u \in x,\ a=YES) \vee (\neg u \in x,\ a=NO)$$

so that its third argument a acts as an explicit 'answer' to the question of whether u belongs to x. Using terms to represent sets, together with an appropriate definition of ε, compose a straightforward procedure set for m :

$$m(u,\Phi\ \ ,a) \leftarrow a=NO$$
$$m(u,v:x,a) \leftarrow u=v,\ a=YES$$
$$m(u,v:x,a) \leftarrow u \neq v,\ m(u,x,a)\ .$$

Now consider an execution $\Gamma 1$, running on a processor P1, of a call $m(E,A,a1)$ where E and A are well-formed input data. Suppose that a concurrent execution $\Gamma 2$, running on a processor P2, is dealing with a call $m(E,B,a2)$ and binds YES to $a2$. If $\Gamma 1$ were somehow 'told' that YES had been bound to $a2$, it could sensibly abandon its attempt to decide $E \varepsilon A$, record an arbitrary answer DONTKNOW for $a1$ and then terminate successfully. This suggests the use of an alternative predicate $m^*(u,x,a1,a2)$ which allows the answer $a1$ computed for the question $u \varepsilon x$ to be contingent upon the state of $a2$. Its specification just extends that for m in order to admit the alternative answer for $a1$:

$$m^*(u,x,a1,a2) \leftrightarrow m(u,x,a1) \vee (a2=YES,\ a1=DONTKNOW)\ .$$

Procedures for m^* then follow by trivial transformation of those for m above :

$$C1: m^*(u,\Phi\ \ ,a1,a2) \leftarrow a1=NO$$
$$C2: m^*(u,v:x,a1,a2) \leftarrow u=v,\ a1=YES$$
$$C3: m^*(u,v:x,a1,a2) \leftarrow u \neq v,\ m^*(u,x,a1,a2)$$
$$C4: m^*(u,x\ \ ,a1,a2) \leftarrow a2=YES,\ a1=DONTKNOW\ .$$

Procedures $C1$-$C4$ can now be used to solve the following goal whose two subgoals are to be dealt with by concurrent processors P1 and P2 respectively :

$$\leftarrow m^*(E,A,a1,a2)\ //\ m^*(E,B,a2,a1)$$

Observe that $C1$-$C4$ compute logically correct answers $a1$ and $a2$ to this goal whether execution of the subgoals is concurrent or sequential. Note also that it is logically impossible to compute $a1 = a2 = DONTKNOW$ but logically possible to compute $a1 = a2 = YES$.

Once again suppose that $\Gamma 2$ confirms $E \varepsilon B$ and assigns YES to $a2$. Meanwhile, $\Gamma 1$ is at the point of evaluating some derived subgoal $m^*(E,A',a1,a2)$ where A' is the subset of A remaining to be searched for E. As soon as $\Gamma 2$ instantiates $a2$ with YES we want $\Gamma 1$ to select $C4$ and terminate rather than select $C1$-$C3$; prior to this, $\Gamma 1$ should have been selecting $C1$-$C3$ in preference to $C4$. These control preferences, which are motivated towards the earliest possible termination, require some modifications to $C1$-$C4$, because they cannot be implemented merely by choosing some other fixed-try ordering or by adding selective data-flow annotation like ? and ^ to their heading arguments.

Therefore insert some checks $a2 \neq YES$ to obtain :

$$C1': m^*(u,\Phi\ \ ,a1,a2) \leftarrow a2 \neq YES,\ a1=NO$$
$$C2': m^*(u,v:x,a1,a2) \leftarrow a2 \neq YES,\ u=v,\ a1=YES$$
$$C3': m^*(u,v:x,a1,a2) \leftarrow a2 \neq YES,\ u \neq v,\ m^*(u,x,a1,a2)$$
$$C4 = C4': m^*(u,x\ \ ,a1,a2) \leftarrow a2=YES,\ a1=DONTKNOW\ .$$

This does not affect (partial) correctness, since *C1'-C4'* are logically implied by *C1-C4*. The checks ensure that when *a2* has been instantiated by *YES*, searching with *C1'-C3'* is discontinued and termination by *C4'* follows immediately. However, these checks must not become operative until *a2* is instantiated, as they would otherwise cause nondeterministic behaviour. To guard against this we can exploit the annotations of Clark and McCabe to specify control alternatives appropriate to the data-flow. Using their scheme, a procedure set for *m** behaving in perfect accordance with our wishes is obtained by forming pairs of annotated control alternatives chosen from both *C1-C4* and *C1'-C4'*, so that each pair acts like a single procedure which can either execute or ignore the call *a2≠YES* :

```
C1 : m*(u,φ  ,a1,a2^) ← a1=NO
C1': m*(u,φ  ,a1,a2?) ← a2≠YES, a1=NO

C2 : m*(u,v:x,a1,a2^) ← u=v, a1=YES
C2': m*(u,v:x,a1,a2?) ← a2≠YES, u=v,a1=YES

C3 : m*(u,v:x,a1,a2^) ← u≠v, m*(u,x,a1,a2)
C3': m*(u,v:x,a1,a2?) ← a2≠YES, u≠v, m*(u,x,a1,a2)

C4': m*(u,x  ,a1,a2?) ← a2=YES, a1=DONTKNOW  .
```

The solutions for the goal variables *(a1, a2)* which are logically computable from these procedures are *(NO,NO)*, *(NO,YES)*, *(YES,YES)*, *(YES,NO)*, *(YES, DONTKNOW)* and *(DONTKNOW, YES)*; but when execution is constrained by the given control annotations, the unwanted answer *(YES,YES)* is not computable.

Observe that it is not necessary for the various sets of control alternatives like {*C1,C1'*} to be logically *equivalent* to a single procedure; it suffices for them to be logically *implied* by a single procedure *(C1)*, so that the annotation scheme allows calls to be selectively skipped as well as resequenced.

Finally, note that an alternative program can be derived by extending rather than modifying *C1-C4* :

```
C1 : m*(u,φ  ,a1,a2^) ← a1=NO
C2 : m*(u,v:x,a1,a2^) ← u=v, a1=YES
C3 : m*(u,v:x,a1,a2^) ← u≠v, m*(u,x,a1,a2)
C4 : m*(u,x  ,a1,a2?) ← a2=YES, a1=DONTKNOW
C5 : m*(u,x  ,a1,a2?) ← a2≠YES, m(u,x,a1)
```

together with the three given procedures for *m*.

This formulation arises by deriving a new *m** procedure

```
m*(u,x,a1,a2) ← m(u,x,a1)
```

inserting a check *a2≠YES* and then annotating *C1-C5* appropriately. It gives much the same behaviour but slightly clearer logic.

5. THE GENERAL PRINCIPLES

The principles underlying the treatment above are as follows:-

(1) Suppose that the problem to be solved has been formulated as a program intended for sequential call execution using a single processor - in other words, as a standard non-concurrent logic program.

Further suppose that the program's goal contains some call *P(t1,...,tr)*. Usually it will be possible to solve the problem more efficiently by simply arranging that one or more other calls are executed concurrently with this call to *P*. However, even greater improvement may be possible when that call is replaced by

$$P^*(t1,\ldots,tr,x_{r+1},\ldots,x_n)$$

where the x_i variables are shared with the other concurrent calls. In effect, this replacement anticipates that useful data from concurrent executions will be transmitted to the x_i variables whilst the call to *P** is being executed, and thereby makes the goal more efficiently solvable than the original one which called *P* instead.

In the example, the goal for the non-concurrent formulation would have been

$$\leftarrow m(E,A,a1),\ m(E,B,a2)$$

This can be solved more efficiently by concurrent processing:

$$\leftarrow m(E,A,a1)\ //\ m(E,B,a2)$$

and more efficiently still by sharing the variables *a1* and *a2* between the calls:

$$\leftarrow m^*(E,A,a1,a2)\ //\ m^*(E,B,a2,a1)$$

So here the transformation process outlined above has been applied to <u>two</u> calls in the goal. In this example the improvement in efficiency lies in the fact that new kinds of answers like

$$(a1,a2) = (DONTKNOW,YES)$$

will be accepted as solutions to the problem, and moreover will be more efficiently computable than *(YES,YES)*.

(2) Compose procedures for *P** capable of solving any activated call to *P** which has only variables in its argument positions *r+1* to *n*. In the example these procedures are *C1-C4*. They are the procedures which, in the final program, will be invoked in response to *P** calls until such time as concurrent executions transmit data through the shared variables.

(3) Now freely insert into these procedures any number of arbitrary calls whose inspection of the arguments in positions $r+1$ to n of the invoking call, communicating data from concurrent executions, improves efficiency. This step inevitably preserves partial correctness. Usually the inserted calls are devised by considering how to obtain optimal behaviour and may not simply arise as a natural result of deriving sufficient P^* procedures to deal with all cases logically admitted by the P^* specification.

In the example the insertions produced $C1'$-$C4'$. They are the procedures which, in the final program, will be invoked in response to P^* calls after data computed by concurrent executions has been transmitted to the shared variables.

(4) Combine the procedures from steps (2) and (3) and adorn them with control annotations - these determine which procedures to invoke in response to P^* calls, according to the instantaneous states (bound or unbound) of the shared variables.

6. ANOTHER EXAMPLE

Owicki and Gries (7) give a detailed account of an algorithm proposed by Rosen (8) which, given an input list L of arbitrary numbers $L(1),\ldots,L(N)$, computes i^* as the least i, if any, satisfying $L(i) > 0$. If no such i exists then i^* is computed as $N+1$. The algorithm pursues two concurrent searches, one inspecting $L(i)$ for even i and the other inspecting $L(i)$ for odd i. The former search computes j as the least even i satisfying $L(i) > 0$ if this exists, but otherwise computes $j = N+1$. The latter search computes k analogously for the odd-indexed members. Finally i^* is computed as the least of j and k. The chief virtue of the algorithm is that if one execution $\Gamma1$ finds some $L(i) > 0$ then the other execution $\Gamma2$ can respond either by setting its answer to $N+1$ and terminating or else by continuing its own search - depending on whether or not $\Gamma2$ has yet searched through L beyond $L(i)$. The usefulness of this consultation between the searches derives from properties of the ordering relation over list indices. It is therefore problem-specific, and this is reflected in the logical structure of the following formulation :

```
←find*(L,N,2,j,k) // find*(L,N,1,k,j), least(j,k,i*)

 find*(x,n,i,j,k^) ← i≤n, x(i)≤0, find*(x,n,i+2,j,k)
 find*(x,n,i,j,k?) ← i≤n, i<k, x(i)≤0, find*(x,n,i+2,j,k)

 find*(x,n,i,j,k^) ← i≤n, x(i)>0, j=i
 find*(x,n,i,j,k?) ← i≤n, i<k, x(i)>0, j=i

 find*(x,n,i,j,k?) ← i≥k, j=n+1

 find*(x,n,i,j,k ) ← i>n, j=n+1   .
```

These procedures can be derived quite easily according to the principles already enunciated, beginning with a simple program in which the two searches proceed sequentially and independently:

```
←find(L,N,2,j) , find(L,N,1,k), least(j,k,i*)

 find(x,n,i,j) ← i≤n, x(i)≤0, find(x,n,i+2,j)
 find(x,n,i,j) ← i≤n, x(i)>0, j=i
 find(x,n,i,j) ← i>n, j=n+1   .
```

Here *find(x,n,i,j)* holds when j is the least $\hat{i} \in \{i,i+2,\ldots,n\}$ satisfying $x(\hat{i}) > 0$, if any; otherwise $j = n+1$. Then *find* is elaborated to *find** and *find** procedures are composed satisfying the specification

find(x,n,i,j,k)* ↔ *find(x,n,i,j)* ∨ (*i≥k, j=n+1*) .

Efficient control of these procedures, governed by the data-flow through k, is then imposed using the annotations shown.

7. DISCUSSION

The treatment illustrated here is consistent with the usual incremental approach to logic program development. Disregarding concurrent capability, a suitable goal and procedure set are composed which would execute correctly under IC-PROLOG, with no restrictions imposed on the use of the available control mechanisms. By reasoning about the program's behaviour we decide how to obtain a more efficient execution were concurrent processing available. If this improvement requires no communication between concurrent executions then it is obtainable by simply writing selected conjunctions as //. Otherwise, new predicates are defined which introduce shared variables as the vehicle of communication, and the program is reformulated such as to compute solutions consulting those variables' states. Correctness-preserving modifications - typically insertion of calls to manipulate the shared variables - are next applied together with any further necessary control annotations. None of these steps violate the formalism's first-order semantics.

The examples given here indicate that for logic programming the introduction of concurrency does not require departure from our usual way of establishing partial correctness (by deriving procedures from specifications). [However, termination proofs may become more tedious, though no different in principle, through having to examine the constraints imposed by control annotations.] In particular we can easily satisfy the 'non-interference' criterion proposed by Owicki and Gries (7) : namely that if a correctness proof for each concurrent execution, acting in isolation, is not invalidated by the addition of logical assertions about what the others compute, then the composite execution also proceeds correctly. For if, for each subgoal, we construct a separate verification of the logic procedures which it invokes, and if the entire goal is

executed (concurrently or sequentially) such as to comply with its logical interpretation, then the union of these verifications is a verification for the complete goal. This is just a consequence of our being able to specify concurrency in the program text without altering its logical meaning.

There are several other aspects of concurrent logic programming which need to be properly researched. For example, it may be necessary to devise ways of specifying temporal coordination. In our examples neither execution needed to wait for results from the other, but in other cases waiting might be necessary; the existing coroutining facilities may not be able to express all such requirements in sufficient detail. At present we could assume a simple default strategy - that if, in response to a call, all logically responding procedures were blocked by data-flow restrictions, then the call would just suspend (rather than, as at present, register a control error) until data became available from other executions. A possible effect of such arrangements is deadlock - the relevance of this to logic programming makes an interesting research topic.

Also, logic programs can accommodate at least two kinds of concurrency (conjunctive and disjunctive) having differing logical associations (nondeterministic call activation and nondeterministic procedure invocation). Some descriptions of other formalisms do not seem to provide clear relationships between their notions of concurrency and nondeterminism, and it may be that clarification could be obtained by comparison with the various nondeterministic features of logic.

Finally, there are other ways in which executions can communicate besides through shared variables. An interesting alternative is the use of global assertions regarded as data structures, supported by mechanisms for enabling concurrent executions to update and interrogate them communally. For both modes of communication it will be useful to discover their practical limitations in order to understand better why the originators of other formalisms, such as Hoare's CSP (4) and Brinch Hansen's Concurrent PASCAL (1) have considered it necessary to rely upon intermediate devices like specialized input-output regimes and monitor processes in preference to global data sharing.

REFERENCES

1. Brinch Hansen, P. The programming language Concurrent PASCAL. IEEE Trans. on Software Engineering, Vol. SE-1, No.2, June 1975.
2. Bruynooghe, M. , Clark, K. L. Parallel programming in predicate logic. Draft Report, Applied Mathematics and Programming Division, Katholieke Universitat, Leuven, Belgium, 1979.
3. Clark, K. L. , McCabe, F. G. The control facilities of IC-PROLOG. Research Report, Dept. of Computing & Control, Imperial College, 1979.
4. Hoare, C. A. R. Cooperating sequential processes. Comm. ACM, 21, No. 8, 1978.
5. Hogger, C. J. Derivation of logic programs. PhD Thesis, University of London, 1979.
6. Kowalski, R. A. Logic for problem solving. Elsevier North Holland, New York, 1979.
7. Owicki, S. , Gries, D. An axiomatic proof technique for parallel programs. Acta Informatica, 6, 1976.
8. Rosen, B. K. Correctness of parallel programs: the Church-Rosser approach. T.J.Watson Research Centre, Yorktown Heights (N.Y.), IBM Research Report RC5107, 1974.

A DATAFLOW INTERPRETER FOR LOGIC PROGRAMS

Paul H. Morris

Department of Information and Computer Science
University of California, Irvine

INTRODUCTION

Dataflow models of computation have been studied by a number of authors [1,2]. In this paper we introduce a high level version of the model. In addition to being concise, our model has a number of interesting features. First, activity counts are not used in our system; instead, certain laws are proposed to restrict interactions among tokens. Second, "indeterminate" computations are permitted; these may be regarded as arising from backtracking. Third, there is a sense in which inputs and outputs are interchangeable, allowing the system to run "backwards." Finally, recursive computations can be represented by a finite network, without an external stacking mechanism.

In a later section of the paper, we consider the Logic Programs of Kowalski [3], and show how they may be reduced to dataflow networks of the type considered here. This is hardly surprising, since data routing is essentially complementary to control in algorithms. Symbolically, A = DF + CF, where A stands for Algorithm, DF for Data Flow, and CF for Control Flow. Contrasting this with Kowalski's equation A = L + C suggests the conclusion Logic = Dataflow. The model presented here is an attempt to reconcile previous dataflow models with the logic view of programming.

The networks we propose here bear a structural resemblance to the flowgraphs of Milne and Milner [4], although their motivation and methods are different. In particular, the laws of balance and minimality do not appear in their system.

EXAMPLE AND INFORMAL DEFINITIONS

We will present the system in an informal manner using an illustrative example.

Figure 1 shows a network which can compute addition for non-negative integers. There are two types of nodes in the network, denoted by circles and boxes in figure 1. The nodes denoted by

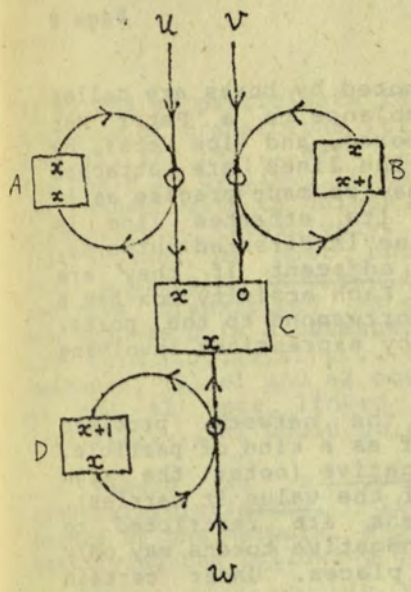


figure 1: Computes $u+v=w$

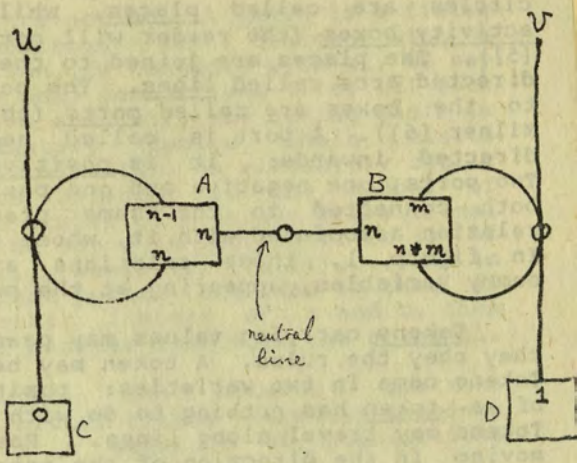


figure 2: Computes $v = factorial(u)$

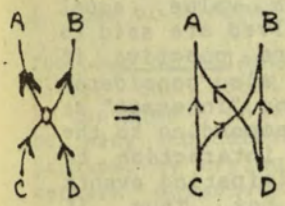


figure 2a: "PLACE," as a hictorial device

plus (X, 0, X).

plus (X, Y+1, Z+1) ← plus (X, Y, Z).

← plus (u, v, w).

figure 3: LOGIC PROGRAM AND GOAL STATEMENT FOR INTEGER ADDITION

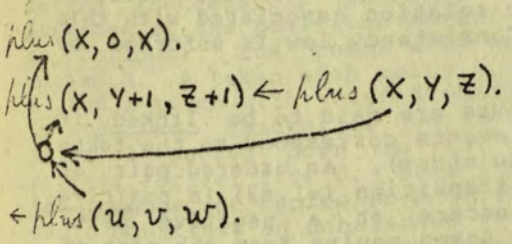


figure 4: MODIFIED CONNECTION GRAPH

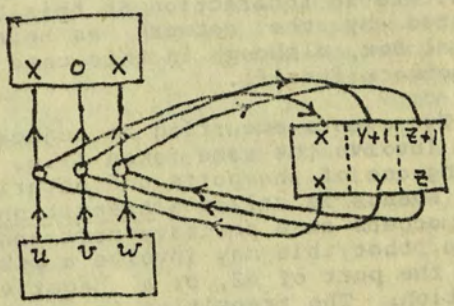


figure 5: DATAFLOW NETWORK

circles are called places, while those denoted by boxes are called activity boxes (the reader will note the resemblance to a Petri Net [5]). The places are joined to the activity boxes, and vice versa, by directed arcs called lines. The points where the lines are attached to the boxes are called ports (this notion may be made precise as in Milner [6]). A port is called negative if its attached line is directed inwards; it is positive if the line is directed outwards. Two ports, one negative and one positive, are adjacent if they are both connected to the same place node. Each activity box has a relation associated with it, whose arguments correspond to the ports. In figure 1, these relations are denoted by expressions involving dummy variables, appearing at the ports.

Tokens carrying values may pass through the network, provided they obey the rules. A token may be thought of as a kind of particle. Tokens come in two varieties: positive and negative (note: the sign of a token has nothing to do with the sign of the value it carries). Tokens may travel along lines. Positive tokens are restricted to moving in the direction of the arrows, while negative tokens may only travel against the arrows. Tokens may wait at places. Under certain circumstances, an interaction may occur at an activity box. This consists of single tokens entering along one or more (possibly all) of the lines, while new tokens, of the correct sign, are emitted along the remaining lines. The tokens that enter are absorbed, and disappear from the network. The act of each token leaving or entering is called an event. Each event is considered to have a value, equal to the value carried by the token. The events involved are said to participate in the interaction. We say two events are coactive if they participate in the same interaction. An event is also considered to be coactive with itself. We may sometimes say tokens "interact" at a box; this means that the events (at that box) corresponding to the tokens are coactive. One of the conditions for an interaction to occur is the Consistency Law: the values of the participating events must satisfy the relation associated with the activity box. Thus, if a token carrying value p enters along a line marked "x" while a token carrying q leaves along a line marked "x+1," then q=p+1.

By convention, the entrances and exits of the network (marked u, v and w in figure 1) are regarded as being attached to a special activity box. The events consisting of the input token(s) entering, and the output token(s) leaving the network are considered to constitute an interaction at this box (we may think of the relation computed by the network as being the relation associated with this special box, although in this case the Consistency Law is enforced by the network itself).

Two events occurring at adjacent ports are said to be linked if they involve the same token (i.e. the events correspond to the token leaving one of the ports and entering the other). An ordered pair of such events is called a transition. A transition (e1,e2) is positive if e1 occurs at a positive port and e2 occurs at a negative port. Notice that this may involve a positive token moving from the port of e1 to the port of e2, or a negative token moving in the opposite direction. The transition is negative if e1 is at a negative port and e2 is at a positive one. Two transitions (e1,e2) and (e3,e4) connect

if e2 and e3 participate in the same interaction. A chain is a finite sequence of transitions {t1,...,tN} such that t sub i connects with t sub i+1, for i=1,...N-1. If, in addition, tN connects with t1, we say the chain is cyclic. If these are the only connections (i.e. tI does not connect with tK unless K=I+1, or I=N and K=1), the cyclic chain is elementary. Intuitively, an elementary cyclic chain visits interactions at most once. We will also call an elementary cyclic chain a circuit. A chain is said to be balanced if the number of positive transitions is equal to the number of negative transitions. Now suppose C is a balanced circuit. We say C is shorted if it is of the form {...,(a,e1),...,tK,(e2,b),...}, where (1) {(a,e1),...,tK} is balanced, (2) e1 and e2 occur at adjacent ports. The idea is that if e1 and e2 were linked to each other instead of a and b, then {(e2,e1),...,tK} would be a circuit of shorter size than the original.

Viewed as a computing device, the network operates nondeterministically. Given some initial tokens placed at boundary points of the network, a computation is a sequence of interactions satisfying the following conditions:

1. At the end of the computation there are no tokens left in the network.

2. Law of Balance: every cyclic chain is balanced.

3. Law of Minimality: no circuit is shorted.

The Laws of Balance and Minimality are unusual in that they are global rather than local conditions on what constitutes a valid computation. At the time a potential interaction is being "considered," it may not be easy to determine whether it is compatible with the Laws (except, perhaps, by trying it and backtracking if necessary). We will not concern ourselves here with how to enforce the Laws in an actual implementation. Our primary interest is in their use as a theoretical device to characterize computations. The meaning of the "no shorted-circuit" condition will become clearer later. Roughly speaking, it ensures that events are linked in such a way as to create the shortest circuits. The motivation for this is to keep values belonging to distinct procedure calls separate.

Consider figure 1 in the light of these definitions. Suppose tokens (necessarily positive) with values 3 and 2, respectively, are inserted at the positions marked "u" and "v." One valid computation is as follows: the token with value 2 is absorbed at the activity box marked B. A token with value 1 emerges. This is in turn absorbed at B and a token with value 0 emerges. This travels down to C, where it is absorbed. Meanwhile, two interactions occur at A and a token with value 3 passes down to C, where it interacts with the token that arrived earlier from B. Note that the Law of Balance requires that the number of cycles at A be equal to the number at B (otherwise, we could construct an unbalanced cyclic chain that starts with the token at "u," follows the cycles through A, then through C, back through the cycles at B and ending at "v").

Now a negative token with value 3 emerges at the bottom of C. The two interactions required by the Law of Balance now take place at D. Finally, a token carrying 5 emerges from the network at the point marked "w." At this point there are no tokens left in the network. The reader may satisfy himself that a valid computation must result in a token with value 5 emerging at "w."

A feature of the model is that the networks are multi-purpose. Figure 1, for example, could be used to compute integer subtraction by inserting initial tokens at the points marked "v" and "w." In this case, the answer token emerges at "u." If we insert a token at "w" alone, the network will nondeterministically compute a pair (u,v) such that u+v=w (we use u, v and w to represent the values on the tokens entering or emerging at the points marked "u," "v" and "w," respectively). That is, for each such pair there is a valid computation that outputs that pair, and every valid computation outputs such a pair. This leads us to speak of the network running "backwards." Another possible mode is to insert tokens at "u," "v" and "w." In this case, a valid computation will simply result in an annihilation of the input tokens. This can occur if and only if u+v=w.

It turns out to be necessary to adopt some restrictions on allowable network configurations. It is convenient to require that the networks be proper in the sense of Milner [6]. In particular, a place node may not be connected to more than one port of the same type (i.e. positive or negative) in the same activity box. Notice that Milner's second condition is a consequence of joining boxes by means of intermediate place nodes.

NEUTRAL LINES

Now that the reader has digested the above, we introduce a third type of token. These are neutral tokens and they travel on neutral lines, which are distinguished in our diagrams by not containing an arrow, i.e. they are undirected. The ports at opposite ends of a neutral line are considered to be adjacent, and are called neutral ports. A neutral transition is one involving a neutral token. Chains may contain neutral transitions. These are ignored in determining whether the chain is balanced.

We will also relax the constraint that at least one token must enter an activity box for an interaction to occur. That is, a box may spontaneously emit tokens along all of its lines. It might be thought that this would result in many more possible computations in figure 1. However, in a valid computation there is a sense in which a positive token leaving C is equivalent to a negative token with the same value entering C. In a manner reminiscent of Richard Feynman's view of antiparticles, a negative token may be thought of as a positive token going backwards in time. We may apply this idea to successively reverse the flow of time on various arcs. By a sequence of such "transpositions," it is possible to reduce the new computations to old ones which involve the same values (although generated in a different

order).

Figure 2 shows an example of the use of neutral lines. This network can be used to compute the factorial function. Suppose a token with value 3 is inserted at "u." One computation proceeds as follows: three interactions occur at A, until a token with value 0 is produced. This travels down to be absorbed at C. Meanwhile, tokens pile up on the neutral line. Next, a negative token with value 1 is spontaneously emitted from D. Three interactions now occur at B, using up the tokens on the neutral line. Notice that for the Law of Balance to be satisfied, the tokens on the neutral line must be absorbed in reverse order. Finally, a token with value 6 emerges at "v." This sequence of interactions simulates a recursive computation of factorial. The same network can simulate an iterative computation. In this sequence, the initial token waits at the first place it encounters. boxes C and D spontaneously emit tokens. Two interactions each at A and B are interleaved. A third interaction occurs at A, with the waiting initial token joining in. A final interaction occurs at B and the answer token emerges at "v."

It is worth noting that the necessity for neutral lines disappears if we are willing to allow non-simultaneous events to participate in an interaction (recall that this is already true for I/O). With this approach we could combine A and B in figure 2 into a single box. Places could also be dispensed with, since it would no longer be necessary for tokens to wait until their "mates" are ready before entering an activity box (however, it is convenient to retain "places" as a pictorial device to reduce the criss-crossing of arcs -- see figure 2a). We will adopt this viewpoint in a subsequent section.

LOGIC PROGRAMS

We now consider the logic programs of Kowalski [3,8]. The reader familiar with these will already have noticed a number of similarities: the multi-purpose character of both systems and their lack of commitment to top-down or bottom-up processing. As we shall see, there is a sense in which dataflow networks, as defined here, are simply a syntactic variant of logic programs. We believe the dataflow interpretation will promote significant advances in the understanding of such programs and of computation in general.

The reduction to a dataflow network proceeds as follows. First write the logic program as a set of clauses, including a general goal clause. We will restrict our attention to Horn clauses. Figure 3 shows an example. The connection graph, introduced by Kowalski [7], links literals among the conditions of a clause with matching literals in the conclusion of the same or other clauses (two literals match if they have the same predicate name). Thus, in figure 3 the literals "plus(U,V,W)" and "plus(X,Y,Z)" are both connected to each of the literals "plus(X,0,X)" and "plus(X,Y+1,Z+1)." We modify this definition slightly: for each predicate name, establish a special node called a predicate node. Now replace each arc in the original connection graph by two arcs, one going from the condition literal to the appropriate predicate node and the other going from the predicate

node to the conclusion literal. We also direct the arcs in the order from condition to conclusion. The result of these modifications is illustrated in figure 4.

By our orientation of the arcs, we do not mean to imply a commitment to top-down processing, but merely to indicate the direction from condition to conclusion. This information would otherwise be lost in our manipulations below.

We are now ready to define a dataflow network. For each clause, draw a box. Write each argument expression, in each literal, at some position on the perimeter of the box. Break each predicate node into several nodes, equal in number to the "arity" of the predicate. Each original arc is likewise split into several arcs, and these are attached to the appropriate argument expressions in the boxes. This is illustrated in figure 5 (ignore the dotted lines in the right-hand box, for the moment). This ends the canonical transformation to a dataflow network. (We can improve its appearance by a means we will not attempt to justify here. "Cut" the right-hand box along the dotted lines, to obtain three boxes. The variables in the right-most two may now be renamed. With minor cosmetic changes, this gives us the network shown in figure 1.)

Given our restrictions on network configuration, we cannot apply this transformation to clauses where the same predicate symbol occurs twice in the body, e.g. xfib(N,2X+Y) <- xfib(N-1,X),xfib(N-2,Y) (in this example, some such restriction is necessary to avoid a possible confusion of tokens). This may first be transformed to the pair:

```
xfib(N,2X+Y) <- xfib(N-1,X),xfib1(N-2,Y).
xfib1(N,Z) <- xfib(N,Z).
```

The transformation may then be completed as above.

It should be clear that this procedure, at least up to the cutting operation, can be applied to every logic program. Assuming that the resulting network computes the same relation as the original program - to be proved below - it follows that every partial recursive function can be computed by some such network, i.e. the dataflow networks constitute a universal programming language.

An interesting property of the networks, as opposed to logic programs, is that they are potentially name free. Naming all the predicates is a tedious aspect of constructing logic programs. Predicates play the same role in logic programs as procedures do in conventional languages, but there are many more of them. In standard languages, programs are broken into procedures to (1) isolate conceptually important segments of the program, (2) allow segments to be reused and (3) support recursive calls on segments. The first use is defeated, to some extent, by the dilution caused by (2) and (3). In our system, the third use is unnecessary. The benefits of the first two uses can be obtained by using single boxes to represent what are actually subnetworks. Box B in figure 2 is actually an example of this. The approach might be referred to as "promoting the primitives."

CORRECTNESS 155

In this section we show the dataflow interpreter is correct in the sense that the network obtained (without cutting) from a logic program computes the same relation as the original program.

Before proceeding, we require some additional concepts. Suppose a and b are events in a computation. Let C = {<e,f>,...,<g,h>} be a chain such that e is coactive with a and h is coactive with b. Let s be the algebraic count of transitions in C, i.e. the number of positive transitions minus the number of negative transitions. Let C1 be another such chain joining a and b, with count s1. Then s = s1; otherwise we could create an unbalanced cyclic chain by splicing together C and C1. Thus the value of s is independent of the particular chain used to join a and b. We call s the separation from a to b, abbreviated as sep(a,b). The following can be easily derived:

1. sep(a,b) = sep(a,c) + sep(c,b), for any event c.
2. sep(b,a) = -sep(a,b).
3. If a and b are coactive, then sep(a,b) = Ø.

Now let eØ be any input/output (I/O) event and let e be any other event. We define the depth of e by:

depth(e) = sep(eØ,e)

Clearly, the value of depth is independent of the particular eØ chosen. From 1 and 2, it follows that sep(a,b) = depth(b) - depth(a).

Next we prove the following lemma for networks arising from the canonical transformation.

LEMMA: Let A and B be two positive ports in an activity box. Let C and D be negative ports of a neighboring box such that A is adjacent to C and B is adjacent to D (see figure 6). Suppose a,b,c and d are events occuring at A,B,C and D, respectively, such that a is coactive with b, c is coactive with d, and a is linked to c. Then b is linked to d.

PROOF: Notice that there may be other, competing, ports adjacent to the ones of interest, as indicated in figure 6. We will call two ports in an activity box "partners" if each is connected to one of the place nodes shown in figure 6. Thus A and B are partners, as are C and D. We will also say two events are partners if (1) their ports are partners, and (2) they are coactive. In this sense a and b are partners, and so are c and d.

Now consider the sequence {e1,e2,e3,e4,...} where e1=b, e2=a, e3=c, e4=d and, for all i,

e sub 2i is the partner of e sub 2i-1
e sub 2i+1 is linked to e sub 2i.

Since the number of events in the entire computation is finite,

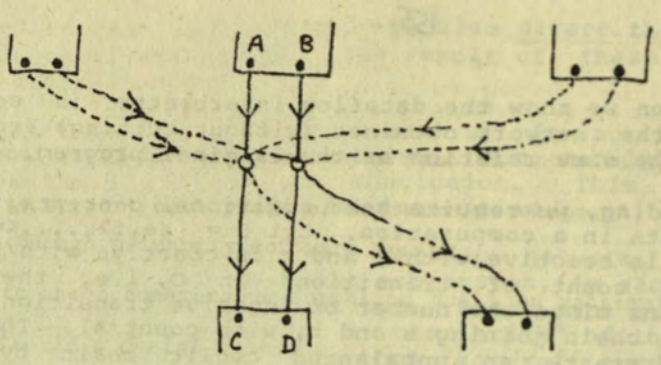


figure 6

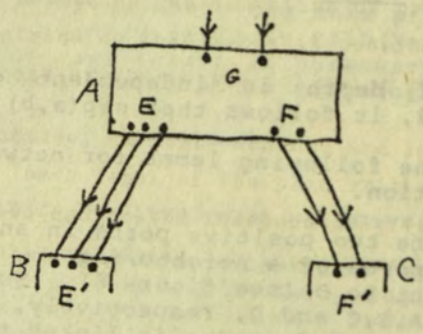


figure 7

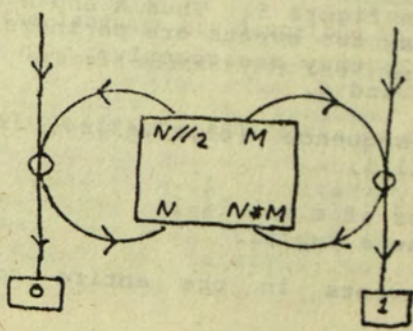


figure 8: NETWORK FOR QFACT

eventually an element in the sequence will recur. Let eN be the first recurring element. Claim: eN=el. Suppose contrariwise that eN=eI where 1 < I < N. Since an event has a unique partner and also a unique "linkee," it follows that e sub I-1 = e sub N-1. That is, e sub N-1 is a recurrent element. This contradicts the fact that eN is the first such element, establishing the claim. N may now be seen to be odd; otherwise e sub N-1 is the partner of el, i.e. a. In fact, N must equal 5. If N > 5 then the sequence is of the form {el,e2,e3,e4,p,...,q,eN}. We may rearrange the events in the sequence to form a cyclic chain {...,<q,el>,<e2,e3>,<e4,p>,...}. Since each event in the chain interacts only with its unique partner (among the events of the chain), the chain is elementary. However, the chain is shorted since {<q,el>,<e2,e3>} is balanced, and el occurs at B while e4 occurs at the adjacent port D. Thus, the assumption N > 5 is incompatible with a valid computation. It follows that N = 5. Hence d is linked to b. This completes the proof of the lemma.

It is now straightforward to prove correctness. We note that the negative ports of a box correspond to the arguments of the head literal of the clause from which the box is derived. We will show that a set of interacting events at these ports must satisfy the predicate of the literal.

PROOF: By downward induction on the depth of the events. Consider events of maximum depth first. These must occur at a box with no positive ports (otherwise there would be events of greater depth). Such a box corresponds to a clause with empty body, i.e an assertion. In this case, the relation associated with the box is such that events satisfying it satisfy by definition the predicate of the head literal. Therefore, the Consistency Law ensures the result for events of maximum depth. Now consider a set at depth n, where n is less than the maximum depth. Assume the result is true for all depths greater than n. If the current box has no positive ports, the result is true as before. Otherwise there are events at the positive ports that are coactive with the set we are considering. These are linked to sets of events of depth n+1 at other boxes. By induction, these satisfy the head literal predicates for their boxes. It follows that the events at the positive ports in the original box satisfy the predicates of the body of the clause. Utilizing the Consistency Law, the events at the negative ports satisfy, by definition, the predicate of the head literal. This completes the induction.

The foregoing step of the proof is illustrated in figure 7. Events are denoted by heavy dots. The events at the region marked E are coactive with those at G and F, and are linked to those at E'. The events at F are linked to those at F'. Recall that box A as a whole corresponds to a clause of the logic program. The ports at G correspond to the arguments of the head literal (i.e. the conclusion). The body of the clause is assumed to have two literals whose arguments correspond to the ports at E and F, respectively. Assuming the events at G are of depth n, those at E' will be of depth n+1. By the inductive hypothesis, the events at E' satisfy the predicate of the head literal of box B. This predicate is the same as that of the literal associated with the ports at E. Since linked events have the same value, the events at E also satisfy this

predicate. Similarly, the events at F satisfy the predicate of the literal there. It follows that the events at G satisfy the predicate of the head literal of box A.

By the preceeding, the I/O events must satisfy the predicate of the clause corresponding to the I/O box, i.e. the goal clause. Thus, the dataflow interpreter is correct. We will not prove completeness here, but it is not hard to see that logic computations which result in fully instantiated variables can be mirrored by dataflow computations of the type described.

DETERMINATE COMPUTATIONS

We hope the machinery developed will prove useful in reformulating algorithms for greater efficiency. For this purpose it is useful to adopt the position mentioned earlier, that events participating in an interaction may be dispersed in time. In this case, it is reasonable to regard a computation as a sequence of events which are related through a structure of links and interactions, rather than as a sequence of interactions. We may have two computations which are identical in structure and values, the only difference being that the events occur in a different order. The recursive and iterative versions of factorial considered earlier are a example of this. Is one such ordering to be preferred to another? We shall see that, from an implementation point of view, the backtracking required for one may be considerably less than that for another. This provides a powerful incentive for choosing the best ordering.

Suppose we are somewhere in the middle of a computation. At this point there will be several potential transitions to choose from. If the wrong one is chosen we may later have to backtrack. Suppose, however, that one of the waiting tokens has only one place it can go (perhaps because of the Consistency Law). If there is any valid continuation from this point, it will have to include that transition sometime. Since the validity of the computation is independent of the order of the transitions, we may as well make that transition now. We are guaranteed it will not have to be undone. We call a transition with this property a <u>determinate</u> transition. A computation in which every transition is determinate is called a determinate computation. It is important not to confuse "determinate" with "deterministic." There may in fact be more than one determinate transition available at a given point, so that the computation is still non-deterministic. In the factorial example considered earlier both the iterative and recursive computations are determinate in our sense.

As a further example, consider a function QFACT defined by:

```
QFACT(Ø)=1
QFACT(N)=N*QFACT(N//2)
```

where // denotes integer division. This can be computed by the dataflow network in figure 8, which resembles figure 2. Here there is also an iterative and a recursive computation. In this case, however, given a value for N//2, there are two possible values for N. Thus the

iterative computation is indeterminate. On the other hand, N//2 is uniquely determined by N, so the recursive computation is still determinate.

It should be pointed out that not every network admits a determinate computation. Some elaboration of this concept would be required for more general applicability.

FUTURE DIRECTIONS

The formulation of the dataflow interpreter developed here may not be the ideal one. It may be worthwhile to experiment with different sets of constraints. There are many open questions concerning allowable "box operations." The general problem of choosing transition sequences to minimize backtracking needs to be vigorously attacked. It would also be desirable to map out the relationship with other models of computation.

The author is of the opinion that many of the earlier models of computation have outlived their usefulness. The need now is to develop models that correspond more closely to the internal mental structures that practising programmers use so flexibly. It is hoped the present work is a small step in that direction.

References

[1] Arvind, Gostelow K. P. and Wil Plouffe, An Asynchronous Programming Language and Computing Machine, Tech. Rep. 114A, Dept. of Info. and Comp. Sci., U. of Cal., Irvine (Dec 1978).

[2] Dennis J. B., First Version of a Data Flow Procedure Language, MAC TM 61, Project MAC, MIT (May 1975).

[3] Kowalski R. A., Algorithm = Logic + Control, Comm. ACM 22, 7 (July 1979), 424-436.

[4] Milne G. and R. Milner, Concurrent processes and their syntax, J. ACM 26, 2 (Apr 1979), 302-321.

[5] Holt A. W., Introduction to occurrence systems, in Associative information techniques (Jacks E. L., ed.), American Elsevier, New York, 1971.

[6] Milner R., Flowgraphs and Flow Algebras, J. ACM 26, 4 (Oct 1979), 794-818.

[7] Kowalski R. A., A proof procedure using connection graphs, J. ACM 22, 4 (Oct 1974), 572-595.

[8] Kowalski R. A., Logic for Problem Solving, North Holland - Elsevier, 1979.

LDM - a Logic Based Software Development Method

P.Szeredi *
K.Balogh **
E.Sántáné-Tóth *
Zs.Farkas *

(This research was supported by State Office of Technical Development (OMFB)).

Abstract

LDM is a software development method based on the ideas of logic programming (PROLOG) and the Vienna Development Method (VDM). The paper gives an overview of the main features of the language of LDM, making also a comparison with VDM and PROLOG. An illustrative example of the development of a simple assembler is presented, finally the experiences in using the method and further plans are summarised.

1. Introduction

This paper gives an account of a Logic-based Development Method (LDM) that has been worked out in two Hungarian institutes for software development, SZKI and NIM IGÜSZI. LDM is intended to give more than its name expresses: it covers not only a method for the development of software-objects, but also a language to be used during development, and a system supporting it. The report primarily deals with the language, since it constitutes the formal basis of the system and reflects fundamental features of the method.

---

* Institute of Co-ordination of Computer Techniques
SZKI, HUNGARY 1368 Budapest, Pf. 224.

** Institute of Industrial Economy and Plant Organisation
of the Ministry of Heavy Industries
NIM IGÜSZI, HUNGARY 1363 Budapest,Pf. 33.

LDM came into being as a combination of two existing languages (respectively methods): the PROLOG language [6], [7] based on logic and the VDM (Vienna Development Method) [2]. Besides these LDM was influenced by the languages IOTA [5], CLEAR [3] and CIP [1], also. The starting point was the PROLOG language; its successful applications resulted in the idea of trying to apply it for formulating plans of software objects. The first experiments in this direction were very promising. On the first hand PROLOG, as a sublanguage of first order logic proved to be suitable for specifying and design software-objects. On the other hand execution mechanism of PROLOG helped considerably in immediate checking, testing of the specification during preparation. However, PROLOG is lacking in tools for adequate handling of the plethora of data structures. Exactly this possiblity was found later in the domain notion of VDM. Thus LDM adopted from PROLOG the logic-based definition mechanism (and so inherited constructivity, that is executability of abstract descriptions), and from VDM the notion of domains (that is data types) and their handling.

With the help of domains and operations in LDM software objects can be modelled on the whole range of abstraction levels. Exploiting this possibility, LDM supports a multi-level development process: an implementation can be reached from the specification through several intermediate design levels having a lower and lower degree of abstraction. Each level gives the whole description of the object in question. Each level is a reformulation of the previous one with some abstract notions replaced by more concrete ones. (This is opposed to the other usual interpretation of the phrase "abstraction level" when the lower level is meant to supply the definition of the notions left undefined in the previous level.) The idea of a multi-level development process is also present in VDM [2], however LDM aims at supporting this process by language and system constructs, as well.

Objectives of LDM are partly the same as those of VDM:

- to support each phase of software development (though primarily software design),
- to use a homogeneous formalism built on mathematical bases,
- to aid a design method applying several abstraction levels,
- to give abstract, descriptive, applicative language elements (types of objects, operations) for specification, design phases, and concrete, algorithmic, imperative sorts for implementation phases.

LDM differs essentially from VDM in that it is intended to design software only, but not to define semantics of programming languages. It is this restriction that makes it possible, for the goals of LDM to exceed those of VDM in the following respects:

- LDM has a constructive, strictly defined and implementable language; therefore each level of the plan written in that language can be executed and tested independently (from execution of other levels), while verifying the interfaces between neighbouring levels;

- language structures reflecting the process of the software design are introduced, both in order to divide plans into levels and to decompose levels functionally.

Basic notions of the language are described in the following. Then a fairly complex example illustrates the introduced notions.

A detailed description of the language can be found in [10].

2. Outline of the features of the LDM language

The LDM language is used for describing different levels of abstractions which model software objects. Accordingly it provides abstract notions corresponding to notions of data and algorithms used in traditional programming languages.

2.1. Domains

Data appear in LDM as suitable mathematical objects (e.g. numbers, sets, functions). The class of objects used for the same purpose, having the same structure constitute a domain.

The notion of domain corresponds to that of a data type, so the expression "is from domain t" is equivalent to "is of type t".

There are simple and composite domains in LDM. A simple domain contains some kind of unstructured simple objects (e.g. numbers). A composite domain (e.g. that of pairs of numbers) contains objects having the same structure and it can be constructed from one or more domains by systematical application of some object constructing operation.

Beside simple and composite domains so-called derived domains can also be defined in LDM. Definition of a derived domain does not introduce new objects, it creates a new domain from objects of existing domains (e.g. the domain of even numbers).

One can use so-called domain expressions for identifying domains in the text of a plan description (e.g. the domain of pairs of numbers is denoted by "struct (number, number)"). With the help of domain definitions one can denote the used domains by identifiers (e.g. "pairs-of-numbers ::= struct(number,number).").

The simple domains of LDM are the following:

number - the domain of integers;

the a - the domain containing a single object named by "a", "a" is any name determined by the designer ;

The composite domains are the following (let "t", "t_1", ..., "t_n" denote any domain, "a" any name):

list t - the domain of finite lists (sequences) of objects from "t";

set t - the domain of finite sets of objects from "t";

struct (t_1, ..., t_n) - the domain of structures (trees), i-th components of which are objects from "t_i";

map (t_1, t_2) - the domain of finite mappings, which map a finite number of objects from "t_1" to objects from "t_2";

t named a - the domain, containing those objects, which are constructed from those of "t" by the operation of naming (tagging) by "a".

Finally the definition of the derived domains is the following ("p" denotes any one-place predicate symbol):

t_1; t_2 - the domain containing all the objects from both "t_1" and "t_2";

t suchthat p - the domain of those objects from "t", for which the predicate "p" holds.

The use of these constructs is illustrated by the following domain definition:

source-program::=(list source-statement) suchthat wf-program.

This means, that a "source-program" is a list of objects of type "source-statement", for which the property "wf-program" holds.

The language contains standard functions and relations for the different kinds of domains.

For instance to an object SP of type "s-prog" the standard list-operations can be applied: among others the function "length SP" gives the length of the list SP, the relation "S elem SP" decides, whether S occurs in SP or not, etc.

The notion of domain in LDM is almost entirely equivalent to that of VDM. The only essential difference between them is that LDM does not allow general (infinite) functions as objects (only finite map objects are allowed). The reason of this restriction is mainly the intention to have a constructive language, and it is justified by the restricted programme of the language (namely LDM is not for describing semantics of languages). A less significant change is, that the operation of composing structures and that of naming (distinction) is separated: VDM structures (non-anonimous trees) correspond to objects of form "struct (...) named ... " in LDM. Finally the notational differences between the two languages are given by the reason, that LDM aspires to be implementable using a restricted character-set.

2.2. Operations

The notion of operation in LDM corresponds to the algorithmic components of other languages. Operations can be described by operation definitions. "Operation" is a comprehensive name for relations, functions and procedures. Among them the notion of relation is the primary, and the two others are reduced to it.

A relation definition is actually a logical formula of resticted form (the character of the restrictions is determined by the requirement of reducibility to Horn clauses - i.e. the possibility of implementation in PROLOG).

The notion of function in LDM is a syntactic one; the function form is interpreted in the usual way as a relation (having an additional argument). In this approach the many-valued (non-deterministic) functions are also included in a natural way.

Relations and functions can be used in the descriptive definitions, in the phase of specification and early phases of design. On those levels of the plan approaching implementation it is necessary to write algorithmic definitions as well.

The so-called procedure definitions serve this purpose. These are also logical formulas, which are considered as algorithms on the basis of the procedural interpretation of (Horn-) logic. It is interesting to remark, that this procedural interpretation in the deterministic case gives just the control structures of the language CDL (designed for the implementation of software systems, [4]), and this language CDL is one of the basic implementation languages joining the LDM design in prospective applications. In the case of procedures - as an additional syntactical extension - the use of global, "changable" variables is also allowed. A relation changing a variable is interpreted as a relation having two further argument positions according to the initial and the final value of the variable. In the procedure definitions the usual language constructs for handling variables (assigments to them and reading their values) are allowed.

In general, the basic building blocks of the operation definitions are the already mentioned, standard operations, from which the logical formulas are constructed by the usual logical connectives. For the sake of comfortable notation operation schemes are introduced to denote the complicated structures (for example conditional, case structure and bounded quantification).

Besides the operation definitions LDM descriptions contain some important additional elements. Every operation definition must be accompanied by a type specification, which fixes the type of the arguments of the operation. The already mentioned global variables can be introduced by variable definitions. Finally, there are additional constructs which can be used for structuring the plan; see the following section.

The notion of the operation in LDM - as delineated above - is based on logic, or more exactly on PROLOG. This approach - besides assuring the executablity of the plan (plan variants), makes the plan itself more concise and simple. Here primarily the following well-known feature of logic (relational) programming is referred to [6]: the same relation definition can be used for computing different functions, by changing the input-output role of the arguments. As a simple consequence of this feature the analogue of construction "let mk-D(...)= ..." of VDM (which breaks down a structure into its components) is assured in LDM automatically.

2.3. The structure of LDM plans - language elements provided for structuring

As already mentioned in the introduction, the purpose of LDM is to support a multi-level design process going from specification towards implementation. Accordingly, an LDM-plan is divided into levels: a plan description is actually a sequence of level descriptions. The main components of level descriptions are the domain and operation definitions introduced in the previous two sections. These are framed by the so-called define- and need-specifications. The notions, with which the level provides the external world, are listed in the define-specification. These are the main operations and domains of the software object to be realized, which are defined by the level. In the need-specification we give the notions used but not defined in the level description, that is, we give our expectations about the external world.

The levels are the decriptions of different abstraction levels of the same software object. Therefore it is very important to make clear the connections between the levels; the so-called interface-specifications serve this purpose.

The main function of the interface-specifications is to fix the correspondence between the domains and operations of two consecutive levels (for the domains using a relation and for the operations by a logical formula).

Beside structuring according to the abstraction levels we also feel that the modularization of level descriptions is necessary, that is, their decomposition into independent parts is required. In LDM the notion of the group serves this purpose. The groups are framed - similarly to the levels - by define- and need--specifications. Outside of the group only those of its operations and domains are visible, which are specified in this frame; that is, information hiding can be realized with the aid of groups. The groups also can be divided into so-called plan-parts; a plan-part (or briefly a part) consists of definitions related from a certain point of view, but without the above mentioned information hiding property. The parts are useful for example when we want to adopt all definitions of a part to the next level without change; in this case on the new level it is enough to refer to the name of the part (using the same keyword).

3. Application of LDM to a simple example

After the short survey of LDM given in the previous section the LDM plan of a simple assembler is examined. The figure shows the structure of the first three levels of the plan.

The first level gives the specification of the assembler (accordingly the name of the level is : "specification"). The group named "programs" says that a well-formed source program is a well-formed list of statements, and a target program is a list of target statements; also it is stated that the result of

translating a well-formed source program is the list of the target statements resulting from translating the source statements in the appropriate order.

The group named "statements" of the level gives the structure of the source and target statements, and defines - using an appropriate address calculating function - the meaning of the translation of a statement (with respect to the source program as an environment). The LDM-program of this level can be found after the figure. (To ease understanding, the more complicated elements are commented with an English language description of the meaning of that element.)

Reading the program it can be seen that though this first level of the plan determines the result of the translation, and it can be executed (the result of the translation of a source program is built up from the translated source statements), this execution is hardly effective. For it can be seen that the address calculating function at every applied occurence of a label repeatedly searches for its single ("uniquely-defined") defining occurence in the source program.

For the sake of a more effective address calculation on the second level in the group "dictionary" the auxiliary notion of the dictionary is introduced, and the meaning of dictionary manipulation is expressed, but only declaratively. Also the group "statement" speaks declaratively about the translation of a statement. The structure of the program and the statements is the same as on the first level; the respective parts are only referred (same) here.

The third level describes a possible way of algorithmization of the translation using a dictionary introduced on the second level, namely, the plan of a two-pass translation. In this plan we give a more algorithmic construction both of the source program, and of the dictionary, replacing the implicit list and

map creating operations by appropriate loop statements; on the other hand, the address calculation is algorithmized by introducing an address-counter.

The example is not typical from the point of view of the general design method in the sense that the source and target programs are modelled by the same domains on every level. However, this example shows, that in this case also it is natural to approach the effectively implementable solution in several steps.

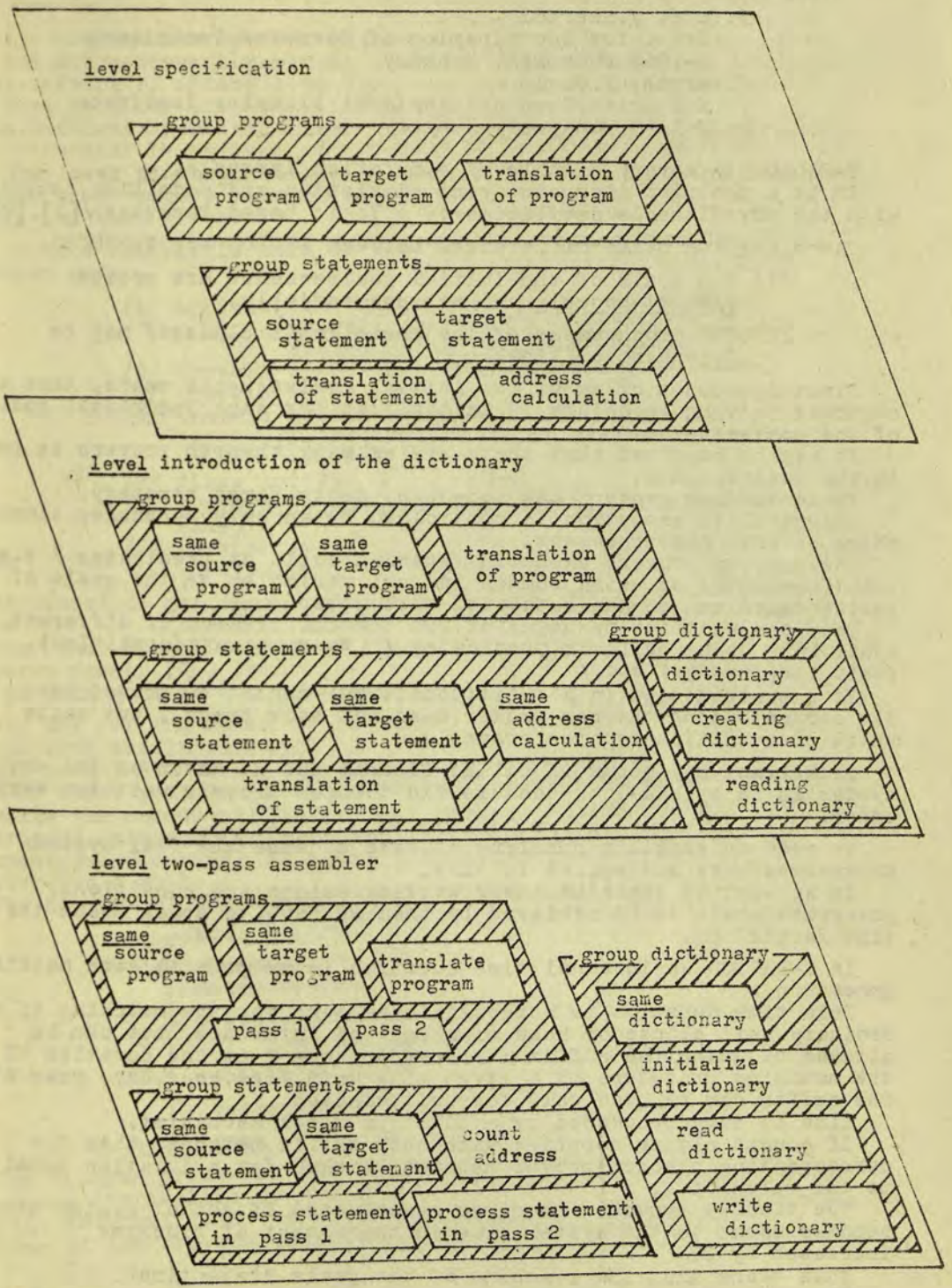


Figure: The structure of the first three levels of the simple assembler

Using T-PROLOG for a long-range regional planning problem

I.Futó, J.Szeredi
Inst. for Coordination of Computer Techniques
H-1368 Budapest, Hungary
E.Barath, P.Szaló
Hungarian Town and Regional Planning Institute
H-1253 Budapest, Hungary

T-PROLOG is a very high level simulation language.
It is a tool for discrete event simulation and modelling, supplied with the advantageous facilities of a logic based language /[1],[2],[3]/.

There are two basic differences between PROLOG and T-PROLOG.

/1/ The goals of the initial goalsequence are proved simoultaneously and not sequentially.

/2/ The truth value of the facts /unit clauses/ may be dependent on time.

Simoultaneous proving of the initial goalsequence means, that a separate proving procedure is initialized for each individual goal of the goalsequence.
It may be supposed that there are as many theorem provers as goals in the goalsequence.
These theorem provers use top-down, depth-first strategy.
The parallel execution is controlled by an only scheduler since there is only one processor.
The theorem provers executing special built in predicates / e.g. :send(message) / or using common logical variables in the goals of the goalsequence can communicate with each other. .
They can wait for messages or for the fulfilement of different conditions using built in predicates./ e. g. :wait(condition), :wait_for(message) etc./
If a precondition in a :wait(condition) cannot be proved yet, the corresponding theorem prover does not back track , but waits until precondition becomes provable.

Back track may occur if called literal has no match or the whole system is in a dead-lock because the theorem provers wait for each other.

We need an explicit handling of time because the real systems themselves work and evolve in time.
If we want to describe their working axioms and conditions, prescribe goals to be achieved by them we have to speak about the time factor too.

In T-PROLOG an internal time duration is assigned to each matching procedure.
This time duration is supposed to be zero for the matching of an ordinary unit clause or head of a rule of inference, but can be altered by using the suffix :during(T) where T is the duration of the matching procedure in a given time unit./sec,hour,day, year etc./ The suffix can be used only for unit clauses.
Time is duly processed by a built in clock mechanism.
If a unit clause supplied with such suffix matches, then the corresponding theorem prover becomes blocked for a duration equal to the value of T.
The theorem prover becomes reactivated if actual system time becomes equal to the system time in the moment of blocking increased by T.
This means that the reaching of the goals "takes time".
The beginning and the end of a goal proving procedure can also be prescribed.

Unit clauses in case of ordinary unit clause are always true, they are not necessarily always true if they are suffixed by :before(T) :at(T),:after(T), :from(T_1 to T_2).

These suffixes are actually control conditions and if any unit clause suffixed by one of these time prescriptions is matched then the system time is checked and if this time condition is not satisfied then back tracking begins.

Back track occurs also if the duration of the proving procedure exceeds the limit given.

A simple example is here to show how the scheduler and clock mechanism work.

```
/1/ arrives(somebody,somewhwere,vehicle,time):
            travels_by(vehicle,somewhere),
            system_time(time).
/2/ travels_by(TRAIN,WIEN ): during(4).
/3/ travels_by(CAR,WIEN ) : during(6).
/4/ travels_by(AIRPLANE,WIEN ):
            flies_to(WIEN ),
            travels_to_the_center_of(WIEN ).
/5/ flies_to(WIEN ): during(1).
/6/ travels_to_the_center_of(WIEN ) : during(1).
    fin.
/7/ :new(arrives(PETER,WIEN ,vehicle,time).nil,PETER )end 3,
     new(output(WIEN -vehicle-time).nil,OUTPUT ) start lo.
```

We wanted to get closer to the formulas used by those working in the field of simulation. For this purpose by process we mean that specific part of the search tree which is traversed by a theorem prover [1]. According to this there is a process corresponding to each goal in the goalsequence.

/1/ and /4/ are rules of inference, /2/,/3/,/5/,/6/ are unit clauses with suffix :during(T).

/7/ is a T-PROLOG goalsequence having two goals.

The first goal is to be proved by the process called PETER, while the second serves for printing out the answer. This second process, named OUTPUT has to start at time unit lo.

:new(goal,procname) is a built in predicate for creating a process with a goal.

The diagram of the execution is the following:

```
                ┌:arrives(PETER,WIEN ,vehicle,time).   system time
                                                       is equal to ∅
/1/             │ somebody:=PETER
                │ somewhere:=WIEN
                ├:travels_by(vehicle,WIEN ),
                │        system_time(time).
/2/             │ vehicle:=TRAIN
                └:system_time(time).
```

back track

The theorem prover of the first goal is to be blocked for 4 time units /"traveling by train"/, but the actual system time + the blocking interval is ∅ + 4 = 4 is greather than 3, the prescribed end time of the proving procedure. The system back tracks.

```
/3/  vehicle:=CAR                                   system time
     :system_time(time)                             is equal to ∅
back track
```

The theorem prover is to be blocked for 6 time units and as 6>3 back tracking begins.

```
/4/  vehicle:=AIRPLANE
     :flies_to(WIEN ),
          travels_to_the_center_of(WIEN ),
               sytem_time(time).
/5/
     :travels_to_the_center_of(WIEN ),
          sytem_time(time).

Process PETER is blocked
for 1 time unit./traveling/
```

As the second process has to start at time moment 1o the next action to perform is the reactivation of PETER at time moment 1.
The system time is set to 1 by the clock mechanism and PETER continues to travel.

```
     :travels_to_the_center_of(WIEN ),          system time
          system_time(time).                    is equal to 1
/6/
Process PETER is blocked
for 1 time unit
```

The system time is icreased by the clock mechanism and is set to 1 + 1 = 2

```
:system_time(time).                             system time
built in predicate                              is equal to 2
time:=2
```

The process PETER succesfully completed its task. The next action to perform is to initialize the process OUTPUT. The system time is set to 1o by the clock mechanism.

```
:output(WIEN -AIRPLANE-2).                      system time
built in predicate                              is equal to 1o
```

The problem is solved, the answer is: PETER travels to WIENA by AIRPLANE and arrives at 2 time units.

T-PROLOG is used first for modelling the possible ways of developement of different regions of Hungary.
The goals in the model reflect the state to be reached in certain time by the counties using different common -shared- resources.
Every county is represented by a process.
The goal of the simulation is to find the appropriate way of developement of the counties according to the goals prescribed.

The system is composed from the hierarchical rules of county categorisation, the data describing the actual state of a county and the data transformation rules requiring time. The latest means that rules are given describing how many time is required to reach a given value of a data type from the actual value of the same data type.

Here we give some examples of the different logical assertions.

```
the_level_of_the_urban_developement_of_an_area_in_the_integrated_
        space_structure(area,DEVELOPED_HIGHER_THAN_THE_AVERAGE)
    :if(
      in_the_functional-territorial_structure(area,INTENSIVELY_
            DEVELOPED)
      or
      in_the_functional_territorial_structure(area,INTENSIVELY_
            DEVELOPED_FROM_THE_POINT_OF_VIEW_OF_THE_PRODUCTION_
            AND_SETTLEMENT_FACILITIES)
      or
      in_the_functional_territorial_structure(area,INTENSIVELY_
            DEVELOPED_FROM_THE_POINT_OF_VIEW_OF_THE_SUPPLY)
    and
      in_the_technical_physical_territorial_structure(area,
            INTENSIVELY_DEVELOPED)
      or
      in_the_technical_physical_territorial_structure(area,
            INTENSIVELY_DEVELOPED_FROM_THE_POINT_OF_VIEW_OF_
            THE_LAND_USE)
      or
      in_the_technical_physical_territorial_structure(area,
            INTENSIVELY DEVELOPED_FROM_THE_POINT_OF_VIEW_OF_
            NETWORKS)).
```

There are some 5o inference rules of this kind in the system yet.

```
the_number_of_industrial_workers_in(AREA,NUMBER).
the_number_of_agrarian_workers_in(AREA,NUMBER).
```

There are some 2o data of this kind about every AREA.

```
the_number_of_industrial_workers_in(AREA,NUMBER2):
    the_number_of_industrial_workers_in(AREA,NUMBER1),
    passes(T).
passes(time):
    hold(time).
```

:hold(time) is a built in predicate. It serves to block a process for a duration equal of the value of time.

The system is under developement and only the first experimental version works yet.

References

[1] I.Futó, J.Szeredi, K. Szenes
T-PROLOG a very high level simulation language
To appear in CLACL North Holland

[2] R. Kowalski
Logic for problem solving
Univ. of Edinburgh Memo no.75 1974.

[3] D.H.D. Warren, L.M. Pereira
PROLOG: The Language and Its Implementation Compared With LISP
SIGPLAN Notices Vol. 12 no.8 aug. 1977

PROLOG applications in Hungary

E. Sántáné-Tóth, P. Szeredi

Institute of Co-ordination of Computer Techniques /SZKI/
HUNGARY 1368 Budapest Pf 224

Abstract

The paper makes an overview of the main PROLOG applications in Hungary. For each application a short description of the problem and the main characteristics of the implementation are given. Finally the paper summarizes the experiences of the described applications.

1. Introduction

Since 1975 /the implementation of the first PROLOG interpreter in Hungary/ many problems have been solved using PROLOG; problems previously either unsolvable /in traditional programming languages/ or solvable only by applying complex algorithms and considerable effort.

A group in NIM IGÜSZI began PROLOG development and application programming. The first applications already showed how clearly and simply programs could be written in PROLOG solving problems needing the facilities of symbol manipulation, pattern matching, serching and deduction.

A version of the PROLOG interpreter written in CDL1 was installed in 1977 on a SIEMENS 7.740 /and later on a 7.755/ computer of SZKI, under the operating system BS2000. This installation served as a basis for later installations on computers compatible with the IBM system 360. Meanwhile this interpreter was extended with the facility of interactivity - among other possibilities.

In 1978 KSH OSZI* provided support for research work aiming at a search for new application areas for PROLOG and aid for solving new problems in PROLOG. The PROLOG applications in Hungary were studied in this framework [Sántáné-Tóth,79] and the results of this research served as a primary source of this survey.

In the following we begin with an enumeration of PROLOG installations in Hungary and then we give a review of the more interesting PROLOG applications. Finally we shall summarize the experimental results and give some comments on the basis of current PROLOG applications. The bibliography comprises the papers, research reports and internal memos published by Hungarian authors to date.

We should like here to thank everybody working in the development and application of PROLOG for their help; everybody mentioned in this material, and for the contributions of those who helped in putting together this paper.

2. PROLOG installations in Hungary

The table below shows the existing Hungarian PROLOG installations in time order. Due to the fact that the interpreters are written in CDL there were no special difficulties when transferring to other machines. /The reference and users' manuals are [Szeredi P,77a], [Köves,78] and [Laufer,79]/.

---

* State Institute for Application of Computer Technique of the Central Statistical Office

Year	Institute	Machine	Op. System
1975	NIM IGÜSZI	ICL 1903A	GEORGE-2
1975	OTSZK	ICL SYSTEM 4/70	MULTI JOB
1976	EVIG	EMG 840	tmin /+mix/
1976	ASZSZ	HWB 66/20	GCOS
1976	ELTE	ODRA 1304	GEORGE 1-2
1977	MÜM SZÁMTI	ICL 1905	GEORGE 1-2
1977	Kőbányai Gyógyszerárugyár	ODRA 1305	GEORGE 1-2
1977	EGYT	ODRA 1305	GEORGE 1-2
1977	SZKI	SIEMENS 7.755	BS2000
1977	SZÁMOK	IBM 370/145	DOS/VS
1978	SZÁMKI	R22	OS MFT
1978	SZÁMOK	IBM 370/145	OS/VS1
1979	ÉTI-ÉGSZI	SIEMENS 4004	BS2000
1979	PMMF	R22	DOS
1979	KFKI	R40	OS MVT
1980	SZTAKI	IBM 3031	CMS

Table: Hungarian PROLOG installations till early 1980

It is worth noting here that an improved PROLOG interpreter, the so-called MPROLOG is now being developed /see [Bendl,78,79b,80] , [Köves,79]/. One of its essential new features is that it facilitates modular programmi in PROLOG and offers better programming aid and execution mechanism.

3. PROLOG applications in Hungary

PROLOG has been used for several purposes so far. The anthology "How to solv it with PROLOG" * well represents the fields where special problems could successfully be solved using the language. Grouped according to topics, here follows a list of successful Hungarian PROLOG applications. For each of them the year of realization, the computer on which the program was developed or adapted, and the relevant publications are given immediately after the title.

3.1 Applications in the pharmaceutical research

3.1.1 PROGRAM-SYSTEM TO CALCULATE PARAMETERS PREDICTING BIOLOGICAL ACTIVITY OF PEPTIDES

1979; SIEMENS 7.755 ; [Darvas,79c].

Peptides compose a compound group of increasing importance in pharmaceutical research. For predicting their activity a model revealing the specific features of peptides is needed.

The mixed-language PROLOG-FORTRAN program-system "generates" a family of structure-activity models and, at the same time facilities prediction of the biological activity on the basis of the models. The system infers chemical structural units /substructures, fragments/ from the aminoacid /i.e. chemical/ composition of peptides and assigns numerical parameters to the units. This is done by PROLOG programs. The relationships between the parameters and the biological activities of peptides is investigated by FORTRAN programs.

---

* H. Coelho, J.C.Cotta, L.M. Pereira: How to solve it with PROLOG? LNEC, Lisboa, 1979.

3.1.2 PROGRAM-SYSTEM SUPPORTING RESEARCH MANAGEMENT IN THE PHARMACEUTICAL INDUSTRY

1979; SIEMENS 7.755 ; -

The program-system helps to solve research management problems in the pharmaceutical industry by information retrieval and automatic inference; the latter aiming at finding new applications for drugs and pesticides. The languages of the system are PROLOG and EDT /the editor of BS2000/.

3.1.3 A PROGRAM-SYSTEM TO CALCULATE PHYSICO-CHEMICAL PARAMETERS FOR DRUG DESIGN PURPOSES

1976-78; ICL 1903A , ODRA , SIEMENS 7.755 ; [Darvas,78a], [Darvas,78d].

In computer-aided drug design, a considerable part of the calculations is based on the so-called logP value of components, a value indicating their lipophillic character. The manual calculation of this value is time-consuming and results are of questionable accuracy. In 1976, when the PROLOG program was written, only one computer program had been published for this calculation.

3.1.4 PROGRAM-SYSTEM FOR PREDICTING DRUG-INTERACTIONS

1975-79; ICL 1903A , ODRA , SIEMENS 7.755 ; [Darvas,76], [Darvas,78c], [Darvas,79b], [Futó,78a].

Modification in clinical effects may arise when drugs are parallelly administered.The so-called drug-interactions constitute an aspect of medical treatment which is not negligble. The present system considers the physico-chemical, pharmacological and chemical properties of drugs and, starting from these, infers the possible drug-interactions.

3.1.5 NUMERICAL ANALYSIS OF LIGAND-BONDING SYSTEMS

1979; SIEMENS 7.755 ; [Kőfalusi,79c].

The PROLOG program generates a FORTRAN SUBROUTINE segment which calculates the proper initial value for the numerical analysis of ligand-bonding systems. The program is based on the program-generator described in 3.7.2.

3.1.6 TESTING AND MODELLING OF SELF-REPRODUCING BIOCHEMICAL PROCESSES

1979; ICL SYSTEM 4/70 , SIEMENS 7.755 ; -

The program permits the analysis and modelling of any biochemcal self-reproducing cycle. With the replacement of the built-in date-base any cycle can be examined. The input data of the program are the formal reaction equations; in the course of processing selected are the nutriments, end-products and attractors. The program is an appropriate example for the fast andconvenient definition of structural system-models in PROLOG; these advantages come from the use of logical expressions treatable by PROLOG.

3.1.7 SEARCH OF ANALOGOUS SUBSTRUCTURES TO ENZYME-SEQUENCES

1979; SIEMENS 7.755 ; [Mátrai,79].

The program serves as a means to find the structures of enzymes with known sequences and similar mechanism, that are presumably relevant from the viewpoint of functioning. The program is suitable for the search of analogous primary sequence units of any size and having any number of error-spots.

3.2 Information-retrieval systems

3.2.1 A LOGIC-BASED CHEMICAL INFORMATIONAL SYSTEM

1976-78; ICL 1903A , ODRA , SIEMENS 7.755 ; [Darvas,78b, 79a].

The program-system consists of statistical programs in FORTRAN and PROLOG programs for automatic inference based on chemical structure and chemical and biological properties as well. PROLOG allows the storage of graphs in the form of clauses. This allows making additions to the programs in the form of logical statements that are connected to the structures and substructures /as conditions/, and refer to the chemical and biological properties of the compounds containing the relevant structures.

3.2.2 AN INTERACTIVE INFORMATION SYSTEM FOR AIR POLLUTION CONTROL

1977; ICL SYSTEM 4/70 ; [Bendl,79a], [Futó,78a].

This program-system handles data about the basis concentration of seven industrial pollutants in Budapest and in the counties of Hungary with each county having 15-20 districts. The system checks whether the air-pollution of working or planned plants is below the permitted level. If not, it calculates the height of the chimney necessary to reduce the concentration, moreover it looks up in its data-base and recommends industrial filtering equipment appropriate to the given industrial branch technology. The system works interactively; the basic motive of its planning was to enable people with different approaches /e.g. managers, designers and research workers/ to use the system.

3.2.3 INFORMATION RETRIEVAL SYSTEM PROCESSING DATA ON PESTS AND PESTICIDES

1977; ICL SYSTEM 4/70 ; [Futó,78a].

This PROLOG program, that was aimed at the examination of results to be expected when applying different pesticides in given situations, can be used to determine interactions among the three following factors:
- diseases, pests, etc. detrimental to a culture;
- insecticides, pesticides, etc. against given diseases;
- cultures, application areas of given insecticides, pesticides, etc.

3.3 Applications in the building industry

3.3.1 PLANNING OF A ONE-LEVEL WORKSHOP BUILDING USING PREFABRICATED PANELS

1975; ICL 1903A ; -

This program is the first PROLOG application in Hungary; it plans a one-level workshop made from prefabricated panels.
The ground-plan is a rectangle. The building is to be constructed from columns, beams and ceiling panels. The data of the available prefabricated elements /geometric size, net weight, supporting strength/ are given in the form of assertions of the program. As initial data the geometric size parameters of the workshop and the intensity of the balanced load of the ceiling are given. The program determines the groundplan /the distribution of the ceiling panels/ and chooses the elements appropriate from the view-point of the geometric and statical conditions.

3.3.2 ARCHITECTURAL PLANNING OF PANEL BUILDINGS

1976; ICL 1903A , R22 ; [Márkusz,77a, 77b].

The program generates the ground plan variants of flats with given size, number of rooms or halfrooms, using the panel elements given in the data base.

3.3.3 PLANNING BUILDINGS WITH MORE THAN ONE LEVEL

1980; SIEMENS 7.755 , IBM 3031 ; [Márkusz,80a, 80b].

The program system provides support in the stages of planning of buildings with more than one level. First it generates the variants of groundplans of all flats according to the special requirements of the customers. The customer is given the possibility to classify the variants, to choose the most advantageous ones, and to exclude the less favourable ones. The program assorts from chosen variants the plan of the whole building satisfying requirements for the horizontal and vertical arrangements; the given measurements, and the conditions depending on the building site.

3.3.4 AUTOMATIZED SOLITAIRE FOUNDATION PLANNING

1979; ICL SYSTEM 4/70 , SIEMENS 4004 ; [Holnapy,79].

The problem solved by the program is the selection of bodies, usable under columns, from a given fixed set of system components /defined by assertions/. An extended version is experimentally tested now, where an arbitrary system of loads /load list/ and a distance list can be given in the goal statement and the result consists of the identifiers of the foundation bodies to be used at the loading forces.
Note: a PROLOG program is planning for the simulation of the results of an architectural selection based technical planning process.

3.4 Software applications

3.4.1 PROGRAM GENERATOR OF COBOL PROGRAMS FOR INPUT CHECKING

1978; SIEMENS 7.755 , ICL 1903A ; [Láng,78].

The program generator is written in PROLOG and is applicable for generating ANSI COBOL programs to be used for checking input data. The generated program outputs the valid data on an output file and print the erroneous data /indicating the cause of the error/. The structure of the files maintained by the generated COBOL program and the aspects of the validation are defined with parameters coming either from terminal or from a file.

3.4.2 GENERATING COBOL PROGRAMS ACCORDING TO COLAMI STANDARDS

1979; SIEMENS 7.755 ; [Futó,79d].

The program generators written in PROLOG generate programs according to the COLAMI standards developed in SZKI. The generators create programs for solving data processing problems of the following type: listing data files, data maintenance, merging two data files and validation of primary input data. The structure of the input/output data maintained by the generated COBOL programs and the work to be done can be determined by uniform parameters given either from a terminal or from a file.

3.4.3 PROM-MAPPING OF INTEL 3000 MICROPROGRAMS

1978; SIEMENS 7.755 ; [Szeredi J,78] , [APPL,78].

INTEL 3000 microprocessors have a special program addressing mechanism. The program store can be thought of as a matrix with every instruction in the matrix pointing to its successor/s/. There is only a limited number of matrix nodes where a successor can be placed /e.g. in the same column as the predecessor/, furthermore the form of limitation depends on the sort of the instruction.
The PROLOG program performs the task of address assignment. Its input consists of the partially loaded store and microprogram. The PROLOG program gives either a possible mapping plan as input, or indicates the impossibility of mapping the given microprogram - in a reasonable amount of CPU time.

3.4.4 ANALYSIS OF PROGRAMMING STYLE AND EFFECTIVENESS

1980; IBM 370/145 ; [Gerő,80].

The aim of the program system is the evaluation of quality of syntactically correct PL/1 and COBOL programs by their analysis according to structural, style, effectivity and complexity aspects. In addition to the recovery of the quality errors in the programs, the program system suggests alternatives for the correction. During the structural analysis the system reveals and prints /in the form of hierarchy diagrams/ the logical construction of the program in question, and notes the structural corrections to be executed. The program system was implemented mostly in PROLOG, the module drawing the hierarchy diagrams is written in optimizing PL/1.

3.4.5 A SYSTEM FOR VERIFYING PROLOG PROGRAMS

1977-78; ICL SYSTEM 4/70 , SIEMENS 7.755 ; [Balogh,77].

The system aims at proving partial correctness of PROLOG programs. A subsystem, consisting of a program for formula transformation and a program for general theorem proving is usable, however, by itself, for interactive theorem proving. The interactive formula transformation program performs natural deduction on the basis of either built-in or interactively generated transformational /inference/ schemas. The general theorem prover program is based on the resolution principle.
This is an experimental system. The studies [Balogh,78] and [Balogh,79a] deal with the conceptual side of the system, too.

3.4.6 PLANNING SOFTWARE AND HARDWARE OBJECTS

1978; ICL 1903A , SIEMENS 7.755 ; [APPL,78].

The experiences with PROLOG applications revealed the suitability of the language for solving problems manageable only with difficulties or not at all in other languages. The programs written in PROLOG reflect very clearly the structure of the problem and the way of solving it.
This - and the facility of validation runs during PROLOG coding - gave the idea that the application of PROLOG as a language for planning might be a worth-while experiment. The plans of sorting programs, a file maintenance system, a module library maintenance program and an RPG parser was developed.

The experimental applications enumerated above proved that PROLOG is usable for planning software /and hardware/ objects but the language lacks the data handling facilities, "real" means for interactive testing and an appropriate methodology of planning. Having drawn the conclusion a logic-based language for software development /the LDM/ was designed, see [Szeredi P, 80].

3.5 Supporting computer architecture design

3.5.1 ETALON-PROGRAM GENERATOR FOR THE EVALUATION OF HIGH-LEVEL LANGUAGE ARCHITECTURES

1978; SIEMENS 7.755 ; [Kiss V, 78].

In design and comparative evaluation of high-level language architectures for the analysis of effectiveness such programs are needed that have average statistical features for given high-level language /e.g. rate of occurence of instruction types or of data types/. A PROLOG program was developed as an experimental tool, for the purpose of generating etalon programs. The program input consists of the syntax rules of the given language and the statistical features we want the generated programs to have.

3.5.2 A SIMULATOR FOR EVALUATING THE DESIGN AND THE EXPERIMENTAL TESTING OF HIGH-LEVEL ARCHITECTURES / DELBOLSIM /

1979; SIEMENS 7.755 ; [Kiss V, 79].

The basic purpose for the development of DELBOLSIM was to provide computerized support for the design process of language-oriented computer architecture. The system is applicable for:
- measuring quantitative factors characterizing the effectiveness of the given architecture,
- the experimental validation of the specified architecture by the means of running test programs, and finally,
- measuring the dynamic statistics of the use of the source language.

The objective of the development of DELBOLSIM was to support architectures complying the Canonic Interpretation Form of programming languages.

3.6 Simulation

3.6.1 A VERY HIGH-LEVEL LANGUAGE SUITABLE FOR THE SOLUTION OF PROBLEMS INVOLVING PARALLELISM

1978; SIEMENS 7.755 ; [Futó,80a].

The interpreter of the language is capable of running an arbitrary number of PROLOG-like goals in parallel. The processes executing the goals can communicate with each other through logical variables, the data-base and a simple demon mechanism. In the case of a deadlock further paths are tried through backtracking.

3.6.2 AN INTERPRETER FOR THE LANGUAGE T-PROLOG

1980; SIEMENS 7.755 ; [Futó,80b, 80c].

T-PROLOG is an extension of the language cited in 3.6.1 with a capability for explicitly and implicitly handling time. This makes T-PROLOG into a full-fledged simulation language. There is now a project underway to use T-PROLOG in the examination of the long-term regional models of VÁTI /an institute concerned with urban development/.

3.6.3 GENERATING MODELS OF TELEPROCESSING NETWORKS

1980; SIEMENS 7.755 ; -

A PROLOG program was developed for supporting the generation of simulation models of remote data processing networks.
First phase: the user enters graphically the network topology to a display screen.

Second phase: the system enquires the network elements to be placed into the nodes, the line algorithm for the edges, and the type of the channels transmitting data.
Third phase: comparing the given data to a data base the system checks whether the given hardware elements were permitted in the given nodes.
Fourth phase: if the check gave positive result then the system enquires the data flow protocols. The protocols give the information concerning the distribution of the data quality among the terminals and the central processor /remote data processing network with one processor/.
Fifth phase: on the basis of the given data the system generates a simulation model of GPSS /SIAS/ language, which can be executed immediately.

The model watches the throughput of the system, and waiting queues at the nodes, etc.

3.7 Other applications

3.7.1 COMPUTERIZED MORPHOLOGICAL ANALYSYS OF HUNGARIAN TEXTS

1979; SIEMENS 7.755 ; [Kiss Z, 79].

The problem of parsing Hungarian texts by computer hasn't been solved yet - because of the complexity of the grammar. Due to the different verb forms, order relations, the assimilation of vocals and consonants it is very difficult to write an algorithm solving this problem.
The first step of text parsing is the morphological analysis of the text. This part of the problem could be solved in PROLOG relatively easily. The program itself actually represents two automata - one for the morphological analysis of the Hungarian verb forms and the other for the morphological analysis of the tagged nouns.

3.7.2 GENERATING THE FIRST N FORMAL DERIVATIVES OF GIVEN VERY COMPLEX REAL FUNCTIONS OF MORE THAN ONE VARIABLE

1979; SIEMENS 7.755 ; [Kőfalusi,79b].

The PROLOG program solving the problem generates a FORTRAN subroutine; during generation phase every variable of the manipulated formulae is factored to the left wherever it is possible. The generated FORTRAN subroutine computes the substitution values of the differential quotients for the variable values characterized by the given parameters.
The PROLOG program is based on a state space concept to be represented by a graph constructed by the author, and used at a first time in this program. This concept of state space might become a new element of PROLOG /see [Kőfalusi,80]/.

3.7.3 SIMPLIFICATION IN MATHEMATICAL STRUCTURES

1979-80; SIEMENS 7.755 ; [Kőfalusi,79a].

The objective of the program under development is the simplification of expressions allowed in a very broad class of mathematical structures /including groups, rings, fields, Boolean lattices etc./. The program treats the expressions represented by binary trees bottom-up, from left to right, and looks in the forward direction for one level, backward for more levels. In the case of an associative chain of operators the program performs sorting according to the appropriate ordering aspects, makes reductions and simplifications according to the operational rules, and then creates prefix structures of n arguments finally further simplifies these structures.

4. The experiences and conclusions derived so far from the PROLOG applications

As it was already mentioned in Section 1. the installation on the SIEMENS 7.755 computer of SZKI can be used in a relatively comfortable way, and provides one of the largest memories for the users because of the virtual store facility. Even this store is not big enough for quite a lot of practical problems. This interpreter works, however, relatively slowly, so the programs need a great amount of time to run /this is partly due to the paging required by the virtual store handling/.

The users who have already solved a lot of problems using traditional languages definitely required the facility of calling subprograms and procedures written in other languages. The objective of the development of the interpreter version [Kőfalusi,79] was to solve this problem.

A number of users required the incorporation of certain procedures into the language, especially ones concerned with floating point arithmetic. The fulfilment of the demand mentioned above would serve as a possibility for solving this problem, too.

The majority of the users who had got used to the traditional languages complained, that having studied the presently available PROLOG reference manuals, they couldn't write by themselves a PROLOG program solving a given problem. They were not able to acquire without help neither the knowledge necessary for the application of the method, nor the proper attitude. Thus the question: "How to teach PROLOG?" arises. This problem is studied for example in [Kőfalusi,79] and [Kaposi,79a].

There is a system called MPROLOG under development which is designed on the basis of the experiences gained from the PROLOG applications until now, and the development conceptions described in [Szeredi P, 79] . See about the subject [Bendl,78, 79, 80] and [Köves,79] . The system is intended to reduce all the problems mentioned above, and even to solve optimally some of them. The store required by the user program will be one fifth of the present one, for example. A new reference manual will appear that is structured in a more didactic way and contains exercises, too.

The applications gave also rise to problems caused by the incompability between the character of the problem, and the /strict/ tree-traversing strategy of the PROLOG interpreter / see e.g. [Kiss Z, 79], and [Kőfalusi,80]/. According to the users the formulation of the problem in PROLOG was even then worth the effort, if, for the sake of efficiency, the program was to be transcribed into a traditional language. It was worthwile, as they couldn't formulate the problem previously in a traditional language in an appropriate way, quickly and clearly. So PROLOG proved to be useful for them as a language for supporting design. Furthermore, the users could advantageously exploit the tools provided by PROLOG for supporting trace and other conveniences, and so check and test the plan itself. However, as the applications described in [APPL,78] also showed, the current PROLOG version is not applicable for design of software /and hardware/ objects, because of it lacks the data description facilities. We note, that a logic- /PROLOG-/based language for supporting design /the LDM/ is under development /see e.g. [Szeredi P, 80] and this language will provide for a solution of this problem.

According to the opinion of the users if somebody has already learned to follow the way of thinking PROLOG requires, the language helped him very much to formulate his problem. Having acquired some practice the programmer could test and correct his program very quickly and easily. We note, that

those services of the above mentioned system MPROLOG supporting the trace and the interactive usage will be even more convenient to use.

The symbol manipulation and list processing facilities of the language are much more sophisticated and feasible than the similar facilities of other, widely used languages, these tools make possible an easier and faster programming than traditional means. PROLOG is an ideal implementation language specially for those problems solvable by means of traversing a tree, because PROLOG supports both the data representation and the description of search algorithm.

Bibliography

Here we give a list of those articles and reports of Hungarian authors which are connected or deal with PROLOG.

[Andréka,76] H. Andréka, I. Németi: The generalized completeness of Horn predicate logic as a programming language. DAI Report, No. 21., Univ. of Edinburgh, 1976.

[APPL,78] The application of the PROLOG language to the design of software and hardware objects. Volume I. - IV. /Hungarian/ NIM IGÜSZI and SZKI reports. SOFTTECH D21, 22, 23 and 27, SZÁMKI, 1978.

[Balogh,75] K.Balogh, K. Lábadi: Software applications of the mathematical logic. /Hungarian/ Proc. of the conference: "Programming Systems'75" , Szeged, 1975. pp.26-44.

[Balogh,77] K. Balogh, I. Futó, K. Lábadi: The documentation of a PROLOG program verification system. NIM IGÜSZI memo for KSH OSZI, 1977.

[Balogh,78] K. Balogh: On an interactive program verifier for PROLOG programs. To appear in Proc. of Coll. on Math. Logic in Programming, Salgótarján, Hungary. North Holland Publ. Comp. 1978.

[Balogh,79a] K. Balogh: On a logical method serving the proof of the semantic features of programs. /Hungarian/ Dissertation, 1979.

[Balogh,79b] K. Balogh, E. Sántáné-Tóth, P. Szeredi: Logic based program design. /Hungarian/ Proc. of First National Conference of NJSZT, Szeged, 1979. pp.36-45.

[Bán,79] P. Bán, J. Kőhegyi, Gy. Suhai, A. Veszprémi, L. Zsakó: Some questions of information system of ANSWER. /Hungarian/ ELTE report. SOFTTECH D38, SZÁMKI, 1979.

[Bendl,78] J. Bendl, K. Varga, M. Kósa, K. Balogh: The specification of an interpreter of a modular PROLOG /Hungarian/ NIM IGÜSZI report. SOFTTECH D20, SZÁMKI, 1978.

[Bendl,79a] J. Bendl, Gy. Lugosi, Z. Márkusz: An interactive system for checking air pollution - an information system. /Hungarian/ Journal Információ Elektronika, XIV. No.1., 1979. pp.55-58.

[Bendl,79b] J. Bendl, J-né Boda, G. Bogdánfy, M. Kósa, L. Naszvadi, J. Visnyovszky: A users' documentation of the system MPROLOG. /Hungarian/ NIM IGÜSZI report for SZÁMKI, 1979.

[Bendl,80] J. Bendl, P. Köves, P. Szeredi: The MPROLOG system /These proc./

[Darvas,76] F. Darvas, I. Futó, P. Szeredi: A program for the automatic filtering of drug interaction. /Hungarian/ Proc. of the coll. "The application of computer technics and cybernetics in the medical sciences and biology"/ed. D. Muszka/, Szeged, 1976. pp.413-422.

[Darvas,78a] F. Darvas, I. Futó, P. Szeredi: Some application of theorem proving based on machine intelligence in QSAR: automatic calculation of molecular properties and automatic interpretation of qualitative structure-activity relationships. Proc. of the Symposium on Chemical Structure-Biological Activity: Quantitative Approaches, Suhl, GDR, Akademie Verlag, Berlin. pp.251-257.

[Darvas,78b] F. Darvas, I. Futó, J. Szeredi, J. Bendl, P. Köves: A PROLOG-based drug design system. /Hungarian/ Proc. ot the conf. "Programming System'78", Szeged, 1978.

[Darvas,78c] F. Darvas, I. Futó, P. Szeredi: A logic-based program system for predicting drug interactions. International Journal of Biomedical Computing, Vol. 9., 1978. pp.259-271.

[Darvas,78d] F. Darvas: Computer analysis of the relationship between the biological effect and the chemical structure. /Hungarian/ Kémiai Közlemények, Vol. 50.,1978. pp.97-116.

[Darvas,79a] F. Darvas, J. Szeredi, I. Futó, J. Rédei: A logic-based chemical information system - theoretical considerations and experiences. /Hungarian/ First National Conference of NJSZT, Szeged, 1979. pp.92-96.

[Darvas,79b] F. Darvas, I. Futó, P. Szeredi: Expected interactions of spirololactions: predictions by computer. Proc. of the Conf. on Pathogenesis of Hyperaldosteronism, ed. E Glaz. 1979. pp.219-220.

[Darvas,79c] F. Darvas, A. Lopata, Gy. Mátrai: A specific QSAR model for peptides. To appear in: "Quantitative structure activity analysis", ed. F. Darvas. Akadémiai Kiadó, Budapest.

[Darvas,80] F. Darvas: Logic programming in chemical information handling and drug design. /These proc./

[Futó,77a] I. Futó, F. Darvas, E. Cholnoky: Practical applications of an AI language /PROLOG/ . Second Hungarian Computer Science . Conference, Budapest, 1977. pp. 388-399.

[Futó,77b] I. Futó, P. Szeredi: AI languages - the PROLOG language. /Hungarian/ Információ Elektronika, XII. No. 2, 3, 1977. pp. 108-113 and 146-152 resp.

[Futó,78a] I. Futó, F. Darvas, P. Szeredi: The application of PROLOG to the development of QA and DBM systems. Logic and Data Bases ed. Gallaire and J. Minker. Plenum Press, New York and London, 1978. pp.347-376.

[Futó,78b] I. Futó, J. Szeredi, J. Rédei: A very high level language supplied with facilities for parallal programming. /Hungarian/ SZKI report, 1978.

[Futó,78c] I. Futó, F. Darvas, P. Szeredi, J. Szeredi, P. Köves: Biodesign, a logic-based system for drug design. To appear in: Proc. of Coll. on Math. Logic in Programming, Salgótarján, 1978. North Holland Publ. Comp.

[Futó,79a] I. Futó, J. Szeredi, J. Rédei: PAPLAN - Users' Reference Manual. /Hungary/ SZKI study, 1979.

[Futó,79b] I. Futó, J. Szeredi, J. Rédei: The very high level language PROPHET and its application. /Hungarian/ First National Conf. of NJSZT,Szeged, 1979. pp.146-157.

[Futó,79c] I. Futó, J. Szeredi, J. Ródei: The very high level language PROPHET. /Hungarian/ SZKI report, 1979.

[Futó,79d] I. Futó et al.: On a program system generating COBOL programs complying with the COLAMI standards. Description for users. /Hungarian/ SZKI report, 1979.

[Futó,80a] I. Futó, J. Szeredi: PAPLAN - an AI language for parallel problem solving. Submitted to: International Conf. on "Artificial Systems of Robots", Smolenice Castle, Czechoslovakia, 1980.

[Futó,80b] I. Futó, J, Szeredi, K. Szenes: A modelling tool based on mathematical logic - T-PROLOG. To appear in: CL & CL, North Holland Publ. Comp.

[Futó,80c] I. Futó, J. Szeredi, E. Baráth, P. Szalo: using T-PROLOG for a long-range regional plannings problem. /These proc./

[Gerő,80] P. Gerő, D-né Halmay: Computer aided supervision as a training of programmers. SZÁMOK report, 1980.

[Holnapy,79] D. Holnapy: On the mathematical foundations of the automatized technical planning - An algebraic structure isomorph to a building of systems. A research report./Hungarian/ ÉTI rep. 197

[Kaposi,79a] A. A. Kaposi, Z. Márkusz: PREMLOG the case for augmented PROLOG programming. Proc. of Informatica'79, Bled, 1979.

[Kaposi,79b] A. A. Kaposi, Z. Márkusz: Introduction of a complexity measure for control of design errors in logic-based CAD programs. Proc. of CAD'80, Brighton, 1980.

[Kiss V,78] V. Kiss, G. Simor: A preliminary specification of an architectur design environment and the analysis of the programming tools applicable in the environment. /Hungarian/ SZKI report for SZTAKI, 1978.

[Kiss V,79] V. Kiss, G. Simor: On a simulator for evaluating the design and the experimental testing of higher level architectures - DELBOLSIM . A desription for designers. /Hungarian/ SZKI report for SZTAKI, 1979.

[Kiss Z,79] Z. Kiss, G Prószéky, L. Tóth: Morphological analysis of Hungaris texts with computer. /Hungarian/ MTA NYTI report. SOFTTECH D41, SZÁMKI, 1979. /An English version to appear in CL & CL ./

[Kőfalusi,79] V. Kőfalusi, F. Bartha: On a possible application and extension of PROLOG: applications based on state space sets. /Hungarian/ SZKI report. SOFTTECH D42, SZÁMKI, 1979.

[Kőfalusi,79a] V. Kőfalusi: On simplification in mathematical structures. /Hungarian/ SOFTTECH D42, SZÁMKI, 1979. pp.12-86.

[Kőfalusi,79b] V. Kőfalusi: A PROLOG program for generating the first n formal derivatives of given, real multi variable analytic functions of great complexity. /Hungarian/ SOFTTECH D42, SZÁMKI, 1979. pp.104-127.

[Kőfalusi,79c] V. Kőfalusi, F. Bartha: Numeric analysis of ligand bonding systems. /Hungarian/ MTA SZBK report. SOFTTECH D42, SZÁMKI, 1979. pp.128-132.

[Kőfalusi,80] The state space set and its applications in PROLOG. /To appear in CL & CL ./

[Köves,78] P. Köves: BS2000 PROLOG users' reference manual. V2.4. /Hungarian/ SZKI report for SZÁMKI, 1978.

[Köves,79] P. Köves: A preliminary users' manual of the debugging and trace subsystem of system MPROLOG. /Hungarian/ SZKI report. SOFTTECH D32, SZÁMKI, 1979.

[Láng,78] O-né Láng: On the generation of data processing ANSI-COBOL programs in PROLOG. /Hungarian/ Proc. of the conf. "Programming Systems'78", Szeged, 1978. pp.364-368.

[Laufer,79] T. Laufer: DOS PROLOG users' reference manual. /Hungarian/ Report for SZÁMKI, Technical Highschool, Pécs, 1979.

[Márkusz,77a] Z. Márkusz: How to design variants of flats using programming language PROLOG, based on mathematical logic. Proc. of IFIP'77, Toronto, 1977.

[Márkusz,77b] Z. Márkusz: The application of the programming language PROLOG for panel house design. /Hungarian/ Információ Elektronika, XII. No. 3. 1977. pp.124-230.

[Márkusz,80a] Z. Márkusz: The application of the programming language PROLOG for house design. /Hungarian/ /Submitted to Információ Elektr./

[Márkusz,80b] Z. Márkusz: An application of PROLOG in designing many storied dwelling houses. /These proceedings/

[Mátrai,79] G. Mártai: The application of PROLOG for search of similar substructures of enzym sequences. /Hungarian/ MTA SZBK rep.1979.

[Mátrai,80a] G. Mátrai, F. Darvas, K. Keleti: Homologous subsequences in dehidrogenases. /Hungarian/ MTA SZBK report, 1980.

[Mátrai,80b] G. Mátrai: Primary structure activity of dehidrogenases. /To appear in J. of Mol. Bio. No. 5., 1980./

[Sántáné-Tóth,79] E. Sántáné-Tóth: PROLOG applications in Hungary in 1979. /Hungarian/ SZKI report. SOFTTECH D42, SZÁMKI, 1979.

[Szeredi J,78] J. Szeredi: On the application of mathematical logic in computer techniques. /Hungarian/ Dissertartion, 1978.

[Szeredi P,75] P. Szeredi: On a high-level programming language based on logic. /Hungarian/ Proc. of the conference: "Programming Systems'75", Szeged, 1975. pp.191-209.

[Szeredi P,77a] P. Szeredi, I. Futó: PROLOG reference manual. /Hungarian/ SZÁMOLÓGÉP, VII., No. 3-4. 1977. pp.5-130.

[Szeredi P,77b] P. Szeredi: PROLOG - A very high level language based on predicate logic. Prep. of Second Hungarian Computer Science Conference, Budapest, 1977. pp.853-856.

[Szeredi P,79] I. Futó, K. Lábadi, P. Szeredi, K. Balogh /ed.: P. Szeredi/: On the implementation methods and theoretical foundations of language PROLOG. /Hungarian/ NIM IGÜSZI and SZKI report. SOFTTECH D34, SZÁMKI, 1979.

[Szeredi P,80] P. Szeredi, K. Balogh, E. Sántáné-Tóth, Z. Farkas: LDM - a logic based development method. /These proc./

[Tóth,74] P. Tóth: Comparison analysis of the very high level programming languages. /Hungarian/ NIM IGÜSZI report for KSH OSZI, 1974.

K.L.Clark & F.C.McCabe

IC-PROLOG: Aspects of its implementation

K.L.Clark & F.G.McCabe

Department of Computing and Control
180 Queens gate
London SW7

Short Paper

In this companion paper to "IC-PROLOG: language features" we will describe some of the implementation problems and solution techniques behind the language features of IC-PROLOG. We examine each feature in turn in a similar order to their presentation in the other paper. We conclude with a brief section on the lessons of the IC-PROLOG implementation.

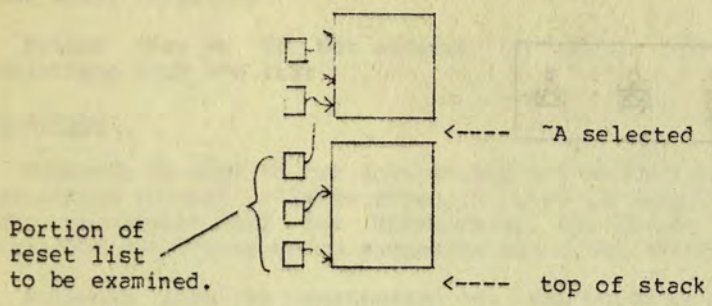
Negation

Negation is usually implemented in PROLOG systems as negation-as-failure, and we use this technique in IC-PROLOG. The major difference between the implementation of negation in IC-PROLOG and other PROLOGs lies in the treatment of variable bindings.

Where a negated atom contains no variables then the normal PROLOG implementation using slash and meta-variables correctly implements the negation-as-failure rule. However if a negated atom contains a variable (say P(x)) we must be careful about the difference between implementing $\neg\exists x\ P(x)$ and $\exists x\ \neg P(x)$. The difference arises in the action taken by the system when the atom in the negated call succeeds and binds a variable in the call. If this is interpreted as failure (the action taken by the normal PROLOG implemention) then we are assuming $\neg\exists x\ P(x)$. I.e. we have shown that $\exists x\ P(x)$ with x=A (say) and then declare $\neg\exists x\ P(x)$ to be false. This is of itself a valid interpretation, but it does not correspond to the normal declarative reading of the clauses as universally quantified implications.

In IC-PROLOG we implement the 'other' form of negation: $\exists x\ \neg P(x)$ does correspond to the reading of clauses as universally quantified implications. If the proof of P(x) succeeds we check to see if x has been bound. If it has, we report a CONTROL error.

Before the IC-PROLOG interpreter reports failure for a negated call ~A, and after it has noted that the atom A in the call has succeeded, a check to see if any variables in A have been bound. This is done by examining the reset lists between the point where we selected the call ~A and the current top of stack:



If there is an entry in one of the reset lists belonging to an activation record at or above the point where the negated call was selected. If one is found then, since the whole of the later (lower) part of the stack belongs to the computation of A, it follows that a variable of A has been bound.

Conditionals

IC-PROLOG allows the user to specify conditionals in the bodies of clauses and in goals. The syntax of conditionals is:

C <- P THEN Q ELSE R

where P must be a single literal, but Q and R can be conjunctions of literals and conditionals. The above clause is logically equivalent to the pair of clauses:

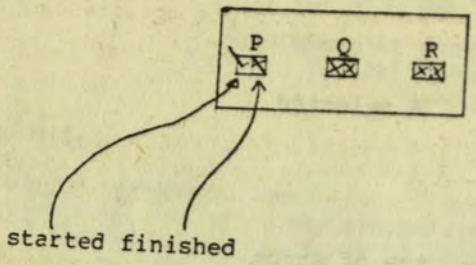
C <- P & Q
C <- ~P & R

IC-PROLOG also takes as control information the fact that the conditional test needs only to be perfomed once.

Conditionals can be programmed up quite readily using the slash feature of PROLOG and the meta variable facility, but in IC-PROLOG a different approach is taken. The method we use relies on a data structure for activation records chosen for other reasons (mainly for efficient implementation of coroutining). In an activation record we keep a record of which of its atoms have been a) started and b) finished. When selecting for the next call to activate instead of following an explicit 'success' pointer, the flags are examined and the next unfinished call is taken up. Suppose we had a goal of the form

<- P THEN Q ELSE R,

then the goal activation record would look like:



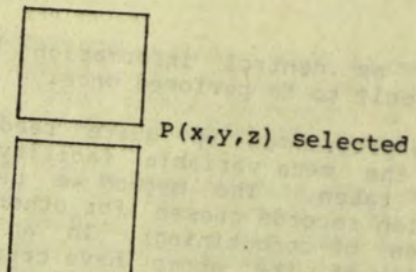
We effect the selection of Q or R by masking these flags after we know the result of trying P. If P succeeds then we pretend that R is already finished, in which case only Q will be started. If on the other hand the P evaluation fails then we set Q as being finished instead. If Q and R are sequences instead of single literals the same method applies, the only thing we need to know are the boundaries between Q and R and P and the rest of the goal.

Sets of Solutions

The goal

$$\leftarrow w = [t(x,y)/P(x,y,z)]$$

results in w being bound to the list of terms $t(a_1,b_2)...t(a_k,b_k).NIL$ where $x/a_i,y/b_i$ are all the solutions to the goal $\exists z\ P(x,y,z)$. We evaluate this by iterating through the proof space of the goal looking for solutions to the goal. Every time P succeeds we construct a copy of t(x,y) in the environment of the proof of P:



Solution to be added is:
t(A,f(u'))

with {x/A,y/f(u),z/B} say

The copy of t(x,y) is then placed in a list of such solutions. Having done the construction of the solution we artificially fail and try to find the next solution of P and so on. When the proof of P(x,y,z) ultimately fails then the list we have built up is bound to the variable w and the equality call succeeded.

When a solution is being constructed any variables appearing in it are given new storage locations on the stack differentiating them from all the other variables.

Notice that we do not attempt to remove multiple occurrences of solutions from the list.

Indexing

Indexing is used to cut down on shallow backtracking particularly when searching through a large relation that is represented extentionally. The programmer asks for indexing by adding an assertion about the relation specifying which arguments are to be indexed.

An index table is constructed for each argument that is specified to have indexing. This table contains references to all the top level function symbols and constants that appear in the relation in that argument position. Associated with each 'key' is a list of clauses that mention the key in the right argument position. A further subset of the relation: those clauses with variables in the argument position are also grouped together. For example the program:

```
P1 P(A,b) <-
P2 P(f(B),C) <-
P3 P(u,C) <-
```

will have the following tables constructed for the two arguments:

```
Arg1  A   P1          Arg2 C   P2,P3
      f   P2               var P1
      var P3
```

When a procedure call involving an indexed relation is about to be entered a check is made first on the constants and function symbols occurring in the call. For each argument that is indexed and that contains a non-variable the appropriate subset is extracted from the index table together with the subset associated with variables. These two lists are unioned together to give a candidate set of clauses for that argument. The candidate sets for all of the indexed arguments are intersected to give the candidate set for the procedure call. Backtracking now only occurs within the candidate set, and typically in optimal conditions it is a single clause.

This indexing scheme is very flexible - there are no arbitry restrictions on the number of arguments that can be indexed, nor do the keys in the index tables have to be unique. Furthermore, the system reverts to sequential access when no use can be made of the index. When there are no non-variable arguments in the call or when more than one clause is returned in the candidate set they are accessed sequentially in the normal way.

Data Flow coroutining

The essential problem with data flow coroutining is to make sure for each variable a) that only 'authorized' sub-proofs are permitted to bind it and b) if a variable has become further instantiated to jump to any sub-proofs waiting for the variable. The data structure we use for this

mechanism is called a "process descriptor" after similar such data structures in more conventional coroutining systems such as operating systems.

The process descriptor model is quite simple: we view each annotated variable as a pipe through which flows a succession of partial approximations. At one end of the pipe is the consumer, and at the other end is the producer. The process descriptor describes the state of the pipe: it tells us whether the consumer or producer is currently active, and it records the last suspension address of the half of the pipe that is not running.

Which subcomputation comprise the producer and consumer processes for a given variable is simply determined. In any clause body at most one occurrence of a variable v can be annotated with a "?" or a "^". Suppose the occurrence is in the atom A_i of a sequence $A_1\&..\&A_m$ of goal atoms of the clause. If the annotation is "?" then the proof of A_i is the consumer process and the proof of the conjunction $A_1\&..\&A_{i-1}\&A_{i+1}\&..\&A_m$ the producer. If the annotation is "^" the producer/consumer roles are reversed. Each atom can have at most one annotated variable.

The default of an IC-PROLOG program is sequential left to right execution. However each time a clause is used which contains annotated variables a process descriptor is set up pointing to the as yet unactivated producer/consumer for each variable. Each variable is then bound to its process descriptor.

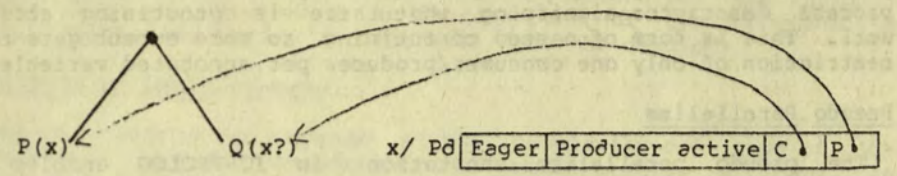
The co-routining interaction is now achieved by a slight modification of the unification algorithm to handle these special process descriptor bindings. Before we overwrite such a process descriptor binding with a non-variable term we must check to see if the current resolution is part of the consumer or producer computation. The term binding is affected only if the step is part of the producer process.

To make this test as simple as possible, we provide a naming scheme for the calls that occur in the proof tree. This name is actually a path to the literal from the root node of the proof tree to any given call. By comparing this path name with the path name in the process descriptor (which is the path name of the call in which the pipe variable appears) we can easily check whether the current resolution step is part of the consumer or producer process.

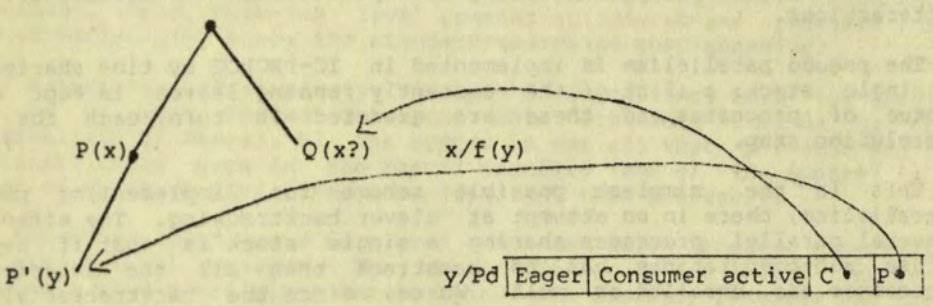
By using binary encoding, and only differentiating literals with annotated variables, the path can be represented as a bit string. We can then use bit string operations to efficiently compare path names.

There are two conditions when the state of the pipe will change: when we tried to bind a variable but were not allowed to (ie. the process descriptor records a suspended producer) and when we have just bound a variable for which there is a suspended consumer recorded. In either case the state of the process descriptor is switched: the current process is suspended, and the suspended process is activated. The folowing diagram shows three steps of an example of a coroutining interaction:

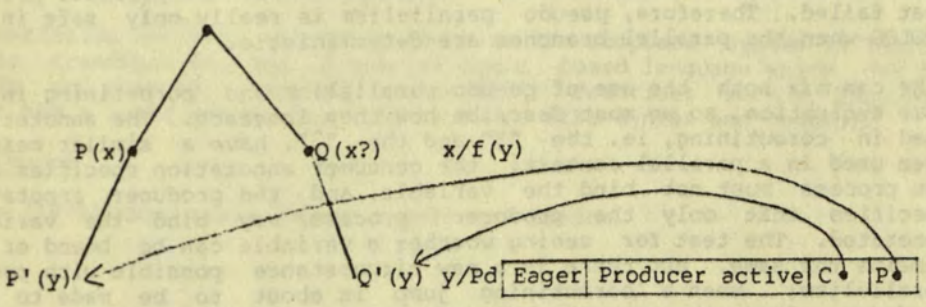
An example coroutining interaction



Initial state of process descriptor with eager consumer



The producer P(x) has just produced a value for x and the suspended consumer on x: Q(x?) is about to be started.



Evaluation of the call Q'(y) tries to bind y.
This is not allowed to do since there is a suspended producer at: P'(y). The suspended producer will now be activated.

When an annotated variable is bound for the first time, or when it is futher instantiated, the annotation must be inherited by all the variables occurring in the binding term. This is because the annotated variable is still the pivot of the coroutining and we must apply the same rules to all the further approximations of the variable.

The annotation is inherited by marking each of the component variables with a pointer (called the channel pointer) to the original variable and hence its process descriptor. Thus a variable may have both a channel pointer, signifying that its part of a higher level coroutining, and a process descriptor signifying that there is coroutining about it as well. This is form of nested coroutining to some extent gets round the restriction of only one consumer/producer per annotated variable.

Pseudo Parallelism

The pseudo parallelism annotation in IC-PROLOG enables several processes to be 'time shared' on a sequential machine. The most common circumstance where one may want parallel evaluation over say a coroutining evaluation is with two or more independent procedure calls which do not interact. However there is also a style of programming which is the dual of using coroutining: namely starting with an all parallel evaluation and restricting the parallelism for certain crucial interactions.

The pseudo parallelism is implemented in IC-PROLOG by time sharing on a single stack: a list of the currently running leaves is kept as a queue of processes and these are executed in turn, each for one resolution step.

This is the simplest possible scheme for implementing pseudo parallelism, there in no attempt at clever backtracking. The effect of several parallel processes sharing a single stack is that if any of these sub-computations has to backtrack then all the interleaved processes are unrolled as well. Worse, since the backtracker always retries the most recent alternative first, and the most recent branch point may not belong to the process that failed, the system will potentially be very redundant in its backtracking search. It will exhaustively retry the other independent branches of the parallel evaluation before it tries another alternative for the parallel process that failed. Therefore, pseudo parallelism is really only safe in IC-PROLOG when the parallel branches are deterministic.

We can mix both the use of pseudo parallelism and coroutining in the same evaluation, so we must describe how they interact. The annotations used in coroutining, ie. the "?" and the "^", have a similar meaning when used in a parallel context: the consumer annotation specifies that the process must not bind the variable, and the producer annotation specifies that only the producer process may bind the variable annotated. The test for seeing whether a variable can be bound or not remains the same, but there is a new circumstance possible with pseudo parallelism: when a coroutining jump is about to be made to the producer, after the consumer has attempted to bind a variable, it is possible that the producer is already running. For example in the goal:

<- P(x)//Q(x?)

the producer of x is running in parallel with the consumer. If P(x) is slow in producing x then Q(x) may attempt to read x too early. In this circumstance we simply move the Q(x) process to the back of the queue of currently executing processes. Each time around the queue the Q(x) will independently see if x has become bound or not, and eventually when P(x) produces the next approximation for x the Q(x) process will automatically resume execution.

Notice that we can detect the deadlock condition quite easily: if we go through an entire 'sweep' of the currently running processes without being able to run any of them then we have a deadlock. In this case we simply run any of them, disregarding the annotations for the variable concerned. This is equivalent to forcing one of the processes to 'guess' its input, and hoping that it will free the deadlock.

Implementation of Logic systems.

Any complete interactive system of the complexity and size of IC-PROLOG demands a careful consideration of implementation strategy. In particular the implementation language needs to be flexible, powerful and preferably portable.

For largely historical reasons and availability we chose Pascal as the implementation language. At the time of this decision Pascal was quite a new language and seemed to offer what we needed, namely good data structuring tools, high-ish level control structures and promised some hope of portability since the standard was quite comprehensive.

Our experience unfortunately has been dissapointing, with the result that a) we were forced into poor data structure choices by the inflexibility of Pascal, b) the system is not all that portable due to some unfortunate gaps in the Pascal standard and c) the system is a factor of three to five times less efficient than necessary.

On the other hand logic is a good implementation language especially for experimental systems. The method we would now adopt involves building a highly efficient kernel system in assembly language and then writing the bulk of the system in logic itself. This promises a faster implementation time, more flexibility and some extra power too. Of course this is the way that the original Marseilles PROLOG was implemented, though we feel that the use of assembler for the kernel is justified rather than using FORTRAN.

A variation on this method is to write the complete system in logic, and to transform it into a special logic based language which can be simply and cheaply compiled. Halim(1979) describes such a language which has a logic semantics, with an ALGOL like syntax and control.

References

Halim [1979] MSc Thesis Imperial College, London

The Comparison of several Prolog systems

C.D.S.Moss

Imperial College, London

The number of Prolog implementations is growing rapidly, as the usefulness of the language and the ease of making an interpreter for it is appreciated. The techniques for these implementations have varied over most of the methods developed in the last 20 years for language systems and their performance varies correspondingly widely. A comparison of the performance of several systems has therefore been made in order to help present users and future implementors assess the relative merits of these techniques in the context of Prolog.

Five implementations are compared:

1. The original Marseilles interpreter, written as a small kernal in Fortran, with a supervisor written in the primitive Prolog language.
2. The Edinburgh compiler, which translates into DEC-10 machine code, with extra machine code and compiled Prolog support routines. The main part of the compiler was itself written in Prolog.
3. The Edinburgh interpreter, also written mostly in compiled Edinburgh Prolog, but avoiding the expensive compilation process.
4. Waterloo Prolog, the first system written totally in assembler for the IBM 370.
5. IC-Prolog, written in Pascal and designed mainly as a testbed for alternative control strategies.

Previous comparisons of Prolog systems have used one or more 'composite' tests to compare Prolog systems, but this is felt to be misleading as there are several distinct measures of efficiency, and the mix in any particular program is hard to disentangle. A number of separate tests were therefore used to measure distinct characteristics of the systems . These were as follows:

1. Basic speed of resolution and tree building; a list of length n is built and reversed using the linear algorithm which makes comparison of widely different systems simple.
2. Printout of results was tested by adding to the above test a printout of the list so built, and this affected the overall run-time of different systems in very different ways.
3. Backtracking speed was tested by using the naive sort algorithm, which generates permutations by deletions and tests them for orderedness.
4. Indexing speed was tested by performing several queries on a medium size database consisting of about 200 clauses describing basic facts about countries of the world.
5. Compilation or read-in speed was tested by reading the same database as described in the test above.

In order to compare several of the Prolog systems, in particular the Edinburgh DEC-10 and Waterloo IBM 370 implementations, which cannot obviously be run on the same machine it was necessary to benchmark the two machines used - the KI-10 processor at Edinburgh and the 370/135 at Imperial College. This was done using two programs written in Pascal for each machine, the implementations on each machine being broadly similar. The programs used were a Pascal version of the list building and reversal program used as test 1, and simple insertion sort. One interesting side result of the first test was that the best of the Prolog systems performed at very similar speeds to a recursively written Pascal program, thus showing that in terms of speed at least, Prolog does not have to apologize any longer to traditional compilers.

The results of these tests are given in the following two tables. The first gives actual run-times, on the two processors, to give an appreciation of the actual machine utilisation. The benchmarking process just described showed the DEC-10 to run at an effective speed of 5.8 times as fast as the IBM 370/135. The second table shows a comparison between the different Prolog systems, using the system that is generally fastest, the Waterloo interpreter, as a basis (given a timing of 1.0 in each case), and allowing for the difference in machine speeds.

Absolute times Test	Marseille Interp: IBM	Edinburgh Compiler DEC	Edinburgh Interp. DEC	Waterloo Interp. IBM	IC-Prolog Interp. IBM
1. Basic speed.	5.02	0.08	0.74	0.20	16.87
2. Speed and printout.	45.11	0.40	1.02	0.33	17.77
3. Backtracking.	7.53	0.36	2.36	1.35	85.03
4. Indexing.	-	0.08	0.46	0.51	29.87
5. Read-in.	(462)	147.0	5.64	3.73	39.06
Relative Times					
1. Basic Speed	25.10	2.32	21.46	1.0	84.35
2. Speed and printout.	136.70	7.03	17.93	1.0	53.83
3. Backtracking.	5.58	1.55	10.14	1.0	62.99
4. Indexing.	-	0.91	5.23	1.0	58.57
5. Read-in	123.86	228.6	8.77	1.0	10.47

The main conclusion that can be drawn from this is the rather surprising one, namely that an interpreter (Waterloo) can perform quite as well as a compiler (Edinburgh) in most areas. This can to some extent be explained by the method of implementation - the Edinburgh system was written by standard bootstrapping techniques which are chiefly designed to save implementation time and give flexibility, and are not renowned for their efficiency. The chief compilation technique shows itself in the indexing test, where the compiler does indeed outperform the interpreter. However, one has to ask how Prolog systems are in fact used, and this question must give great importance to read-in speed (the compiler is extremely slow) and to the printout of results, where the compiler does not significantly shine over an interpreter, as this is done by procedure call anyway.

It is difficult to compare the implementation time for the different systems because the work in most cases overlapped with development and refinement. But it appears that the Waterloo system, being more tightly defined and limited, took probably less time to write than the others, with the possible exception of the Marseilles system (which developed many of the techniques used in other systems).

Comments on the individual systems are as follows:

1. The Marseilles system was the first Prolog system and is understand-

ably less efficient, particularly in input-output (this prevented the completion of the database test, the difficulty of which was compounded by its relatively poor error recovery on input). The current Marseilles system was not available on a compatible machine, but is said to be considerably better. This test does show the necessity for a Prolog system to contain higher level input output procedures.

2. The compiler is considerably more efficient than earlier interpreters, and has considerably more effective space saving features (throwing away of 'local' variables on completion) than either Marseilles or Waterloo. These space saving features would probably slow down the Waterloo system considerably. But the main disadvantage of the compiler is its long compilation time, which means that it is most useful for programs which are used repeatedly: given the experimental nature of most Prolog programming up to this time, these are probably not in the majority.

3. The Edinburgh interpreter is distinguished by its large range of evaluable predicates, which is more comprehensive than any other, and its superior diagnostics, including trace facilities. These support facilities are written in a mixture of compiled Prolog and assembler and, at well over 100 compared with about 40 for Marseille and Waterloo, may be more difficult to learn and use effectively.

4. The Waterloo interpreter is a sensible 'basic' system, omitting some of the features of the other systems, but implementing what it has sensibly. The effect of the assembler implementation is to make it extremely fast, and because of its compactness, it would probably be easier to copy to another machine than the Edinburgh system. Although it doesn't have the Edinburgh space saving features, its data structures are themselves more compact, and thus the proof stack grows less quickly in size - this tradeoff must be carefully considered with regard to both machine characteristics and expected applications.

5. The IC-Prolog system is not shown to advantage by this comparison, as its main purpose, as was stated earlier, was to experiment with different control strategies and also provide a 'clean' version of Prolog. Its behaviour does however teach some lessons. Pascal is probably <u>not</u> a good implementation language for Prolog systems owing to the difficulty of handling data structures such as stacks in the way that most Prolog systems do. Also the 'one level' approach of writing a system entirely in a conventional high level language is probably not desirable. On the other side must be placed the portability of the system, which is already running on at least 3 different machines, and its use of annotations, which reduce an order factorial problem (in the sorting example) to an order n squared.

Is there a 'best' implementation strategy for Prolog? Probably not, although the results of this comparison suggest one possible strategy. The basic kernel of Prolog, which includes resolution, backtracking, and evaluable predicates including a reasonably high level of input output (at the 'term' level) is a relatively small program and can be implemented on most machines in assembler in perhaps 2-3 programmer-months. Other high level facilities can then be added in Prolog, including better syntax, trace facilities and more complex contro facilities. To allow for this requires careful design of the basic evaluable predicates, as all the systems studied have lacunae which make it difficult to implement some of these things. The provision of a 'macro implementation of Prolog' should also be considered.

The MPROLOG System

J.Bendl*, P.Köves**, P. Szeredi**

Abstract

A new PROLOG system is described which facilitates modular programming and efficiency both in execution and in program development.

Introduction

Based on experience with existing PROLOG interpreters a new implementation of the language was developed. This implementation is called the MPROLOG system reflecting the fact that the system provides facilities for modular PROLOG program development.
The aim of our new implementation was to develop a system more suitable for practical purposes. We wanted to reduce time and space requirements to run PROLOG programs, and to provide facilities to aid program development and testing.
The system consists of four components, namely the precompiler, the consolidator, the interpreter and the interactive program development subsystem. The precompiler, the consolidator and the interpreter are implemented in the compiler writing language CDL2, the program development system is itself an MPROLOG program.

* Institute of Industrial Economy and Plant Organisation of the Ministry of Heavy Industries
NIM IGÜSZI, HUNGARY, 1363 Budapest, Pf. 33.

** Institute of Co-ordination of Computer Techniques
SzKI, HUNGARY, 1368 Budapest, Pf. 224.

The precompiler reads an MPROLOG source module, does lexical analysis, syntactical analysis, static semantic checking, optimises and produces a compact internal form of the module.
The modules in internal form which make-up a complete program are consolidated into an interpretable program-module by the consolidator; this is a PROLOG level linkage editor. Such a program-module can be interpreted by the interpreter.
The rest of the paper will describe in more detail the PROLOG language variant accepted by the MPROLOG system and the interpreter and program development subsystem components of the system.
The implementation is now functionally complete [Bendl 79] ; the system is currently undergoing tuning. It runs on SIEMENS 7700 computers under BS2000. The porting of the system to IBM 360/370 and Rjad computers under OS and DOS is underway; this is especially simplified by the exceptionally high degree of portability of software developed in CDL2.
The system was designed by P. Szeredi, K. Balogh, J. Bendl, G. Bogdánfy and P. Köves. The precompiler was written by G. Bogdánfy, M. Kósa, J.né Boda. The consolidator was written by J. Visnyovszky. The interpreter was written by J. Bendl, M. Kósa, L. Naszvadi. The program development suosystem was written by P. Köves.

The MPROLOG language

MPROLOG syntax is similar to that of CDL2 /and also to that of DEC-10 PROLOG/, the well-known procedure "member of a list" for example can be described as follows:

Member(x,x.ℓ).
Member(x,y.ℓ):Member(x,ℓ).

The syntax permits also the use of alternatives and groups. For example the above definition could also be written as

```
Member(x,y.ℓ): x=y;
               Member(x,ℓ).
```

MPROLOG modules contain a so-called interface specification, which declares the module's connections with other modules.
Any symbol /name of an object, name of a predicate, etc/.which is to be used outside this module, must be "exported", and any symbol which is used in this module but not defined must be "imported".

Symbols not involved in module-connections are encoded by the precompiler, and loose their character-form, to save space in the symbol-table.

Thus the internal symbols of different modules in a program will never be confused.

An MPROLOG program may also contain Mode declarations [Warren, 77] and Match_order declarations. In a Match_order declaration one may specify those argument positions of a predicate which should be used in a classification of the clauses for the predicate. Based on this information the precompiler will construct a tree which is used by the interpreter to quickly select that subset of clauses with which unification is feasible for a given call. This subset selection is also advantageous because it facilitates the recognition of determinism with a consequent memory saving.

The interpreter

The main aim of the interpreter of the MPROLOG system is to execute programs using much less memory in comparison with the "old" PROLOG. Furthermore there

is also an increase in time-efficiency mainly due to more elaborate coding and to the high level optimization features of CDL2, the implementation language. Finally the inclusion of algorithms written in a traditional language /as new built-in procedures/ is made simpler.

Space saving in MPROLOG is achieved in two respects. First, due to the precompilation and consolidation techniques, the symbol table of the interpreter contains only those symbols whose character form is needed during the running of the program, i.e. only those that are to be input or to be output. In this way e.g. the names of PROLOG partitions /procedures/ are basically not present in the symbol table of the interpreter.

The second, and more important way of space saving is achieved using the stack-management described in [Warren 77]. Thus the stack regime of the interpreter is split into three parts:

- the main stack containing the administration parts and the so called local variables;
- the global stack containing global variables, and
- the trail that contains information for the undoing operations /at backtrack/.

There are some minor changes in the organisation of stacks. Due to the fact that it cannot be assumed that the target computer's word size permits the storage of two addresses in one word a new mechanism for storing molecules was introduced. The basic directly addressable cells of both global and local stacks are words, thus cannot store a molecule. When one has to store a new molecule in such a cell a double word cell is created on the top of the global stack and

its address is stored instead. Of course care must be taken to avoid unnecessary duplication of molecules, e.g. when an existing molecule is unified with a variable the "old" address of the molecule is simply assigned.

There is an improvement in the interpretation mechanism of MPROLOG aiming at avoiding growth of stacks when performing ordinary loops. The technique applied is a bit more general: when the interpreter reaches the last call in a given clause and there are no backtrack choices up to and including its parent /which is the case in a deterministic loop/, then the whole local frame corresponding to the call is moved down onto the frame of its parent, thus recovering both the administration space and the local cells of the latter. The operation is performed only if there are no cells in the frame of the call pointing to the frame of the parent.

MPROLOG has a feature analogous to the indexing of clauses [Warren 77], as described above under Match_order.

The Program Development Subsystem /PDSS/

The MPROLOG system was designed in such a way as to obtain a very high degree of optimality in the execution of PROLOG programs. For this reason the basic interpreter contains very few facilities to directly support interactive program development. Instead, provisions were made to enable the development of a PDSS in PROLOG.

We envision the use of the MPROLOG system as follows. A PROLOG program consists of several modules. At any point of time a certain number of these modules are

complete and have been translated by the precompiler, a single module is under development in the PDSS, several modules have not yet been written. The PDSS will provide facilities for the linking of precompiled modules and the simulation of the interfaces of nonexistent modules. The user will be conversing with the dialog manager component /DM/ of the PDSS; this will provide programer's assistant services in the style of INTERLISP. He will be able to edit and examine the module under development with a specialized PROLOG editor /EDIT/; the services of the editor will include semi-automated generation of interface specifications for the module. It will be possible to interactively debug and trace programs as well as manage errors /BREAK/.

A program measurement facility will be available which will aid the programmer in selecting candidate procedures for translation into lower-level languages for optimization purposes. Currently only a very minimal DM and the BREAK subsystem are complete. In the sequel these will be described.

The DM currently is simply an MPROLOG program which implements a read-interpret loop. It reads clauses from the terminal and performs actions based on the type of clause read. Two DM commands are available at this time: STOP with the obvious meaning and CONSULT fn which causes reading to switch to the file fn; reading resumes at the terminal when a STOP is read from the file. Clauses which are goals are executed while all other clauses are added to the database.

The BREAK subsystem is responsible for managing errors and providing tracing and debuging facilities. The subsystem is modeled on the BREAK package of INTERLISP. PROLOG systems known to us either provide no tracing

or provide too much. It is our view that it should be possible for the user to control very precisely the amount and type of tracing information he recieves if debuging is to be effective. For this reason the BREAK subsystem makes it relatively difficult for the user to specify the production of a large amount of tracing. The basic trace may be requested by the call

:Trace (pr/n).

Subsequent to the execution of this call the predicate pr will be traced; note however that only the n argument version of pr is selected. The name and arguments of the procedure will be provided at procedure entry and successful procedure exit; only the name of the procedure is displayed on backtracking into the procedure and on failure exit. It is also possible to specify the procedure to be traced as pr_1 IN pr_2, in which case only calls of pr_1 made from pr_2 will be traced. The next level of tracing may be invoked by the call

:Break (pr/n).

In this case execution will stop at procedure entry, exit and on backtracking into the procedure. The user is then in an interactive break. At this point the full facilities of the PDSS are available to him, i.e. he may define new procedures, read files, edit, etc. In addition a set of specialized BREAK commands are available to help in examining the state of the currently running goal. A brief description will now be given of the most important BREAK commands.

BT, BTA, AT, ATA are backtrace commands. They display the call history with or without arguments, ancestors only or all calls.

ARGS displays the arguments of the broken procedure.

WHO displays the name of the broken procedure.

n? displays the n-th argument of the broken procedure.

n=term unifies term with the n-th argument of the broken procedure.

var?, var=term } similar to the above commands var is the name of a variable that occurs in the clause that called the broken procedure.

OK, OK!, OK!! } are commands that exit from the BREAK and cause execution to resume. In addition if OK! is given deeper calls of the broken procedure are not traced, if OK!! is given no deeper call is traced.

ABORT causes exit from the break with the computation aborted. Return is made to the previous level of supervision /BREAK or PDSS/.

SUCCEED, FAIL } a context switch is made to this state of the broken procedure.

HELP the most important command.

When setting the break it is also possible to specify a list of BREAK commands to be executed in addition to or instead of interacting with the user.

In the MPROLOG system errors are controlled through an exception handling mechanism. Errors are system defined exceptions; the user may define his own exceptions. When an exception occurs the procedure call causing the exception is replaced by a call on an exception handler. Exception handlers are specific to the type of the exception. The default handler for all exceptions causes a break to occur. Exceptions may be raised explicitly by a call on the procedure Raise_error. If a handler decides that it cannot adequately handle the exception it may propagate it, in a break

this is done with the BREAK command PROPAGATE. At any point it is possible to issue the call

Error_protect(call, handler).

This call is equivalent to "call" if no exceptions which are propagated occur during the execution of "call". If this is not the case execution of "call" is replaced by the execution of "handler". Of course if exception propagation occurs during the execution of "handler" this is propagated to a higher level Error_protect /if any/.

References

[Bendl 79] J. Bendl, J-né Boda, G. Bogdánfy, M. Kósa, L. Naszvadi, J. Visnyovszky: MPROLOG user's documentation /Hungarian/ NIM IGÜSZI, 1979

[Köves 79] P. Köves: A preliminary user's manual on the debuging and trace subsystem of MPROLOG /Hungarian/ SZKI report, SOFTTECH D32 1979

[Warren 77] D.H.D. Warren: Implementing Prolog-compiling predicate logic programs D.A.I. Research Report No. 39-40. University of Edinburgh 1977

A Set - Oriented Predicate Logic Programming Language

by Jack Minker

The use of predicate logic as a programming language proposed by Kowalski [1974] has been achieved with a number of effective implementation of PROLOG (Colmerauer[1973, 1979], Bruynooghe[1976], Warren[1977], and Roberts[1977]). A listing of a number of programs written in predicate logic has been achieved by Tarnlund[1975] and Pereira[1979].

The control structure available with PROLOG has provided a sequential search with some very powerful additional features. A major problem that remains for predicate logic languages is that of providing a more flexible control structure for wide classes of problems. The control of backtracking as caused by non-determinism associated with fully instantiated unit is one such problem. Some approaches to handling such problems are described by Clark[1979], and by Bruynooghe[1979].

In this paper we discuss a predicate logic programming language based upon set operations. The use of set operations is shown to alleviate some problems associated with backtracking. A clause within this language, referred to as a Π-clause, represents a set of clauses in first-order logic. A Π-clause consists of an ordered pair, $C = (T, \Phi)$, where T is a Horn Clause template and Φ is a finite set of substitution sets, $\Phi = \{\phi_1, \ldots, \phi_n\}$. A Horn clause template is a Horn clause in first-order logic which is free of constants and where the predicate names have been replaced by variables. Each $\phi_i \in \Phi$ is a set of substitutions sets where one may substitute for variables in a template. A substitution for a variable may be a set of constants, a set of predicate names, or boolean combinations of types as in a typed-predicate logic.

Examples of Π-clauses are:

C_1 $\quad C_1 = (\alpha(x,y).,\ \{\{[P]/\alpha\ ,\ [a,b]/x,\ [c_1,c_2,c_3,c_4]/y\ \}\}),$

C_2 $\quad C_2 = (\alpha(z,y) \leftarrow \beta(x,y)\ \&\ \gamma(x,z),\ \{\{[P]/\alpha\ ,\ [F]/\beta\ ,\ [H]/\gamma\ ,\ \underline{male}/x,$
$\underline{human}/y,\ \underline{female}/z\}\}\ ,$

C_3 $\quad C_3 = (\alpha(x_1),\ g(x_2,\ x_3) \leftarrow \beta(f(x_1),\ x_2)\ \&\ (h(x_1),\ x_3),$
$\{\{[F]/\alpha\ ,\ [F]/\beta\ ,\ [F]/\gamma\ ,\ \underline{integer}/x_1,\ \underline{integer}/x_2,\ \underline{integer}/x_3\}\}).$

The Π-clause, C_1 may be interpreted as the eight unit clause in the first-order predicate calculus.

$P(a,\ c_1).$ $\qquad$ $P(b,\ c_1).$

$P(a,\ c_2).$ $\qquad$ $P(b,\ c_2).$

$P(a,\ c_3).$ $\qquad$ $P(b,\ c_3).$

$P(a,\ c_4).$ $\qquad$ $P(b,\ c_4).$

If the predicate P is interpreted as PARENT, then the children of a and b are specified by the Π-clause and the corresponding unit clauses.

The Π-clause C_2 corresponds to a single predicate calculus clause where the variables belong to different types. That is, it corresponds to the typed first-order clause,

$(\forall x \in \underline{male})\ (\forall y \in \underline{human})\ (\forall z \in \underline{female})\ \ (P(y,z) \leftarrow F(x,y)\ \&\ H(x,z))$

The Π-clause C_3 corresponds to the typed first-order clause,

$(\forall x_1\ \forall x_2\ \forall x_3 \in \underline{integer})\ \ (F(x_1,\ g(x_1,\ x_3)) \leftarrow F(f(x_1),\ x_2),\ F(h(x_1),\ x_3).$

If we interpret

$F \equiv$ FIBONACCI

$f(x_1) \equiv x_1 - 1$

$h(x_1) \equiv x_1 - 2,$

$g(x_2,\ x_3) \equiv x_2 + x_3,$

then the Π-clause C_3 defines the FIBONACCI numbers.

The notation of Π-clauses (Fishman and Minker[1975]) permits sets of first-order clauses to be treated as individual clauses. It effectively permits

all solutions for proof paths having the same template structure to be obtained in paralled. If so desired, the user may avoid the set operation features by permitting substitutions to consist of individual constants. A typeless first-order system may be obtained by allowing only the universal type.

An operational prototype system which runs in an interpretive mode on the UNIVAC 1108 has been developed. All predicates within the system are indexed on all argument positions. A call for unification of a literal with entries in the database consisting of unit and non-unit Π-clauses results in all Π-clauses being returned that have a literal that can unify with the given literal. The unification algorithm permits two Π-clauses to be unified. The algorithm performs type-checking, that is, a variable to be substituted for another variable must have a type that overlaps with the type of the variable for which it is to be substituted to be acceptable. The type of the new variable then becomes the type of the overlap between the two types. Thus, dynamic type-checking is achieved (McSkimin[1976], McSkimin and Minker[1977,1979]).

The inference system of LUSH-resolution is used with the bookkeeping developed for LUST-resolution (Minker and Zanon[1979]). If desired, the system can run in a trace mode which permits every Π-clause in the search space to retain the history of its derivation. This feature can be useful in a debugging mode.

The control structure (Minker[1978]) permits literals to be selected in any position in a Π-clause. A Π-literal selected in a Π-clause is expanded by the Π-clause with best "merit". A dynamic generalized "slash" operator is provided within the control structure of the system. For a given predicate for which $m \geq 1$ solutions pertain, "slash(m)" will terminate expansion of the literal when m solutions are found for the literal.

Additional features which permit the ability to output answer and reason steps in symbolic, natural language, or voice output are noted (Minker and Powell[1979]). Experimental results will be presented.

REFERENCES

1. Bruynooghe, M. [1976] "An Interpreter for Predicate Logic Programs" Report CW 10, Katholieke Universiteit Leuven (Belgium), Oct. 1976.

2. Bruynooghe, M. [1979] "Solving Combinational Search Problems by Intelligent Backtracking" Report CW 18, Katholieke Universiteit Leuven (Belgium), September 1979.

3. Clark, K. and McCabe, F.G. [1979] "The Control Facilities of IC-PROLOG CCD Report, Imperial College, London 1979.

4. Colmerauer, A. et al. [1973] "Un Systeme de Communication Homme - Machine en Francais" Groupe de IA, UER de Luming Univ., d'Aix Marseille, France.

5. Colmerauer et al. [1979] "Etude et Realisation d'Un Systeme PROLOG Groupe de IA, UER de Luming Univ. d'six Marseille, France.

6. Fishman, D.H. and Minker, J. [1975] "Π-Representation - A Clause Representation for Parallel Search" Artificial Intelligence 6, 103-127.

7. Kowalski, R. [1974] "Predicate Logic as Programming Language" Proceedings of the IFIP Congress 1974, North Holland Publishing Company.

8. McSkimin, J.R. [1976] "The Use of Semantic Information in Deductive Question-Answering Systems" Ph.D. Dissertation, Department of Computer Science, University of Maryland, College Park.

9. McSkimin, J.R. and Minker, J. [1971] "The Use of a Semantic Network in a Deductive Question-Answering System" Proceedings IJCAI-77, Cambridge, Mass., 1977, 50-58.

10. McSkimin, J.R. and Minker, J. [1979] "A Predicate Calculus Based Semantic Network for Deductive Searching" In: Associate Networks: The Representation and Use of Knowledge (N. Findler, Ed.), Academic Press Inc., New York, 205-238.

11. Minker, J. [1978] "An Experimental Relational Data Base System Based on Logic" In: Logic and Data Bases (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, 107-147.

12. Minker, J. and Powell, P.B. [1979] "Answer and Reason Extraction, Natural Language and Voice Output for Deductive Relational Data Bases" In: Natuaral Language Based Computer Systems (L. Bolc, Ed.)

13. Minker, J. and Zanon, G. [1979] "LUST Resolution: Resolution with Arbitary Selection Function" Univ. of Md. Tech. Report TR-736, February 1979.

14. Pereira, L.M., Coclho, H. and Cotta, J.M. [1979] "How to Solve It With PROLOG" Laboratorio Nacional de Engenharia Civil, Universidade Nova de Lisboa, Lisbon.

15. Roberts, G.M. [1977] "An Implementation of PROLOG" DCS, Univ. of Waterloo, Canada.

16. Tarnlund, S.A. [1975] "Logic Information Processing" Report TRITA - IBADB - 1034 Dept. of Information Processing, University of Stockholm, Sweden.

17. Warren, D.H. [1977] "Implementing PROLOG - Compiling Predicate Logic Programs" Univ. of Edinburgh, DAI Research Report 39 and 40.

The meaning of logical programs (abstract)

BRIAN MAYOR ÅRHUS UNIV

In a recent paper Kowalski ( ) has advocated replacing the slogan "Algorithm - Program + Data Structure" by the slogan "Algorithm = Logic + Control". If the semantics of a programming language are to give us a function M from Algorithms to Meanings, the second slogan suggests defining this function as a member of

Logic → (Control → Meanings)

In this paper we use this factorization of M to bring some uniformity in the definition of the semantics of logical programming languages like LUCID and PROLOG. We describe how a context free grammar can be assigned to each logical program and we identify Control with the language generated by the grammar. This reduces the problem of defining the function M to the problem of defining the semantics of a traditional programming language because the syntax of such a language is given by a grammar and the semantics gives a meaning to any of the "programs" generated by the grammar.

In the sections on AND/OR (Harel), PROLOG, LUCID, and extended attribute grammar logical programming we show that our approach can give the "official" semantics of the language - showing this is the main part of the paper but it cannot be conveniently abstracted - and we illustrate the approach by the same two examples. We have used the example of square root extraction because it is simple enough to make our basic idea clear; we have taken the LUCID primality example

```
N = first input
first I = 2
begin
      first J = I × I
      next J = J + I
      D = J eq N  as soon as  J ≥ N
end
next I = I + 1
output = ¬ D  as soon as  D ∨ I × I ≥ N
```

because it shows how our approach can handle the more obscure parts of LUCID's semantics.

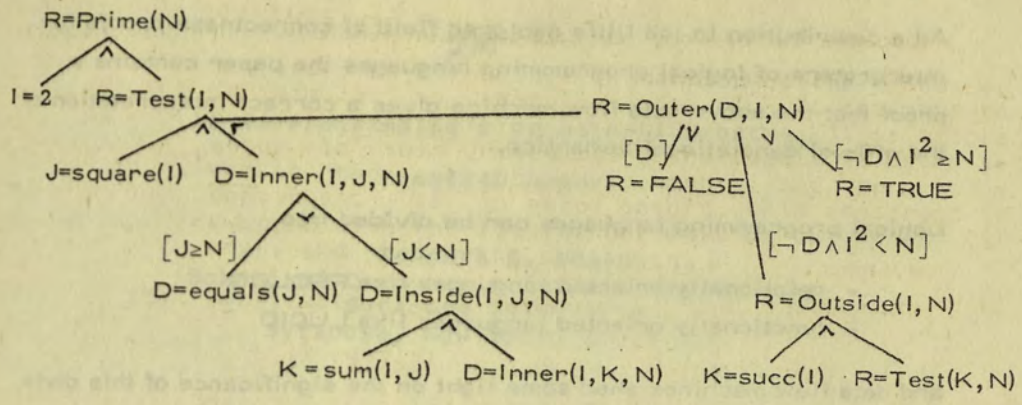
Attribute grammars are not usually thought of as logical programs, but in the extended form (Watt & Madsen) they can be very convenient. For those without access to attribute grammar based compiler generating systems ( ) this convenience may not be obvious until one sees the extended attribute grammar solution of the primality example

```
Prime(↓N ↑R) : : = Test (↓2 ↓N ↑R)
Test  (↓I ↓N ↑R)  :: = Inner(↓I ↓I×I ↓N ↑D)  Outer(↓D ↓I ↓N ↑R)

Outer (↓TRUE ↓I ↓N ↑FALSE) ::=
Outer (↓FALSE ↓I ↓N ↑R) :: = less than (↓I×I ↓N ↑TRUE) Test(↓I+1 ↓N↑F
Outer (↓FALSE ↓I ↓N ↑TRUE) :: = less than (↓I×I ↓N ↑FALSE)

Inner((↓I ↓J ↓N ↑J=N) :: = less than (↓J ↓N ↑FALSE)
Inner (↓I ↓J ↓N ↑D) :: = less than (↓J ↓N ↑TRUE) Inner (↓I ↓J+I ↓N↑D)
```

This solution should be compared with the PROLOG solution

```
Prime (N, R) ← Test (2, N, R)
Test (I, N, R) ← square (I, J) Inner (I, J, N, D) Outer (D, I, N, R)
Outer (TRUE, I, N, FALSE) ←
Outer (FALSE, I, N, R) ← square(I, J) less than (J, N, TRUE)
                                  SUCC(I, K)  Test(K, N, R)
Outer (FALSE , I, N, TRUE) ← square (I, J) less than (J, N, FALSE)
Inner (I, J, N, D) ← less than (J, N, FALSE) eq (J, N, D)
Inner (I, J, N, D) ← less than (J, N, TRUE) sum (I, J, K) Inner (I, K, N, D)
```

and the AND/OR solution



These logical programs would have been much simpler if we were content with showing that composites are not primes. All four of them have the underlying grammar

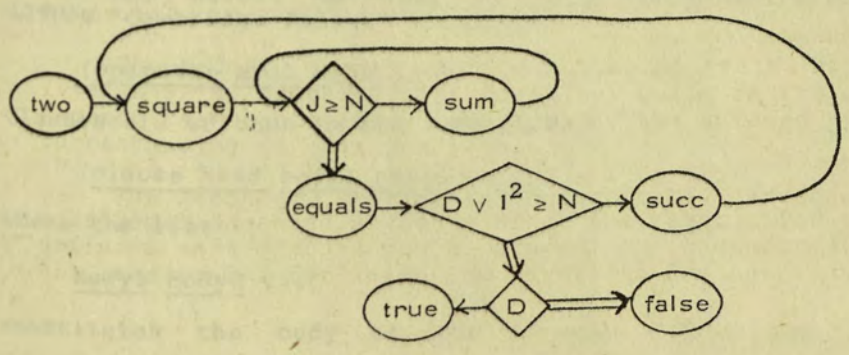
Prime : : = two Test

Test : : = square Inner Outer

Outer : : = [D] false | [¬D ∧ I^2 < N] succ Test
| [¬D ∧ I^2 ≥ N] true

Inner : : = [J ≥ N] equals | [J < N] sum Inner.

There is a close connection between logical programs and data flow machines because both of them abandon assignment and stores. The last section of the paper uses data flow machines to give an operational semantics of logical programming languages that can be compared with 1) their official denotational semantics 2)actual interpreters. The data flow machine for our primality example is



As a contribution to the little explored field of correctness proofs for interpreters of logical programming languages the paper contains a proof that the above data flow machine gives a correct interpretation of the official denotational semantics.

Logical programming languages can be divided into

- relationally oriented languages like PROLOG
- functionally oriented languages like LUCID

and data flow machines shed some light on the significance of this division.

Logic Programming & Relational Databases

Progress Report

Kenneth A. Bowen
School of Computer & Information Science
Syracuse University
Syraucse, New York, 13210 USA

This is a progress report on a project investigating the coupling of a logic programming system with a relational database management system. The point of view in this endeavor is that of the logic programming system. The goal is seen as the addition of facilities to the logic programming system which will make it possible for the logic processor to efficiently store and retrieve its clauses on a combination of backing store and primary memory, instead of primary memory alone as has been the case in most previous general-purpose logic programming systems. The use of these additional storage mechanisms is transparent to the logic programs and to the casual user. The action of these mechanisms is controlled by the use of assertions in the processor's database of clauses.

The logic programming system being used is based on one designed and implemented by Robinson and Sibert in LISP. (The highly developed facilities of LISP made it quite suitable as a "systems programming" language. However, the choice of this approach was most strongly conditioned by the goal of Robinson and Sibert to effect a merger of LISP and logic in which the top-level logic processor is a LISP-callable function, and arbitrary LISP function expressions can occur as evaluable terms in logic clauses.) Terms and atomic formulas in this system are written in LISP's "Cambridge Polish" notation:

(operator arg1 arg2 ...)

Clauses are written in the list format

(clause head body1 body2 ...)

where the list

body1 body2 ...

constitutes the body of the clause. Thus the logic

definition of the append relation, usually presented as

```
append(NIL, _Y, _Y)
append(_H._T, _Y, _H._Z)<--append(_T, _Y, _Z)
```

is now written as:

```
((append NIL _Y _Y))
((append (_H . _T) _Y (_H . _Z))(append _T _Y _Z))
```

(Of course a front-end allowing input in the usual format could be added to the system.)

The top level of the logic system is a LISP function called LOGIC_PROCESSOR. It accepts two arguments, HOW_MANY? and GOALS. The latter is a list of literals collectively consitituting the initial goal clause for the processor. HOW_MANY? is a bit of control information. At present, its acceptable values are non-negative integers and the atom ALL. The output of LOGIC_PROCESSOR is a list L of substitutions S described as pairs

S = (VARS VALS)

where VARS is a list of the logic variables occurring in GOALS. Each such substitution S is a successful solution for GOALS relative to the current database of clauses. The length of the list L is determined as follows. L is the longest non-repetitive list of substitutions satisfying GOALS such that length(L) $\leq$ HOW_MANY?, where the atom ALL is treated as positive infinity.

Beyond this control facility, the principal other features of the processor alone are:

1. When a literal (pred args) rises to the top of the processing stack, the system first checks to see whether pred occurs on the list PRIMITIVE_SYSTEM_RELATIONS. If so, there should exist a LISP definition for pred as a (boolean) LISP function. The processor attempts to run this function as a LISP function. If pred successfully runs (without error) and returns a non-NIL value, this literal is regarded as successfully solved, and is popped off the stack. If pred runs without error and returns value NIL, the literal is regarded as logically unsolvable, and the current branch of the computation tree is failed. Finally, if an error occurs during the LISP execution of (pred args), and if there remains another literal below (pred args) on the stack, then this latter literal is swapped with (pred args) on the stack, and processing continues. (The error typically occurs because of the presence of incompletely bound logic variables in args.)

2. Constructs of the form

(IS var S-expression)

may occur as literals. The action of the processor is similar to that in case 1. It attempts to evaluate S-expression as a LISP term. If no errors occur in this evaluation, the result is bound to var in the current environment, and processing continues. If an error occurs, this construct is swapped with the literal below it on the stack (if any), and processing continues.

3. Constructs of the form

(? goals)

may occur as literals. These are recursive calls of the logic processor. Essentially, (LOGIC_PROCESSOR 1 goals) is run. If a solution is found, the current environment is extended by the bindings created by this solution, and processing coninues with the remainder of the stack.

The direct interface to the database management system is via the two functions RESOLVENTS and RETRIEVE. RESOLVENTS takes a logic procedure-call as its single argument, and returns a list of all procedure bodies whose heads successfully matched the given procedure call. To obtain the candidates for matching, RESOLVENTS constructs a concrete literal out of the virtual representation in the procedure-call which it was passed, and passes this concrete term to RETRIEVE. The latter function extracts appropriate information from the concrete term which it recieves, accesses the relational database machinery, and returns to RESOLVENTS a list of all candidate matching procedures.

The storage strategy and mechanisms in use at present are as follows. All conditional clauses together with all indefinite unit clauses (i.e., those with at least one logic variable) of a procedure are stored in primary storage. Definite unit clauses may be stored either in primary storage or on backing store. (The default is primary store.)

The primary memory storage mechanism is a hash table (via LISP's property list facility) built on the predicate names. Each table entry is a list of the entered procedures corresponding to that prediate.

The secondary storage is conceptually arranged in a relational database style -- hence the restriction that only definite unit clauses can be stored on secondary storage. All the tuples for which the predicate has been asserted to

hold are conceptually grouped together (though they may be distributed physically in blocks on different pages of secondary memory.) The system automatically provides indexing on the first element of a tuple. The user has the option of requiring indicies on other arguments or combinations of arguments. The indicies at present are dense indicies implemented by means of unbalanced binary search trees. The trees themselves are maintained on secondary storage, and the system is arranged so that on retrieval, only a small portion of the tree need be read in from secondary storage in order to make a key match. Any number of indicies may be maintained for a given predicate.

Let pred be a primitive predicate with a large extension stored on secondary storage, and no conditional clauses in its definition. Normally a call of the form

(pred args-with-vars),

where at least one variable occurs, would return a list of solutions grouped in an unusable order reflecting the actual physical distribution of the tuples of pred on the pages of secondary storage. However, one can control this order by adding clauses to the database of the form

(TRAVERSE pred index).

Here index is a description of an index for pred which is being maintained by the system. The effect of the presence of this clause is that calls of the form described above will return the solutions in an order reflecting the traversal of the index tree. It is planned to experiment with other such control predicates, such as "run in-primary-storage {conditional} clauses before {after} database accesses," etc.

Though the present database access and storage procedures have been programmed in LISP, it is planned to add other methods (specifically more sophicticated tree structures) by programming them in logic and allowing the logic processor to execute them. Potential sources of circularity here are of course avoided by maintaining all of these definitions in primary storage.

Small experimental toy databases have been constructed using this system, and the performance has been quite satisfactory. At present a real-world bibliographic database using this system is under construction. In the first phase it will consist of approximately 2000 conceptual items. The raw descriptions of these items (library periodicals) averages about 400 characters each. In the internal form, the entities are described using from 15 to 25 predicates, almost all of them binary. The fields of most of the predicates are quite variable strings. Since much of the user-access in this application involves

extensive string pattern-matching (e.g., for titles or variants of names, etc.), it is planned to implement a fast pattern-matcher at a low level and interface it to the logic system by means of facilities 1/ and 2/ above. Performance experience for this application will be available by July.

The point of this project is not construct a "production" system, but rather to create an experimental vehicle suitable for a few selected real applications, and to gain experience from those applications. Presumably the experience resulting from the system construction and from the applications will make possible the construction of a high-class system of logic programming capable of deal with extremely large amounts of basic data. Potential applications range from ordinary databases with very user-friendly front-ends (perhaps even natural language) to knowledge-based expert systems.

EXTENDED ABSTRACT

Hoare's Program FIND Revisited

by

Sharon Sickel
William McKeeman

Hoare [Hoare, 71] gave a proof of correctness of the algorithm, FIND. In this paper we give a different proof of a computationally equivalent algorithm. The proof is based on a sequence of logic programs that get progressively more specific, and successive members of the sequence are shown to be equivalent. Our goal is not to find fault with the groundbreaking work of Hoare in this first non-trivial proof of correctness to appear in the literature, but rather to extend the concepts to a different format that we feel is conducive to clarity of program design. We feel that this derivation and proof of the algorithm are more intuitive than Hoare's treatment of the algorithm, and the derivation actually is part of the program development. One does not need to know the final solution at the beginning and in fact, this series of logic program refinements serves two useful roles: 1) as a history of the program development, 2) as the framework for the correctness proof.

The problem solved by FIND is as follows: Given an array A of N comparable elements, and a natural number, f, permute the array such that A(f) contains element r and all smaller elements appear earlier in the array and all larger elements later. This is similar to one iteration in the Quicksort algorithm. Hoare's algorithm appears in Table I. We shall derive a logic program that works equivalently.

PROGRAM TRANSFORMATION BY A FUNCTION THAT MAPS SIMPLE LISTS ONTO D-LISTS

Åke Hansson and Sten-Åke Tärnlund

UPMAIL
Computer Science Department
Uppsala University
Sweden

Introduction

We shall introduce a function that maps a simple list to a d-list, which was formalized in Clark and Tärnlund [1977] . This function gives a convenient method for developing programs. The main idea to transform a program to another program by data structure mappings seems to go back to Burstall and Darlington [1975]. We have made use of it for logic programs in Hansson and Tärnlund [1979a,1979b] and we shall develop that idea further here. The success of this method is dependent on whether or not there are dextrous mappings between data structures. The main contribution of this paper is the formalization of such a mapping between simple lists and d-lists in definition 1. The merits of our mapping function is reflected by the short derivations of programs on d-lists from programs on simple lists. This is fortunate since derivations of programs are usually quite long as has been demonstrated by works already mentioned in the text as well as by Manna and Waldinger [1978], Hogger [1979] and Clark and Darlington [1980].

Deriving programs by data structure mappings

It is of course easier to derive a program on a simple data structure. However, to make this program efficient we may have to substitute the data structure and change a few procedures. The latter step can be taken as we shall see with a proper mapping function. Let us illustrate this method by an example. Suppose that we specify the idea of sorting in the following way; y is a sorted version of x exactly when y is ordered and a permutation of x. This is written more precisely next.

$$\text{sort}(x)=y \leftrightarrow \text{ordered}(y) \;\&\; \text{permutation}(x,y) \qquad (1)$$

(Universal quantifiers are omitted in front of the entire sentence). The definitions of ordered and permutation are not important here so we leave them out. Quite a lengthy derivation of the following quick sort program from the specification in (1) is given in Hansson [1980].

The result of sorting the empty list is the empty list, and moreover the result of sorting a list x.y is to append a list consisting of the element x followed by a quick sorted list y" to a quick sorted list y' if the ordered pair (y',y") is the result of partitioning the list y with respect to x such that all elements less than x are on y' and those greater than x on y". We can now write this program in our programming language (see Hansson, Haridi, Tärnlund [1980]).

```
q(0)=0
q(x.y)=append(q(y'),x.q(y")) <- partition(x,y)=(y',y")        (2)
```

The program makes use of a functional notation and two data structures: a simple list written x.q(y"), where x is the first element and q(y") the rest of the list, and a Cartesian product written (y',y"). Partition is of little interest here so we can leave it out. The inefficiency of program (2) is due to append so we may as well wright it down. The result of appending a simple list u to the empty list is u, moreover, the result of appending a simple list z to a simple list x.y is a simple list where x is the first element and the rest is the result of appending z to y. We have:

```
append(0,u)=u
append(x.y,z)=x.append(y,z)                                   (3)
```

We have to use append in this program since we are using a simple list as data structure. But we would like this program to be a bit more efficient e.g., by a new data structure where we do not need append. So our problem may be solved by a data structure mapping that also takes away the append procedure. Let us charcterize a mapping m from simple lists to d-lists that as we shall see solves our problem. The function m maps a simple list, composed of a simple list y.z appended to a simple list x, to a d-list <u,w> exactly when m maps the simple list x to the d-list <u,y.v> and the simple list z to the d-list <v,w>.

```
m : simple list -> d-list

Definition 1

m(append(x,y.z))=<u,w> <--> m(x)=<u,y.v> & m(z)=<v,w>

m(0)=<u,u>
```

The quick sort program in (2) on a simple list and definition 1 with our mapping function m lead to an equivalent quick sort program (big Q) on d-lists.

Definition 2

```
Q(y)=<u,w> <--> m(q(y))=<u,w>
```

We can now write down a short derivation in our natural dedution system that arrives at a quick-sort program on d-lists from our program in (2).

```
1. Q(x.y)=<u,w> <--> m(q(x.y))=<u,w>          UI def. 2
2. Q(y')=<u,x.v> <--> m(q(y'))=<u,x.v>        UI def. 2
3. Q(y")=<v,w> <--> m(q(y"))=<v,w>            UI def. 2
4.* Q(y')=<u,x.v> & Q(y")=<v,w>               Hypothesis
5.* m(q(y'))=<u,x.v> & m(q(y"))=<v,w>         <--> elim. 2,3,4
6.* m(append(q(y'),q(y"))=<u,w>               <--> elim. 5 and def. 1
7.**  part(x.y)=(y',y")                       Hypothesis
8.**  q(x.y)=append(q(y'),q(y"))              --> elim. 7 and (2)
9.**  m(q(x.y))=<u,w>                         Identity 6 and 8
10.** Q(x.y)=<u,w>                           <--> elim. 9 and def. 2
11. Q(x.y)=<u,w> <-- Q(y')=<u,x.v> &          -->  intro. 4 and 7
                     Q(y")=<v,w>  &
                     part(x.y)=(y',y")
```

The base case gives immediately: Q(0)=<u,u> that we leave for the reader to check. So, together with this base case and step 11 we have derived an efficient quick-sort program on d-lists.

```
Q(0)=<u,u>
Q(x.y)=<u,w> <- Q(y')=<u,x.v> & Q(y")=<v,w> &
                                part(x.y)=(y',y")          (4)
```

Formal program development can, of course, not only be applied to derivations of programs from (abstract) specifications. It may also be applied to program transformations and in this way yield alternative programs of which some can be more efficient than the original programs. In fact, the quick sort program on d-lists above is an example of such a transformation. We shall give a final illustration of a transformation where our mapping function in definition 7 is very useful. Suppose that we want to reverse a list and have the following program. The result of reversing an empty list is the empty list, and moreover the result of reversing a list x.y is to append the list x.0 to the list y' if y' is the result of reversing the the list y. We can now write this program more precisely, where we exploit the functional notation.

```
rev(0)=0
rev(x.y)=append(rev(y),x.0)                            (5)
```

This program is also inefficient due to the behaviour of append, so we want an equivalent program that does not make any use of append. For this purpose we define from (5) and definition 1 a reverse relation (big R) on d-lists which we can make use of to develop a reverse program on d-lists.

Definition 3

R(z)=<u,w> <--> m(rev(z))=<u,w>

We can now derive a more efficient program on d-lists.

1.	R(x,y)=<u,w> <--> m(rev(x.y))=<u,w>	UI def.3
2.	R(x.y)=<u,w> <--> m(append(rev(y),x.0))=<u,w>	Identity 1 and (5)
3.	m(append(rev(y),x.0))=<u,w> <--> m(rev(y))=<u,x.w>	Identity, *
4.*	R(y)=<u,x.w>	Hypothesis
5.*	m(rev(y))=<u,x.w>	-> elim. 4, def. 3
6.*	m(append(rev(y),x.0))=<u,w>	-> elim. 3, 6
7.*	R(x,y)=<u,w>	-> elim. 6, 2
8.	R(x.y)=<u,w> <- R(y)=<u,x.w>	-> intro 4, 7

The last step together with a trivial base case comprise an efficient program for reversing a list:

R(0)=<u,u>
R(x.y)=<u,w> <- R(y)=<u,x.w>

(6)

Conclusion

We have developed our programs by hand derivations in a natural deduction system. This is pleasant when there are methods like the mapping function in definition 1 available and moreover, the programs are of modest size. However, it is not difficult to make an error along a derivation, so, it would be helpful to have software in which we could make these derivations comfortably and get them checked out. We have in fact used two such semi-automatic systems, FOL (see Filman and Weyhrauch [1976]) and NATDED (see Hansson and Johansson [1980]). Both these systems are natural deduction systems of Prawitz [1965] type and make it possible to carry out quite complicated derivations of programs.

---

* Note that we are also using equality for d-lists in step 3 i.e., <u,u>=<v,w> leads to v=w.

References

Burstall R.M. & Darlington J. [1975] Some transformations for developing recursive programs, Proc. Int. Conf. Reliable Software, Los Angeles, California, pp 465-472

Clark K. & Darlington J. [1980] Classification through Synthesis, To appear in the Computer Journal

Clark K., & Tärnlund S-Å [1977] A First Order Theory of Data and Programs, Proc IFIP Congress 1977, North-Holland Publishing Company Amsterdam

Filman R.E. & Weyhrauch R.W. [1976] An FOL Primer, Stanford AI-lab, Memo AIM-288, Computer Science Dept. Stanford University

Hansson Å. & Tärnlund S-Å. [1979a] A Natural Programming Calculus, 6th Int. Joint Conference on Artificial Intelligence, Tokyo, 20-24 August

Hansson Å. & Tärnlund S-Å. [1979b] Derivations of Programs in a Natural Programming Calculus, Electrotechnical Laboratory Tokyo August 27-28

Hansson Å. [1980] A Formal Development of Programs, Ph.D. Thesis, Computer Science Dept., Royal Institute of Tech. and University of Stockholm

Hansson Å & Haridi S. & Tärnlund S-Å. [1980] Some Aspects on a Logic Machine Prototype, Logic Programming Workshop, John von Neumann Computer Science Society, Hungary, 14-16 June

Hansson B. & Johansson A-L. [1980] Development of Software for Deductive Reasoning, Logic Programming Workshop, John von Neumann Computer Science Society, Hungary, 14-16 June

Hogger C. [1979] Derivation of Logic Programs, Ph.D. Thesis, Dept. of Computing and Control, Imperial College, London

Manna Z. & Waldinger R. [1978] A Deductive Approach to Program Synthesis Technical Report, Computer Science Dept. Stanford University, Stanford and Artificial Intelligence Center, SRI International, Menlo Park, Ca.

Prawitz D. [1965] Natural Deduction, A Proof-Theoretical Study, Almqvist & Wiksell, Stockholm

DEVELOPMENT OF SOFTWARE FOR DEDUCTIVE REASONING

Bertil Hansson and Anna-Lena Johansson

Department of Information Processing and Computer Science,
The Royal Institute of Technology
and
The University of Stockholm
Sweden

1. Introduction

This paper is a progress report on software development for deductive reasoning in a semi-automatic or automatic mode. The implementation language is DECsystem-10 PROLOG [PEREIRA-78].

The work has been supported by the Swedish Board for Technical Development.

The increasing use of computers in to-day's society leads to strong demands on the programs used. We want to make sure that the programs are correct in different aspects. This is a difficult problem to solve and we expect that a solution is not found unless advanced software for program reasoning is developed. A computer-based system with its high accuracy, speed and endurance ought to be a great aid to the creative but error-prone human mind.

In a programming calculus, where the data structures are axiomatized in first-order predicate logic with identity and the programs are predicate logic programs (Horn-clauses), a natural deduction system can be used to

- prove that a given program fulfills specified properties
- prove the correctness of a given program
- synthesize programs
- transform programs

This is illustrated in [CLARK-77], [HANSSON-79A], [HANSSON-79B] and in [HANSSON-80].

As a first step towards automatic reasoning about programs and program correctness we build an interactive system, NATDED, in which natural deductions can be performed in a comfortable way. In NATDED a proof is interactively developed stepwise using the rules of natural deduction. NATDED checks the validity of the steps. The natural deduction system on which NATDED is based can be studied in [PRAWITZ-65].

EXTRAPOSITION GRAMMARS

Fernando Pereira
Department of Artificial Intelligence
University of Edinburgh

Abstract

"Extraposition grammars" are an extension of "definite clause grammars", and are similarly defined in terms of logic clauses. The extended formalism makes it easy to describe left extraposition of constituents, an important feature of natural language syntax.

1. Introduction

This paper presents a grammar formalism for natural language analysis, called extraposition grammars (XGs), based on the subset of predicate calculus known as definite, or Horn, clauses. It is argued that certain important linguistic phenomena, collectively known in transformational grammar as left extraposition, can be better described in XGs than in earlier grammar formalisms based on definite clauses.

The XG formalism is an extension of the definite clause grammar (DCG) [5] formalism, which is itself a restriction of the original grammar formalism based on definite clauses, Colmerauer's metamorphosis grammars (MGs) [2]. Thus XGs and MGs may be seen as two alternative extensions of the same basic formalism, DCGs.

The argument for XGs will start with a comparison with DCGs. I should point out, however, that the motivation for the development of XGs came from studying large MGs for natural language [3, 7].

The relationship between MGs and DCGs is analogous to that between type-0 grammars and context-free grammars. So, some of the linguistic phenomena which are seen as rewriting one sequence of constituents into another might be described better in a MG than in a DCG. However, it will be shown that rewritings like those involved in left extraposition cannot be easily described in any of the two formalisms.

Left extraposition has been used by grammarians to describe the form of interrogative sentences and relative clauses, at least in languages such as English, French, Spanish and Portuguese. The importance of these constructions, even in simplified subsets of natural language, such as those used in database interfaces, suggests that a grammar formalism should be able to express them in a clear and concise manner. This is the purpose of XGs.

The reader is expected to have had some previous contact with grammar formalisms based on definite clauses [2, 5], with Prolog [9] and with the syntax conventions of DEC-10 Prolog [6].

2. Left Extraposition

Roughly speaking, left extraposition occurs in a natural language sentence when a subconstituent of some constituent is missing, and some other constituent, to the left of the incomplete one, represents the missing constituent in some way. It is useful to think that an empty constituent, the trace, occupies the "hole" left by the missing constituent, and that the constituent to the left which represents the missing part is a marker, indicating that a constituent to its right contains a trace (cf. [1]). One can then say that the constituent in whose place the trace stands has been extraposed to the left, and, in its new position, is represented by the marker. For instance, relative clauses are formed by a marker, which in the simpler cases is just a relative pronoun, followed by a sentence where some noun phrase has been substituted by a trace. This is represented in the following annotated surface structure:-

The man that$_i$[$_s$John met t_i] is a grammarian.

In this example, t stands for the trace, 'that' is the surface form of the marker, and the connection between the the two is indicated by the common index i.

The concept of left extraposition plays an essential role, directly or indirectly, in many formal descriptions of relative and interrogative clauses. Related to this concept, there are several "global constraints", the "island constraints", that have been introduced to restrict the situations in which left extraposition can be applied. For instance, one of those constraints, the Ross complex NP constraint, (cf. [8]), implies that any relative pronoun occurring outside a given noun phrase cannot be bound to a trace occurring inside a relative clause which is a subconstituent of the noun phrase. This means that it is not possible to have a configuration like

$$X_1 \ldots [_{np} \ldots [_{rel}\ X_2\ [_s \ldots\ t_2 \ldots\ t_1 \ldots\ ]] \ldots\ ]$$

Note that I here use the concept of left extraposition in a loose sense, without relating it to transformations as in transformational grammar. In XGs, and also in other formalisms for describing languages (cf. for instance the context-free rule schemas of [4]), the notion of trasformation is not used, but a conceptual operation of some kind is required for instance to relate a relative pronoun to a "hole" in the structural representation of the constituent following the pronoun.

3. Limitations of Other Formalisms

Section 5.2 of [5] explains how to handle left extraposition in a DCG, by adding extra arguments to every non-terminal which can possibly dominate a trace. This technique is analogous to the introduction of "derived" rules in [4], and is exemplified by the following simplified grammar for relative clauses:-

```
full_sentence --> sentence(nil).

sentence(Hole0) -->
     noun_phrase(Hole0,Hole1), verb_phrase(Hole1).

noun_phrase(Hole,Hole) --> proper_noun.
noun_phrase(Hole,Hole) -->
     determiner, noun, relative.
noun_phrase(Hole0,Hole) -->
     determiner, noun, prep_phrase(Hole0,Hole).
noun_phrase(trace,nil) --> [].

verb_phrase(Hole) -->
     verb, noun_phrase(Hole,nil).
verb_phrase(nil) --> verb.

relative --> [].
relative -->
     rel_pronoun, sentence(trace).

prep_phrase(Hole0,Hole) -->
     preposition, noun_phrase(Hole0,Hole).
```

Figure 3-1: DCG for relative clauses

The variables 'Hole...' denote either 'trace' or 'nil', depending on whether the trace of an extraposed constituent is expected in the rest of the string. The final rule for 'noun_phrase' recognizes an extraposed 'noun_phrase' as a trace, empty in the surface string, and returns 'nil' denoting the absence of an extraposed constituent. It is obvious that in a more extensive grammar, many non-terminals would need extraposition arguments, and the increased complication would make the grammar larger and less readable.

An alternative way to express left extraposition, available in MGs,

involves the use of rules whose left-hand side is a non-terminal followed by a string of terminal symbols which do not occur in the input vocabulary. An example of such a rule is:-

```
rel_marker, [t] --> rel_pronoun.
```

Its meaning is that a 'rel_pronoun' can be rewritten into a 'rel_marker' followed by the dummy terminal 't', representing a trace. Note that after the application of this rule, the symbol 't' will be at the front of the rest of the input, and subsequent rules will need to cope explicitly with such dummy terminals. This method has been used in several published grammars [2, 3, 7], but in a large grammar it has the same (if not worse) problems of size and clarity as the previous method. It also suffers from a theoretical problem: in general, the language defined by such a grammar will contain extra sentences involving the dummy terminals. For parsing, however, no problem arises, because the input sentences are not supposed to contain dummy terminals. These inadequacies of MGs were the main motivation for the development of XGs.

4. Informal Description of XGs

The only difference between XGs and DCGs concerns what is allowed on the left-hand side of a rule. The left-hand side of an XG rule can be any sequence of segments, where a segment is any sequence of non-terminals and lists of terminals, with the sole restriction that the first symbol of the first segment, the leading symbol, must be a non-terminal. The notation for an XG rule is:-

$s_1 \ldots s_2$ etc. $s_{k-1} \ldots s_k$ --> r. (1)

where the s_i are segments. The following are examples of XG rules:-

```
fronted_verb ... verb(V), [not] --> verb(V), [not].

rel_marker ... trace --> rel_pronoun.

open ... close --> [].
```

Roughly speaking, the meaning of a rule like (1) is that any sequence of symbols of the form

$s_1 X_1 s_2 X_2 \ldots s_{k-1} X_{k-1} s_k$

with arbitrary Xs, can be rewritten into $r X_1 X_2 \ldots X_{k-1}$. This loose description could be made rigorous by using the notion of derivation graph - derivation graphs are for XGs what parse trees are for context-free grammars. In this paper, however, derivation graphs and the meaning of XGs will only be discussed informally.

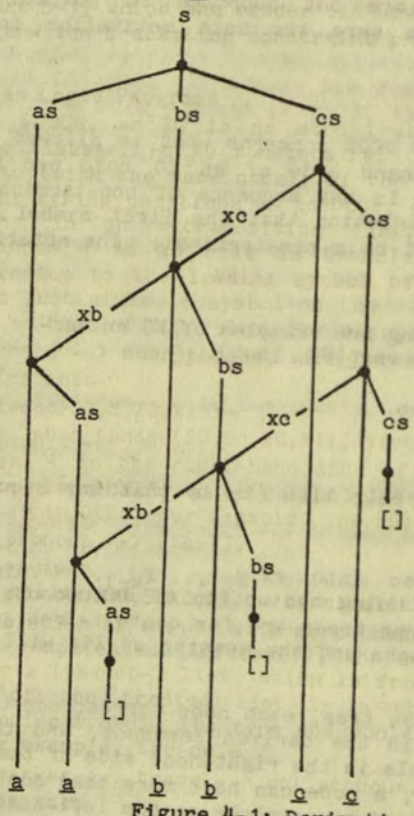
In a derivation graph, as in a parse tree, each node corresponds to a rule application or to a terminal symbol in the derived sentence, and the edges leaving a node correspond to the symbols in the right-hand side of that node's rule. In a derivation graph, however, a node can have more than one incoming edge - in fact, one such edge for each of the symbols on the left-hand side of the rule corresponding to that node. Of these edges, only the the one corresponding to the leading symbol is used to define the left-to-right order of the symbols in the sentence whose derivation is represented by the graph. If one deletes all except the first of the incoming edges to every node from a derivation graph, the result is a tree analogous to a parse tree.

XGs, even without arguments, are strictly more powerful than context-free grammars. For example, figure 4-1 shows the derivation graph for the string "aabbcc" according to the XG:-

```
s --> as, bs, cs.

as --> [].
as ... xb --> [a], as.

bs --> [].
bs ... xc --> xb, [b], bs.

cs --> [].
cs --> xc, [c], cs.
```

This XG defines the context-sensitive language formed by the set of all strings

$a^n b^n c^n$ for $n \geq 0$.



Conventions:

- • = rule application (node)
- x = non terminal
- x = terminal
- [] = empty string

Figure 4-1: Derivation graph for "aabbcc"

Two consecutive symbols in the left-hand side of a rule correspond of course to consecutive edges entering any node for that rule in a derivation graph. If two such symbols are in the same segment, the corresponding edges are said to be next to each other. Requiring edges to be next to each other imposes a restriction on the form of derivation graphs. In a configuration like that of figure 4-2, paths p' and p'' surround a fragment of derivation graph, the sector between them. The terminal string derived from that sector may be non-empty, and then one says that there is a gap between edges e' and e''. However, when edges e' and e'' are next to each other, the terminal string derived from the sector must be empty.

Thinking procedurally, one can say that when a non-terminal is expanded by matching it to the leading symbol on the left-hand side of a rule, the rest of the left-hand side is "put aside" to wait for the derivation of symbols which

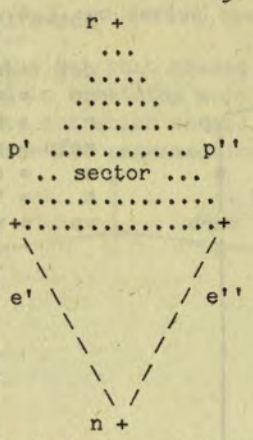


Figure 4-2: Node n has two consecutive edges e' and e''

match each of its symbols in sequence. This sequence of symbols can be interrupted by gaps, which are arbitrary sequences of symbols paired to the occurrences of '...' on the left-hand side of the rule.

I can now show how simple it is to express left extraposition with an XG. The following XG fragment describes essentially the same fragment of language as that of figure 3-1:-

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> proper_noun.
noun_phrase --> determiner, noun, relative.
noun_phrase --> determiner, noun, prep_phrase.
noun_phrase --> trace.

verb_phrase --> verb, noun_phrase.
verb_phrase --> verb.

relative --> [].
relative --> rel_marker, sentence.                    (2)

rel_marker ... trace --> rel_pronoun.

prep_phrase --> preposition, noun_phrase.
```

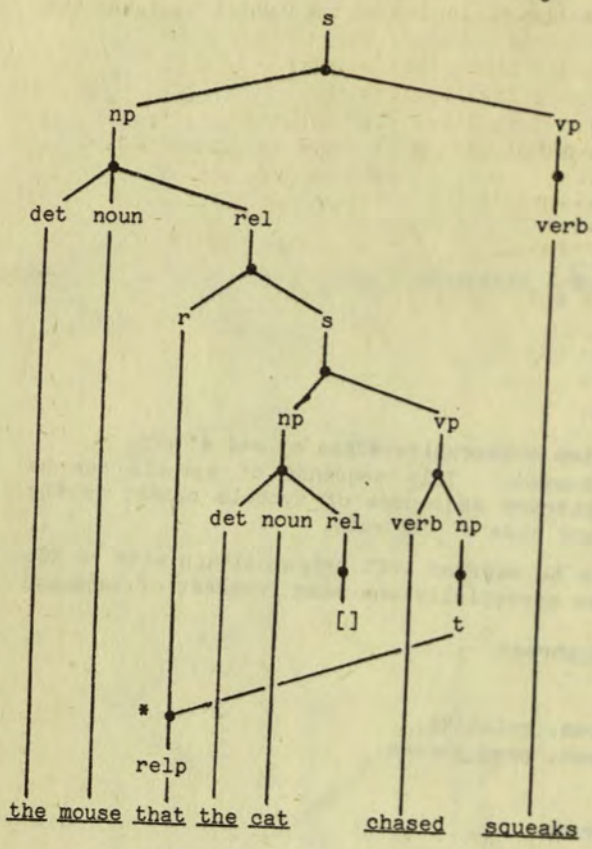
Figure 4-3: XG for relative clauses

In this grammar, the sentence

The mouse that the cat chased squeaks.

has the derivation graph depicted in figure 4-4. The left extraposition implicit in the structure of the sentence is represented in the derivation graph by the application of the rule for 'rel_marker', at the node marked (*) in the figure. One can say that the left extraposition has been "reversed" in the derivation by the use of this rule, which may be looked at as repositioning 'trace' to the right, thus "reversing" the extraposition of the original sentence.

In the rest of this paper, I will often refer to a constituent being repositioned into a fragment of a derivation graph, to mean that a rule having that constituent as a non-leading symbol in the left-hand side has been applied, and the symbol corresponds to some edge in the fragment.



Abbreviations:

det =determiner
np = noun_phrase
r = rel_marker
relp = rel_pronoun
s = sentence
t = trace
vp = verb_phrase

Figure 4-4: Example of derivation graph for the XG above

5. The Bracketing Constraint

In the example of figure 4-4, there is only one application of a non-DCG rule, at the place marked (*). When, however, a derivation contains several applications of such rules, the applications must obey a global constraint, the bracketing constraint of the XG formalism. This constraint is implicit in the translation of XGs into logic programs, to be discussed in the next section. The discussion of the constraint given here is not rigorous, as that would require a lengthy formal discussion of derivation graphs.

In a derivation graph, a node corresponding to the application of a rule with more than one symbol in its left-hand side determines certain sectors in the derivation graph, as defined in the previous section. The bracketing constraint forbids the occurrence in the derivation graph of any node being a descendant both of a node in a sector and of a node outside that sector. That is, no rule can be applied whose left-hand side matches both symbols inside a sector and symbols outside it. However, a rule application may "straddle" a sector, in which case this sector will be a part of a sector induced by the application. The rule applications which create such two sectors are said to be nested.

The constraint and its use is better shown with an example. From the sentences

The mouse squeaks.
The cat likes fish.
The cat chased the mouse.

the grammar of figure 4-3 can derive the following string, which violates the complex NP constraint:-

* The mouse that the cat that chased likes fish squeaks.

This might be rendered in something more like English as:-

The mouse, that the cat which chased it likes fish, squeaks.

The derivation graph for the ungrammatical string is shown in figure 5-1.

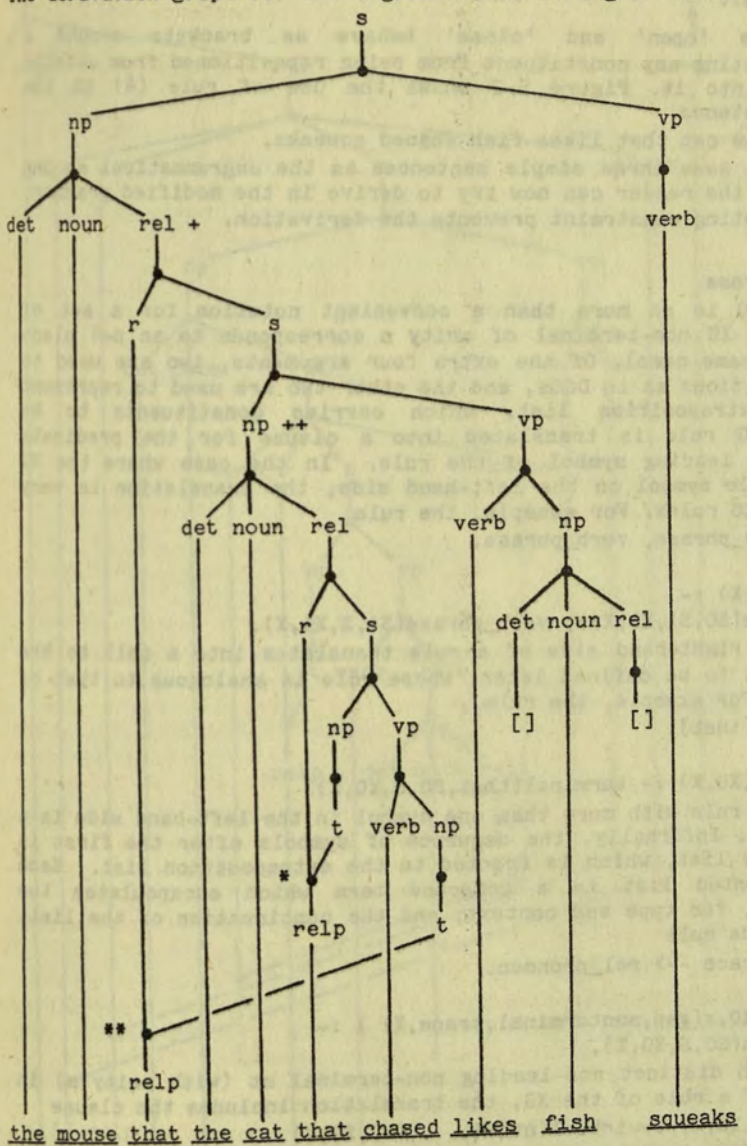


Figure 5-1: Violation of the complex NP constraint

In the graph, (*) and (**) mark two nested applications of the rule for 'rel_marker'. The sentence is ungrammatical because the higher 'relative' (marked (+) in the graph) binds a trace occurring inside a sentence which is part of the subordinated 'noun_phrase' (++).

Now, using the bracketing constraint of XGs one can neatly express the complex NP constraint. It is only necessary to change the second rule for 'relative' in figure 4-3 to (3)

relative --> open, rel_marker, sentence, close.

and add the rule

open ... close --> []. (4)

With this modified grammar, it is no longer possible to violate the complex NP constraint, because no constituent can be repositioned from outside into the sector created by the application of rule (4) to the result of applying the rule for relatives (3).

The non-terminals 'open' and 'close' behave as brackets around a subderivation, preventing any constituent from being repositioned from outside that subderivation into it. Figure 5-2 shows the use of rule (4) in the derivation of the sentence

The mouse that the cat that likes fish chased squeaks.

This is based on the same three simple sentences as the ungrammatical string of figure 5-1, which the reader can now try to derive in the modified grammar, to see how the bracketing constraint prevents the derivation.

6. XGs as Logic Programs

Like a DCG, an XG is no more than a convenient notation for a set of definite clauses. An XG non-terminal of arity n corresponds to an n+4 place predicate (with the same name). Of the extra four arguments, two are used to represent string positions as in DCGs, and the other two are used to represent positions in an extraposition list, which carries constituents to be repositioned. An XG rule is translated into a clause for the predicate corresponding to the leading symbol of the rule. In the case where the XG rule has just a single symbol on the left-hand side, the translation is very similar to that of DCG rules. For example, the rule

```
sentence --> noun_phrase, verb_phrase.
```

translates into

```
sentence(S0,S,X0,X) :-
        noun_phrase(S0,S1,X0,X1), verb_phrase(S1,S,X1,X).
```

A terminal t in the right-hand side of a rule translates into a call to the predicate 'terminal', to be defined later, whose role is analogous to that of 'connects' in DCGs. For example, the rule

```
rel_pronoun --> [that].
```

translates into

```
rel_pronoun(S0,S,X0,X) :- terminal(that,S0,S,X0,X).
```

The translation of a rule with more than one symbol in the left-hand side is a bit more complicated. Informally, the sequence of symbols after the first is made into a (pseudo-) list, which is fronted to the extraposition list. Each element of the fronted list is a compound term which encapsulates the corresponding symbol, its type and context, and the continuation of the list. Thus, for example, the rule

```
rel_marker ... trace --> rel_pronoun.
```

translates into

```
rel_marker(S0,S,X0,x(gap,nonterminal,trace,X) ) :-
        rel_pronoun(S0,S,X0,X).
```

Furthermore, for each distinct non-leading non-terminal nt (with arity n) in the left-hand side of a rule of the XG, the translation includes the clause

```
nt(V1,...,Vn,S,S,X0,X) :-virtual(nt(V1,...,Vn),X0,X).
```

where 'virtual(C,X0,X)', defined later, can be read as "C is the constituent between X0 and X in the extraposition list".

Each element of the extraposition list is a 4-tuple

```
x(context, type, symbol, xlist)
```

where context is either 'gap', if the symbol is preceded by '...', or 'nogap', if the symbol is preceded by ','; type may be 'terminal' or 'nonterminal', with the obvious meaning; symbol is the term which is extraposed; xlist is the remainder of the extraposition list (an empty list being represented by '[]'). So, the rule

```
marker(Var), [the] ... [of,whom], trace(Var) --> [whose].
```

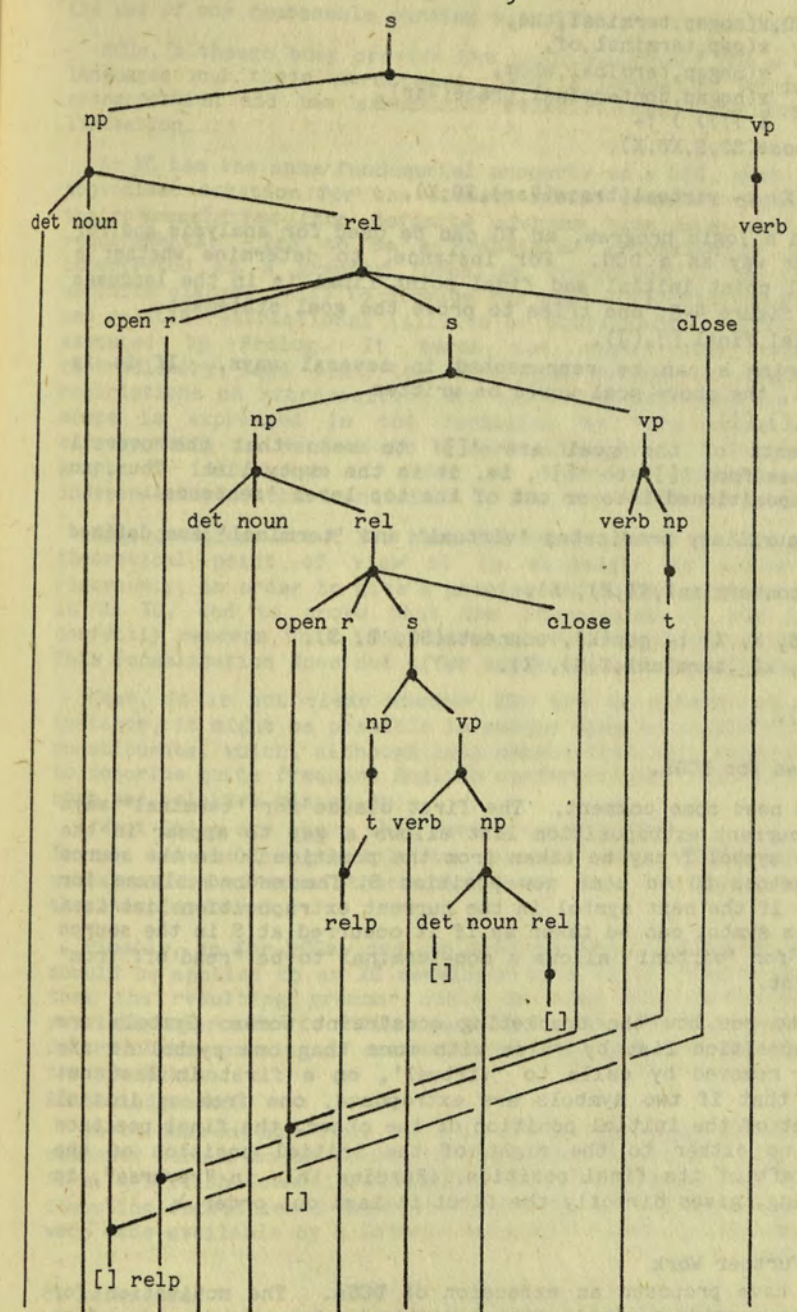


Figure 5-2: Implementation of the complex NP constraint which can be used in a more complex grammar of relative clauses to transform "whose X" into "the X of whom", corresponds to the clauses:-

```
marker(Var,S0,S,X0,x(nogap,terminal,the,
                     x(gap,terminal,of,
                     x(nogap,terminal,whom,
                     x(nogap,nonterminal,trace(Var),
                     X )))) ) :-
        terminal(whose,S0,S,X0,X).

trace(Var,S,S,X0,X) :- virtual(trace(Var),X0,X).
```

Being no more than a logic program, an XG can be used for analysis and for synthesis in the same way as a DCG. For instance, to determine whether a string s with initial point initial and final point final is in the language defined by the XG of figure 4-3, one tries to prove the goal statement

```
:- sentence(initial,final,[],[]).
```

As for DCGs, the string s can be represented in several ways. If it is represented as a list, the above goal would be written

```
:- sentence(s,[],[],[]).
```

The last two arguments of the goal are '[]' to mean that the overall extraposition list goes from '[]' to '[]', ie. it is the empty list. Thus, no constituent can be repositioned into or out of the top level 'sentence'.

Finally, the two auxiliary predicates 'virtual' and 'terminal' are defined as follows:-

```
virtual(NT, x(_,nonterminal,NT,X), X).

terminal(T, S0, S, X, X) :- gap(X), connects(S0, T, S).
terminal(T, S, S, x(_,terminal,T,X), X).

gap(x(gap,_,_,_)).
gap([]).
```

where 'connects' is as for DCGs.

These definitions need some comment. The first clause for 'terminal' says that, provided the current extraposition list allows a gap to appear in the derivation, terminal symbol T may be taken from the position S0 in the source string, where T connects S0 to some new position S. The second clause for 'terminal' says that if the next symbol in the current extraposition list is a terminal T, then this symbol can be taken as if it occurred at S in the source string. The clause for 'virtual' allows a non-terminal to be "read off from" the extraposition list.

Now it is easy to see how the bracketing constraint works. Symbols are placed in the extraposition list by rules with more than one symbol in the left-hand side, and removed by calls to 'virtual', on a first in last out basis. This means that if two symbols are extraposed, one from an initial position to the right of the initial position of the other, the final position of the first must be either to the right of the initial position of the second, or to the left of its final position. (Reading this in "reverse", in terms of repositioning, gives directly the first in last out order.)

7. Conclusions and Further Work

In this paper I have proposed an extension of DCGs. The motivation for this extension was to provide a simple formal device to describe the structure of such important natural language constructions as relative clauses and interrogative sentences. In transformational grammar, these constructions have usually been analysed in terms of left extraposition, together with global constraints, such as the complex NP constraint, which restrict the range of the extraposition. Global constraints are not explicit in the grammar rules, but are given externally to be enforced across rule applications. This is objectionable, both on theoretical grounds, because it is an "ad hoc" device, and on practical grounds, because it leads to obscure grammars and prevents

the use of any reasonable parsing algorithm.

DCGs, although they provide the basic machinery for a clear description of languages and their structures, lack a mechanism to describe simply left extraposition and the associated restrictions. XGs are an answer to this limitation.

An XG has the same fundamental property as a DCG, that it is no more than a convenient notation for the clauses of an ordinary logic program. XGs and their translation into definite clauses have been designed to meet three requirements: (i) to be a principled extension of DCGs, which can be interpreted as a grammar formalism independently of its translation into definite clauses; (ii) to provide for simple description of left extraposition and related restrictions; (iii) to be comparable in efficiency with DCGs when executed by Prolog. It turns out that these requirements are not contradictory, and that the resulting design is extremely simple. The restrictions on extraposition are naturally expressed in terms of scope, and scope is expressed in the formalism by "bracketing out" subderivations corresponding to sectors in a derivation graph. The notion of derivation graph is introduced in order to describe extraposition and bracketing independently of the translation of XGs into logic programs.

Some questions about XGs have not been tackled in this paper. First, from a theoretical point of view it is necessary to define derivation graphs rigorously, in order to give a precise definition of the concept of derivation in an XG, and to prove that the translation of XGs into logic programs correctly renders this independent characterisation of the semantics of XGs. This formalisation does not offer any substantial problems.

Next, it is not clear whether XGs are as general as they could be. For instance, it might be possible to extend them to handle right extraposition of constituents, which, although less common than left extraposition, can be used to describe quite frequent English constructions, such as the gap between head noun and relative clause in:-

What files are there that were created today?

It may however be possible to describe such situations in terms of left extraposition of some other constituent (eg. the verb phrase "are there" in the example above).

Finally, as for DCGs, one can consider the question of what transformations should be applied to an XG developed as a clear description of a language, so that the resulting grammar could be used more efficiently in parsing. A possible approach to this question might be to generalise results on deterministic parsing of context-free languages into appropriate principles of transformation.

Acknowledgements

David Warren has read several drafts of this paper, and his detailed comments were a major source of improvements, both to the content and to the form. A British Council Fellowship is supporting my work in this subject. The computing facilities I used to experiment with XGs and to prepare this paper were made available by a Science Research Council grant (GR/A 74432).

8. References

1. Chomsky, N. Reflections on Language. Pantheon, 1975.

2. Colmerauer, A. Metamorphosis Grammars. In L .Bolc, Ed., Natural Language Communication with Computers, Springer-Verlag, 1978. First appeared as an internal report, 'Les Grammaires de Metamorphose', in November 1975

3. Dahl, V. Un Systeme Deductif d'Interrogation de Banques de Donnees en Espagnol. Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1977.

4. Gazdar, G. English as a Context-Free Language. School of Social Sciences, University of Sussex, April, 1979.

5. Pereira, F. and Warren, D. H. D. Definite Clause Grammars Compared with Augmented Transition Networks. Research Report 58, Dept. of A. I., University of Edinburgh, October, 1978.

6. Pereira, L. M., Pereira, F. and Warren, D. H. D. User's Guide to DECsystem-10 Prolog. Dept. of A. I., University of Edinburgh, 1978.

7. Pique, J. F. Interrogation en Francais d'une Base de Donnee Relationnelle. Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1978.

8. Ross, J. R. Excerpts from 'Constraints on Variables in Syntax'. In G. Harman, Ed., On Noam Chomsky: Critical Essays, Anchor Books, 1974.

9. Roussel, P. Prolog : Manuel de Reference et Utilisation. Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, 1975.

Feliks Kluźniak, Stanisław Szpakowicz
Institute of Informatics
Warsaw University
POB 1210, 00-901 Warszawa, POLAND

April 1980

A Note on Teaching Prolog

Prolog can be taught either to a layman or to an experienced programmer. The former will gladly accept any plausible justification of the way the language is constructed - first-order logic seems the best justification possible. The latter, however, will be demoralized by bad Fortran habits or, at best, so used to the Structured-Programming-in-Pascal approach that he will need a lot of convincing before he considers Prolog a useful programming language.

We are going to outline the crucial points of a method of presenting Prolog fundamentals which gives due attention to their proximity to "normal" programming language constructions and emphasizes their advantages.

We are deemphasizing logic intuitions as we believe that any practical use of Prolog requires drastic short cuts on the way from a logic explication to a running program. One can exploit first-order parallels of Prolog clauses to facilitate program verification. Normally, though, a clause is much more concise than its logic counterpart and it is even more so with complete procedures. Try, for example, to account reasonably for such (evidently not first-order!) features as the cut procedure (the "slash"), variable literals and - last, not least - the conscious use of backtracking as a programming tool rather than as a search space generator.

We also do without AI intuitions which do not help to make the clarity of Prolog obvious enough. We stress Prolog's uniformity and generality of data and control structures. These are the very features of a good programming language, not only a language

"for AI".

As a contrast to Prolog we have chosen not Lisp but Pascal which is clearly the most up-to-date of all the widespread languages. Its design was influenced by the recommendations of Programming Methodology, the same that we use to show the main (and somewhat startling) features of Prolog in a favourable light.

* * *

In a very general sense terms can be thought of as specifications of data types, that is of classes of objects. These vary from the most general (the class of all objects) to the most specific ones (a class containing one object). In Prolog objects are denoted directly by descriptions of their types - this provides the sufficient flexibility to let us specify only those attributes of objects which are interesting in a particular context.

A simple (unstructured) object is denoted by a name and we do not associate with it any interpretation. Simple objects are the Prolog counterparts of Pascal's constants but without some of their flaws. Simple objects need not be declared, neither do they require a specification of type-constant relation which is obligatory in Pascal.

A compound object is built of components which can be quite arbitrary. Their connection with the object is expressed by its name (which is, in fact, a close counterpart of type name in Pascal). A term denoting an object is $name(c_1, \ldots, c_n)$ with c_i - a description of i-th component which can be also a compound object. We usually speak of functor and arguments instead of "name" and "components", as the notation resembles the one used in mathematics. For the sake of homogeneity of approach we also speak of functors of zero arguments which describe simple objects.

Some syntactic sugar is welcome to facilitate description of objects but it must not make things obscure. We introduce fix func-

tors (infix, prefix and postfix) which help to simplify the notation (we think it confusing to call them "operators"). Other facilities, such as infix procedure names or list brackets, aren't in the least necessary.

Terms containing variables are used to describe objects with incompletely specified attributes; a variable denotes an object of an unknown type. Partially unspecified objects are used in a manner similar to Pascal's formal value-parameters in that they serve as the handles of concrete objects. There is ~~the~~ a close similarity to "variables" of high-school algebra, cf $g(x)=x^3+3$. After a Prolog variable gets instantiated, it likewise ceases to be a variable: later on it denotes a specific object. That object in turn can contain variables - quite naturally it means that it still remains only partially specified.

Terms are certainly more general than Pascal's value-parameters. Their instantiation can be performed in discrete steps and not necessarily at once; it can also take place both on the called and the calling side. This generality is due to the generality of unification used as the parameter passing mechanism. This particular mechanism enables us to write procedures in a natural and concise form, as illustrated by

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

The operation of a procedure depends on the attributes of its actual parameters but it is usually (as in Pascal) obscured by the piecemeal fashion of testing attributes to determine control flow. The dependency becomes clear in a multi-clause procedure where every clause gives rise to a distinct course of computation. Surprisingly enough, similar multi-use routines appear in so modern a language as ADA in the clumsy form of "overloaded" procedures; each version of such a procedure handles another set of parameter types.

The types of the actual parameters should match those of the formal ones in at least one clause heading of the called procedure. Otherwise the call is erroneous. Treating failure as an error condition helps to explain the meaning of backtracking. Experience shows that this notion can be particularly alien to a programmer's intuitions. We decided then to slip through the psychological barrier by paraphrasing it as a general exception-handling mechanism with one major peculiarity. In Prolog every procedure can - in suitable circumstances - serve as an exception-handler and its operation consists in redirecting the computation in order to avoid failure.

The misuse of backtracking for extended type-checking (e.g. at run time) and for the construction of nondeterministic procedures is then shown to equip Prolog programs with so much power, elegance and clarity that it cannot be advised against, even though it makes backtracking virtually useless in its principal application, namely exception handling. Fortunately we need not really make a choice since the cut procedure appears to be an effective tool for taming nondeterminism and for tailoring the backtracking mechanism to our needs.

* * *

This short paper was not intended as a detailed description of our method. A complete lecture on Prolog will appear - in Polish - in a book being written by J.Bień and the authors. We only wanted to suggest a way of demonstrating that Prolog is not as exotic as it has seemed up to now. If Prolog is to be generally regarded as a programming language in its own right, we must try to make it comprehensible to the programming community at large. To make it comprehensible we must, however, explain terms, unification, multi-clause procedures, backtracking and the cut by the notions relevant to a programmer's way of thinking.

\- - -

Peter TANCIG, Damjan BOJADŽIEV

University of Edvard Kardelj in Ljubljana
JOSEF STEFAN INSTITUTE
Department of Computer Science and Informatics
Ljubljana, Yugoslavia

SOVA - An Integrated Question-Answering System Based on ATN (for syntax) and PROLOG (for semantics) in LISP Environment

Extended abstract of the paper intended for
COLING-80, the 8-th International Conference
on Computational Linguistics,
TOKYO, Sept. 30 - Oct. 4., 1980

Subject Headings:

integrated natural language question-answering systems
ATN subsystem for syntax/semantic analysis
PROLOG subsystem for semantic interpretation, assimilation,
and search in data base
interfaces among different parts of the system
LISP environment
CDC CYBER computer

In the paper we present a methodology and an application of 2 powerful tools, implemented and embedded in LISP, for experimenting with question-answering systems with natural language input.

Parsing of input sentences is done with an ATN subsystem which allows one to experiment with different syntax/semantics-oriented approaches and to produce different kinds of "deep" structures.

These "deep" structures may already be the PROLOG clausal form or they may be "translated" into it from another "deep" structure rep-

The following function composition:

```
(setq OUTSNTC
  (generate (setq VALUES
              (interprete (setq INPCLAUSE
                            (clausalize (setq INPDEEP
                                          (parse (setq INPSNTC
                                                   (read]
```

illustrates the major 5 steps (functional parts; subsystems) in processing a natural language input sentence with the SOVA QAS.

The 5 (global) variables and their values are:

INPSNTC <-- input sentence (question; assertion) in natural language

INPDEEP <-- "deep" structure of INPSNTC as a result of ATN parsing

INPCLAUSE <-- PROLOG input clausal form of INPDEEP (i.e. of INPSNTC)

VALUES <-- PROLOG's answer after evaluation of INPCLAUSE

OUTSNTC <-- output sentence (answer) in natural language.

One way of dealing with the transfer of the good old Fido would be the following:

assertion

```
INPSNTC <-- (Peter gives Fido to Mary)
INPDEEP <-- (S     DCL
               (NP     (NPR     Peter))
               (VP     (V     Gives)
                       (NP     (NPR Fido))
                       (PP/     (PREP  to)
                                (NP     (NPR   Mary)))))
INPCLAUSE <-- (LET (EVENT "PETER "GIVES "FIDO "MARY))
VALUES <-- (EVENT)
OUTSNTC <-- O.K.
```

question

```
INPSNTC <-- (What does Peter give to Mary)
INPDEEP <-- (S     Q
               (NP     (NPR     Peter))
               (VP     (V     Gives)
                       (NP     ?)
                       (PP/     (PREP  to)
                                (NP     (NPR   Mary)))))
INPCLAUSE <-- (GOALS (EVENT "PETER "GIVES OBJ "MARY))
VALUES <-- ((OBJ) (FIDO))
OUTSNTC <-- Peter gives Fido to Mary
```

APPLICATION OF PROLOG IN DESIGNING MANY-STORIED DWELLING HOUSES

BY ZSUZSANNA MÁRKUSZ

Computer and Automation Institute
Hungarian Academy of Sciences
/1111 Budapest, Kende u. 13-17./
Hungary

1. PREFACE

The purpose of this paper is to show a new application of PROLOG in the field of CAD. A previous architectural CAD application of PROLOG has been published in [Mar, 3]. In the course of solving the problem laid out there - namely the designing of different ground-plans for big-panel apartments - it turned out that PROLOG can be well applied in solving architectural problems. This program system makes use of the experiments performed in the previous work, however it is based on some other architectural conceptions. Furthermore this new program system does not stop at producing variants of apartment ground plans, but aims at contributing to architectural design of complete dwelling houses. The method of program developing appears to be new as well. It will be reported in Chapter 4, and it was also published in greater detail in [KM, 4] and [KM, 5] . The results of its application come up in this paper, too.

The programs were tested and run on the Siemens 7.7.55 computer of the Institute for Coordination of Computer Technics, the PROLOG system which was installed by Peter Köves [Köv, 6].

2. THE DESIGN SYSTEM OF THE APARTMENT HOUSE, DATA BASE, STRUCTURES

In this program system basic building cells with given measurements and functions serve as the basic elements for designing apartments. Three cells with different measurements are taken into the data base:

measurements		name
2.4 x 4.8 meter	/narrow/	A,B,C,D
4.2 x 4.8	/wide/	M
4.2 x 3.6	/wide/	N,L

Each apartment is constructed by joining 3 wide cells and 1 to 4 narrow cells. A general apartment is made up of a maximum of 7 cells has the following geometrical measurements:

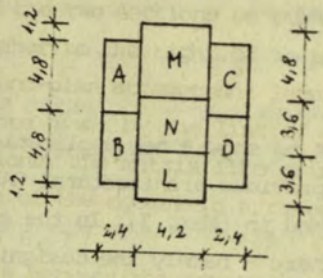


Figure 1. Outline of an apartment

Depending on how large the apartment is some of the narrow cells /A,B,C,D/ can be omitted. The functions and codes of the cells are as follows:

living room	1	dining room	2
bedroom	3	entrance	4
kitchen	5	bathroom	6

One cell can serve two functions. According to the number and the position of the doors, cells with the same measurements and functions can stand for different versions. The number of versions is the last digit of the cells codes contained in the data base. In the present program system there are as many as 18 cells with different codes involved /Fig.2./ There are two digit and three digit codes in accordance with the functions of the cells. The last digit always points to the number of versions, while the first or the first two digits stand for the above functional codes.

In the program one apartment is represented by a list as follow:

*M. *N. *L; *A. *B. *C. *D

or *MIDDLE; *OUTSIDE

where MIDDLE means the list of the middle wide cells,

OUTSIDE means the list of the outside narrow cells.

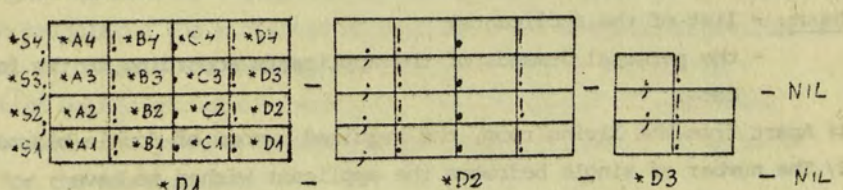
In the course of apartment design the constant values of the chosen cells are assigned to the variables of the list. If the apartment contains less than 7 cells, a variable still remains in the unused part of the list.

Two apartments and a staircase make up one section level. Horizontal placement of one or two section levels makes one dilatation unit level. The vertical placement of 2-4 section levels makes one section, while the vertical placement of 2-4 dilatation unit levels makes one dilatation unit. Any number of dilatation units joined horizontally make up one building. One building is represented by a compound list in which four different operators are present.

According to the priority level the operators are as follows:

- ! joins apartments /horizontally/
- . joins section levels /horizontally/
- ; joins dilatation levels /vertically/
- - joins dilatation units /horizontally/

To set an example let us take a building with 4 levels and 3 dilatation units:



where *D1, *D2, *D3 represent the certain dilatation units,

*D1 breaks down according to levels:

*D1 = *S4; *S3; *S2; *S1; NIL

*S1 breaks down according to apartments:

*S1 = *A1 ! *B1 . *C1 ! *D1 . NIL

Figure 3. An example for illustrating a building in the program

The building is designed so that the list of contants of the finished apartments will replace the variables /*A1,.../ representing the apartments.

3. *PARTS OF THE PROGRAM SYSTEM*

The program system is composed of the following four programs:

1. FLAT program to design variations of ground.plans to meet individual demands;
2. ORD program to decide the priority ordering of variations of apartments;
3. BUILD program to create the vertical structure of the building to be designed;
4. TOTAL program to design the arrangement of ground-plans of dwelling houses level by level.

3.1. FLAT

Taking into consideration the applicant's special demands, this program produces all the possible ground-plans of the apartment, then it provides these variants with the code of the applicant and stores them in a file. This program is to be run as many times as the number of applicants.

Input: - list of the applicants;
- the personal demands of the applicants according to the following data:

1/ Apart from the living room, the required number of double bedrooms /0-3/
2/ The number of single bedrooms the applicant wishes to have
3/ Preference of functional combinations /one of the following/:
/1/ kitchen - dining room
/2/ living room - dining room
/3/ living room - dining room - kitchen
/The possible answers are 1 or 2 or 3./
4/ Second choice /1,2,3 or 0 if no second choice/.
5/ Does not accept the third solution? /1,2,3 or 0/.
6/ Does he want a double bedroom for parents? /Yes, No/.
7/ Does he agree to having a double bedroom opening into the living room? /Yes, No/.
8/ Does he want a study? /Yes, No/.

Output: - the lists of ground-plans fulfilling the input requirements /in a file/

- on the printer the same lists in the form of a matrix with data concerning area and other quantities:

1/ the area of the whole apartment

2/ living area

3/ number of beds

4/ the ratio of living area and the whole area

5/ area for one person

3.2. ORD

This program enables the applicant to order the apartments according to preference that is the from the best choice to satisfactory in such a way that the applicant may exclude those versions which do not fulfill his needs. This is done by assigning a priority number to the apartments on the list.

Input: the output of the FLAT program

Output: according to the applicant's demand the program orders the apartments in the file.

3.3 BUILD

As a function of the maximum number of levels of the building to be designed as well as a function of the number of apartments, this program calculates the number of dilatation units the building will contain. It designs the general and the last - maybe incomplete - dilatation unit's structure /sections, levels, number of apartments/. It elaborates the general outline of the building which will serve as input for the next program.

Input: - the number of apartments in the building to be designed;

- maximum number of levels;

- number of sections is one dilatation unit;

Output: the general outline of the whole building printed.

3.4. TOTAL

This program makes use of the results obtained from the precious three. Its aim is to fill the structure made by BUILD and several runs of the FLAT program taking into account the area and geometric requirements of the building. The program places the apartments beside and above one another so that it selects those which correspond to the horizontal and vertical geometric and functional connections.

- 6 -

Input: - the type of the left end of the dilatation unit /open or closed/;
- the type of the right end of the dilatation unit;
- the connection of the first apartment to the second in the first staircase is to be shifted up or down by half level?
- the same concerning the second staircase;
- how many narrow cells should there be at the joint of the first and second apartments?
- the same with the 2nd and 3rd apartments;
- the same with the 3rd and 4th apartments;
- should the two sections be pushed horizontally as compared to each other to the right or to the left?

Output: the ground-plan arrangement level by level of the whole building. the apartments are given by the list of cell codes and the serial number of the applications. The matrix like printing of the code numbers provides a visual picture of the egometric arrangement of functional elements and apartments /Fig. 6, 7 /.

The relations between the apartments inside the building are presented on Fig .5:

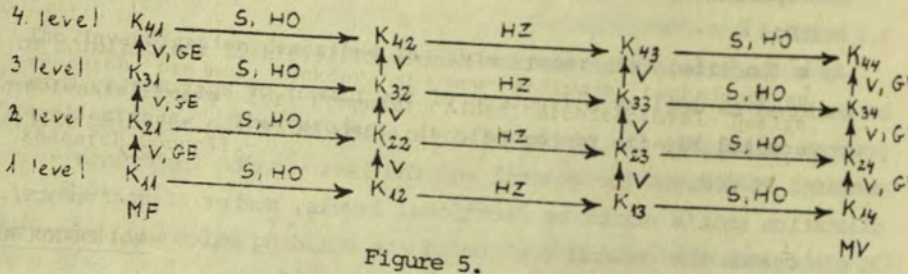


Figure 5.

where K nm /1≤ n, m≤ 4/ is an apartment;

K nm $\xrightarrow{P,Q}$ K n m+1 - the horizontal joining of two apartments. To the apartment K nm we select a K n m+1 apartment which meets requirements P,Q.

K n+1 m
↑ R,S - the vertical joining of two apartments. To K nm we slect a K n+1 m apartment which meets requirements R and S.
K nm

Abbreviations:

S - symmetry
HZ - horizontal
HO - horstair
MF - suitable left

MV - suitable right
V - vertical
GE - geometric left
GV - geometric right

To every connection a procedure is assigned the functions of which are as follows:

1. SYMMETRY

It produces the symmetrical version of an apartment designed by the FLAT program with respect to the Y axis.

2. HORSTAIR

It controls whether the staircase can be put between two apartments in the way that the entrances open from the staircase.

3. HORIZONT

It checks whether the geometric structure of the right cells in the second apartment is suitable to be joined to the left cells of the third apartment.

4. SUITABLELEFT

This program checks whether the right cells of the first apartment on the first level which is situated on the left side of the dilatation unit corresponds to the type of the dilatation unit on the left. If this type is OPEN then it does not join another dilatation unit, therefore the outline of the elements can be arbitrary. If it is CLOSED, the left side of the apartment's outline is allowed to contain only flat surfaces.

5. SUITABLERIGHT

Similar to 4. on the right end of the dilatation unit.

6. VERTICAL

This procedure is necessary for selecting any apartments above the ground floor. It checks whether

- the water-block of the chosen apartment is above the water-block of the lower apartment,
- the entrances are above each other,
- there are cells under all cells.

7. GEOMETRICLEFT

This procedure is necessary to choose apartments on the left side of the dilatation units above the ground floor. It checks whether the left element of the upper apartment corresponds to the type of the left side

of the dilatation unit, and whether the geometry of the upper and lower apartments match.

8. GEOMETRICRIGHT

The same as 7 but refers to the right side apartments.

Filling up the structure of the dilatation unit with apartments is performed in the following way:

- applying the SUITABLELEFT condition we choose the apartments K 11 to the left side of the first level,
- we fill up the first level with apartments in order of K 12, K 13, K 14 taking into account the horizontal conditions pointed out in Fig. 5,
- we choose apartment K 21 to the left side of the second level so that the respective vertical conditions are applied to the lower apartment K 11,
- the conditions of fitting the next /K 22/ apartment are determined by apartment K 21 beside an K 12 below it. This system of horizontal and vertical conditions can be seen in Fig. 5. The other apartments /K 23, K 24/ of the second level fit in the same way,
- the 3rd and 4th floors are treated similarly to the 2nd.

Fig.6. shows the output form of the first level from the second dilatation unit of the same building. Fig.7. is the respective drawing of the above.

4. *THE COMPLEXITY OF THE PROGRAM SYSTEM*

In the course of developing the system the theory referring to the complexity of PROLOG programs outlined in KM, 5 was taken into account. This theory aims at reducing the semantical program errors as well as at the cutting of test time through the introduction of local and global complexity of the programs.

Now we shall not go into details of the theory, just observe the basic principles and the achieved results.

All the programs have been developed in an easy-to-survey hierarchical organization so that the parts of the programs can be written almost independently from one another. PROLOG especially is suitable for that,

The independent program units are called partitions. The definitions of the 6 different sorts of partitions /AND, OR, DATA, CASE, RECURSION, TASK/ can be found in[KM, 5.].The programs were written 'top-down' and we decided at the hierarchical decomposition every time what sort of partition we needed for the further devision of the task. The partition was defined so that the number of clauses and arguments should not exceed 4. Having written the program before running it, we calculated the local and average complexity level of the program. The partitions with high complexity were devided to simple ones and the program was altered accordingly. Table 1. contains the test data of the programs, the number of sematic errors and complexity indeces. We found the average local complexity 6.7 calculated for the whole program system to be optimal.

The number of sematic errors is 34 which is fairly small in respect to the whole program system. We can say that complexity calculation contributed a great deal in writing this fairly difficult task and putting it into working condition with only a few semantic errors in a short time, It took two persons a month's work to write and test the program system.

The computer aided design procedure laid out in this paper can be adopted to other architectural applications. To create a new design system, the redefinition of the building's structure and the supporting data base is necessary. The program level formalization of steps of the design solved by this system can be further applied.

For the time being the program system only serves our research goals however, by altering the data base, the system can be made suitable for any particular application.

5. ACKNOWLEDGEMENTS

I would like to express my thanks architect Mr. István Rákossy for working out the architectural concepts in this program system and for helping set the architectural way of thinking into computer programs.

References

[Kow, 1] : Kowalski, R.: Predicate Logic as a Programming Language. Proceedings of IFIP, 1974

[SzF, 2] : Szeredi, P; I. Futó: PROLOG MANUEL, Számológép, 7. 3-4.sz.1977. in Hungarian.

[Mar, 3] : Márkusz, Zs: How to design variants of flats using programming language Prolog, based on mathematical logic. Proc. of IFIP, 1977. Toronto, Canada.

[KM, 4] : Kaposi, A.A; Zs.Márkusz: Primlog: A case of augmented Prolog-programming. Proc. of Informatica'79, Bled, Yugoslavia.

[KM, 5] : Kaposi, A.A; Márkusz, Zs: Introduction of a complexity measure for control of design errors in logic-based CAD programs. Proc. of CAD'8o Brighton, Great Britain.

[Köv, 6] : Köves, P: BS 2ooo PROLOG USERS'S MANUEL, V2.4. SzKI, 1978. in Hung.

[MR, 7] : Márkusz, ZS; Rákossy, I: CAD System for Many-Storied Apartment Houses. Manuscript. 1979.

Name of the program	Design /man-days/	Testing /man-days/	Number of partitions	Global complexity
FLAT	lo	11	78	5o2
BUILD	3	3	39	266
TOTAL	8	16	85	6o8
ORD	1	1	13	76
	22	31	215	1452

	Number of semantic errors	Number of different types of partitions T	S	C	Average local complexity
FLAT	11	14	43	21	6.4
BUILD	4	8	16	15	6.8
TOTAL	18	16	31	37	7.1
ORD	1	5	4	4	5.8
	34	43	94	77	

The average local complexity of the program system:

6.7

Table 1. Summary

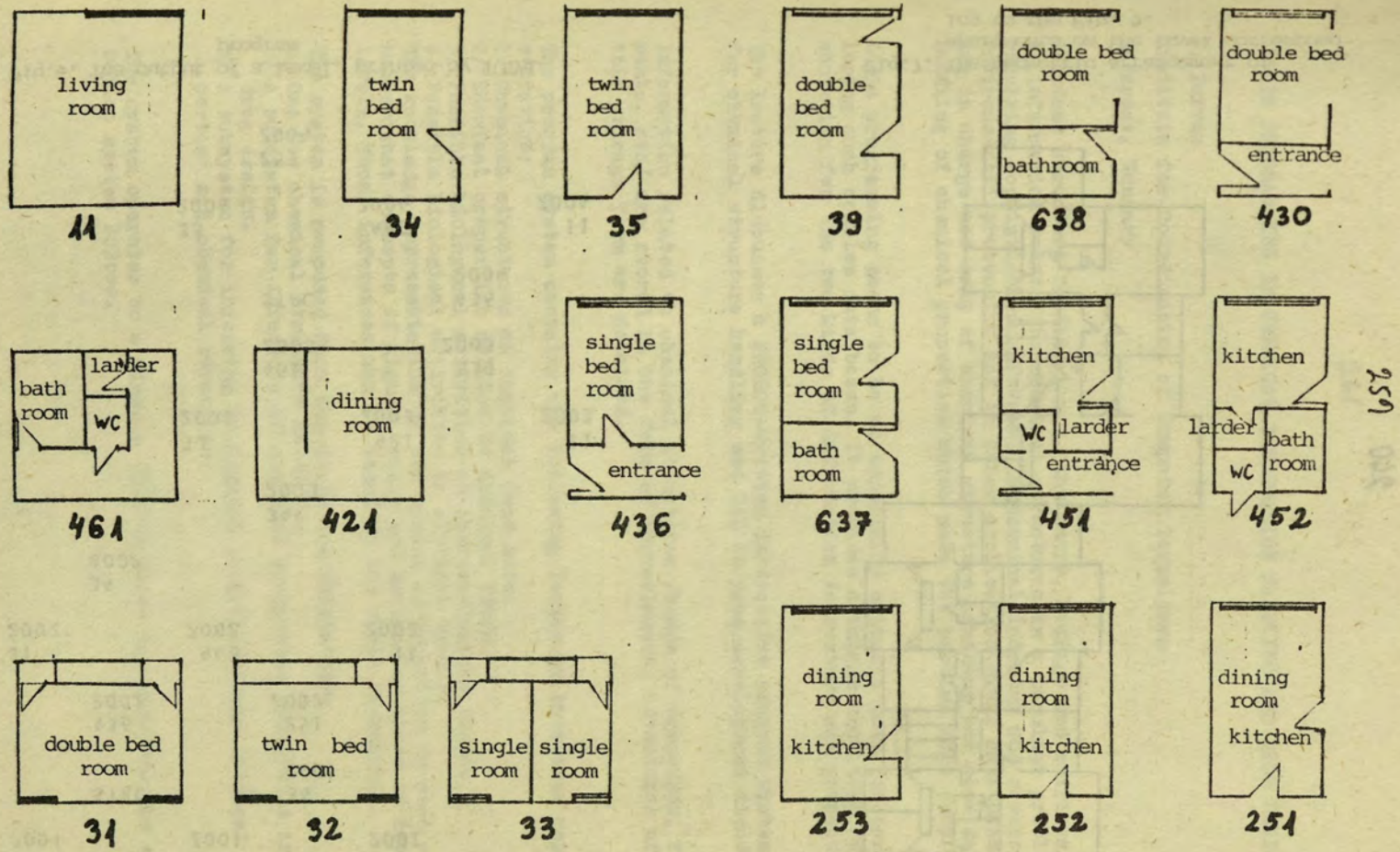


Figure 2. Data base for the FLAT program

252
2001

39
2001

33
2001

461
2001

11
2001

STRC

34
2001

436
2002

251
2002

31
2002

462
2002

11
2002

34
2002

34
2003

33
2003

421
2003

11
2003

451
2003

638
2003

STRC

436
2004

33
2004

462
2004

11
2004

253
2004

Fig.6. The output of a level, printed by TOTAL program

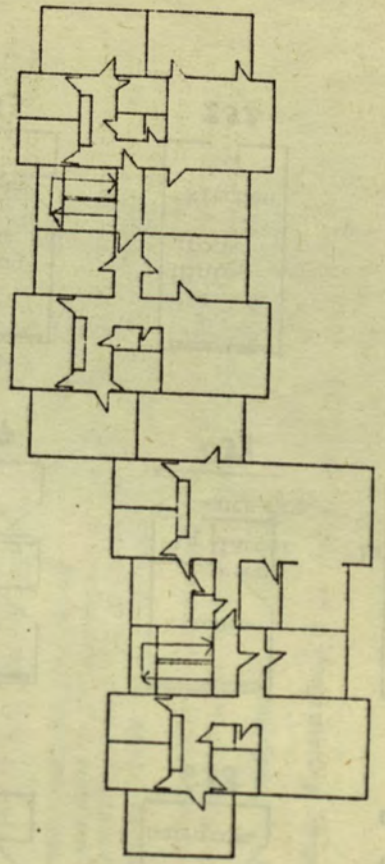


Fig.7. The geometric arrangement of apartments on the level corresponding to the Fig. 9.

LOGIC PROGRAMMING IN CHEMICAL INFORMATION HANDLING AND DRUG DESIGN

F. Darvas

Institute for Coordination of Computer Techniques

Budapest, Hungary

Chemical structure handling, i.e. the entry, storage and retrieval of information related to chemical structures is a peculiar problem requiring sophisticated software. In chemical industry and research, environmental protection and in other application areas data bases are in charge not only of chemical structure handling but also data handling of chemical properties which make the problem more complex.

Logic programming seems to be an especially suitable tool for developing such complex data bases. It provides a unique knowledge representation for the two kinds of information /structure and properties/.

The lecture discusses a PROLOG-written interactive program system for chemical structure handling and CAD in drug development field.

Information related to chemical structures /bonds of compounds, fragments, etc/ are stored in the form of Horn-clauses. Advantages of this storage form are detailed.

The program system contains the following levels of knowledge representation:
1/ Chemical structures as chemical bond sets.
2/ Chemical properties related to chemical bonds.
3/ Possible biological activities in pharmacological tests.
4/ Possible biological activities in clinical trials.
The knowledge representation of the system is organized in such a manner that concepts of higher levels are defined by those at lower levels. Thus, inferences can be based on the whole hyerarchy.

The system is composed from the following subsystems:
a/ One for chemical structure handling.
b/ A subsystem for inference of chemical properties applicable in drug design.
c/ A subsystem for inferring biological activities from chemical properties and chemical structure.

The system operates on a Siemens 7755 computer and requires the operation system BS2000.

PRESCRIPTIVE TO DESCRIPTIVE PROGRAMMING
A Way Ahead for CAAD

Peter S. G. Swinson
July 1980

EdCAAD Studies
Department of Architecture
University of Edinburgh

ABSTRACT

Fairly extravagant claims have been made by PROLOG enthusiasts about the ease of programming and ease of understanding the finished code by people other than the program originator. To test these claims, an apparently simple problem was set which, although somewhat abstract in nature, illustrates some of the problems associated with Computer Aided Architectural Design programming. The problem was then tackled using the FORTRAN and C computer languages involving what can be described as prescriptive programming, and in PROLOG involving descriptive programming. The following report describes the difficulties of CAAD programming, the progress of the exercise, the unexpected ultimate version of the PROLOG program and the conclusions drawn from the exercise.

The investigation described in this paper is part of a research project looking at new software techniques as potential tools for Computer Aided Architectural Design applications. The project is funded by the UK Science Research Council.

1. INTRODUCTION

1.1. Computing in Architecture Today

Computing has now become widely used in many disciplines. In architecture, however, there has not yet been a large scale impact. Apart from programs to tackle specific evaluative operations - primarily for specialist consultants to architects - there is little for building designers. There are very few large integrated design systems around, and even fewer in use. This is despite the efforts of several research groups working in the field of Computer Aided Architectural Design (CAAD) over the past decade. An integrated CAAD system is one in which the complete description of a building is stored in a comprehensive data base, and where a wide range of interrelated operations may be performed to generate further design and production information.

The experience of the Scottish Special Housing Association (SSHA) serves to illustrate some of the major difficulties. It needs to be said that the following comments are in no way critical of the SSHA who were prepared to be involved as pioneers in CAAD. Rather, the experience has highlighted the inherent problems of current integrated CAAD systems in general. The SSHA is a central government funded organisation set up to build and factor public housing throughout Scotland, supplementing the work of local authorities. In co-operation with the EdCAAD group of the University of Edinburgh an integrated CAAD system was developed to produce the contract documentation (drawings and bills of quantities) for housing schemes (1). Most of this has been in production use for about four years.

The system has highlighted three difficulties:
- The time taken to get the system into production was much longer than expected.
- Despite interest from several outside bodies, no other organisation has implemented the system or any part of it.
- The system is becoming harder and harder to adapt to the changing needs of the SSHA.

These difficulties stem from three basic aspects of CAAD today, namely the nature of CAAD problems, the tools currently available to resolve these problems, and the compound effect of these two taken together.

1.2. The Nature of CAAD Problems

In common with many other computing applications, CAAD systems can be seen as ways of helping people to cope with the increasing volume of information involved in their work. (Whether all this information is necessary may be debatable, but that is not the concern here.) Unlike most other computing systems, however, design activity is dominant in integrated CAAD systems. The creative and subjective nature of design means that there are no "right" or determinate answers to design problems. The needs of one designer are not often met by the tools which meet the needs of another designer, or even the same designer in a different situation or occasion. The more sophisticated the system, the more this is so.

Not only individuals but organisations approach design differently from each other. To a system designer it may well be an attractive idea tc have a system reused in many places but, even if technical matters at each installation could be resolved, it would not be desirable that individuals or organisations should have to tailor their their way of working to conform to a given system.

The same problem of differing views on a common data base occurs within any one implementation of a system within an organisation. The same information is used by architect, engineer, quantity surveyor and site operative. As well as differing views among these people, motivated by their different tasks, the same information is used by any one person in different ways. It is not a simple matter to resolve the various views and categorisations of the common data.

1.3. The Tools Currently Available

Advances in hardware technology (the machines) has outstripped advances in software technology (the programs). Conventional programming techniques in use for CAAD can be described as imperative or prescriptive. Every step that the machine has to take has to be spelled out in stupefying detail, every operation that may be called for needs to have been anticipated and catered for in full. The programs become larger and larger. Additions become more and more difficult to integrate satisfactorily into the existing code, whilst ensuring that the "patches" do not introduce errors into the program. Dependence on the program author's "local knowledge" of the code increases at an alarming rate.

To operate these software tools, specialist staff are required. When programming CAAD applications, however, other specialist skills are also required, based on knowledge of design practices. It is then difficult to get both skills to bear on CAAD problems satisfactorily; not least is the difficulty of avoiding interdisciplinary conflicts among staff concerned with their separate status and career prospects.

1.4. The Compounding Problem

The complexity of the code, the prescribed nature of the programs, the difficulty in getting suitable staff, and the time taken to get an integrated CAAD system off the ground are all factors which compound together to make further problems.

Staff turnover is inevitable in the typical lifespan of a CAAD undertaking. Prescriptive code is notoriously difficult for persons other than its author to pick up and develop. Difficulties in getting different people to work on the same code means that system development soon lacks a co-ordinated approach, program code becomes fragmented and, typically, versions of any one utility become as numerous as there are programmers requiring it. Documentation, which is particularly necessary with complex prescriptive code, is by the same token, particularly difficult and time consuming to produce. It is often sacrificed where there is pressure to get the system up and working.

The programs always lag behind the aspirations of the users. The difficulties of making changes has been commented on earlier. A CAAD system cannot be viewed as a finished product. Building regulations change, in-house policy changes, the needs met by the organisation change. Changes in the specification occur frequently, even before the system first goes into production use. The programs become more and more patched until the system becomes unworkable.

Sooner or later some change will be required which the strained and mutilated programs simply cannot support. The remedy in theory is very simple - replace the system as one would replace any obsolete office resource. In practice, however, this implies a recognition of the limited life-span of the system such that the replacement system would have been under development alongside the one in use. Such a fruitful recognition of a CAAD system's lifespan is not easy in the light of the software investment that is commonly involved. All too easily, the system would eventually die without an heir, leaving many users with a distorted opinion of CAAD programs having experienced a system in its lingering senility.

1.5. The Future?

In discussing the difficulties of conventional or prescriptive programming, John Backus highlighted two levels of the problem (2). Firstly there is the machine architecture itself, made up of a central processing unit, a store and a connecting tube. Up and down this tube single words are pumped back and forth as the program accesses and modifies the data in the store. This tube is aptly described as a bottleneck. Secondly, conventional programming languages follow the same philosophy, being built round the assignment statement which works with one word at a time. Backus pointed out that this approach not only sets up a literal bottleneck in the machine but an intellectual bottleneck, preventing radical new thinking. The same tunnel vision is also evident on a third level where application programmers are constrained to work with fully prescribed data structures on the same word-at-a-time basis. Programmers, with their investment in knowledge of conventional programming techniques, too readily try to mould each and every problem to suit their tools, rather than develop their tools to suit the problems. If we stay with conventional or prescriptive programming, the only future for the wide scale use of integrated CAAD systems would be to enforce conformity on both individuals and organisations. Even if this were possible, it would not be desirable.

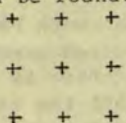
In a report on integrated CAAD systems by EdCAAD (3) two new software techniques were identified as indicating the first steps towards a new software technology which can overcome these problems, namely the relational view of data and descriptive languages. This paper follows an initial investigation into PROLOG, a descriptive language based on predicate calculus (4). PROLOG is described as descriptive rather than prescriptive because programs written in it describe the world of known information and relationships, rather than prescribe the steps to be taken by the machine in dealing with any one particular problem.

The rest of the paper takes an apparently simple problem and attempts to solve it with prescriptive languages and with PROLOG. The problem, set up at the SSHA, is somewhat abstract in nature but illustrates well some of the difficulties associated with CAAD programming (5). The solutions are compared and the differences between the prescriptive and descriptive approaches drawn out. It is apposite to point out that the programmers involved were experienced with prescriptive languages (6) but not in PROLOG, a factor which strengthens the conclusions.

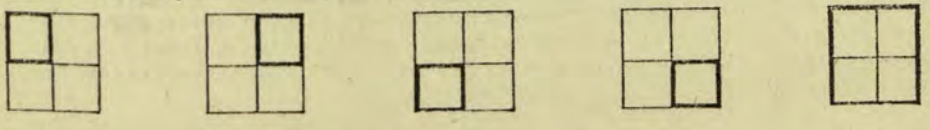
2. THE SHAPES PROBLEM

2.1. The Problem

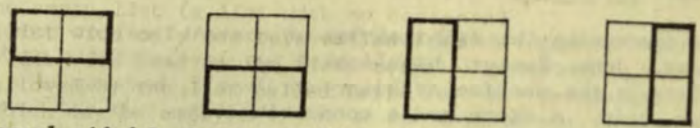
Given a data base of points (x and y co-ordinates), determine how many shapes of specified form can be found. For example in the given data base:



There are 5 squares:



There are 9 rectangles:

(5 squares plus) 

There are also 20 equal sided L shapes and 36 arbitrary L shapes.

The solutions in FORTRAN, C and PROLOG are given later.

2.2. The Prescriptive Solutions

FORTRAN is still one of the most commonly used languages, whereas C is an example of a newer generation of languages suitable for technical applications (7). This particular problem does not really draw out the benefits of the newer languages over FORTRAN, but the fundamental principle of prescriptive programming is common to both.

These FORTRAN and C programs satisfactorily answer the questions:

How many rectangles are there?
How many squares are there?
How many equal sided L shapes are there?
How many arbitrary L shapes are there?

If the user then chooses to ask how many T shapes are there, the programmer would reply:

'Ah! You didn't tell me that you would want to know about T shapes. I will have to develop my program further.'

And so it would continue. The prescriptive approach can only offer programs to answer prescribed questions. The programmer must anticipate the questions that will be asked. His programs are likely to grow and grow as more and more is asked of them, with all the attendant problems that that brings.

2.3. The Descriptive Solution

The PROLOG solution is here called descriptive because rather than prescribing a solution to a particular problem step by step, generalised statements of what is true are held. When the user requests a shape, he is in fact asking for all the possible conditions under which his defined shape exists. If for example, he wants to know how many squares there are, he will set the goal as a square and the program will determine how many ways in which the goal can be satisfied. He could equally well set the goal as an L shape, a T shape, or any shape he wishes.

Many readers may well be unfamiliar with PROLOG and so an explanation of the code and how questions are asked accompanies the listing. Clearly the interrogation procedures would have to be enhanced if this were to be offered as a user package, but the exercise was purely to draw out the underlying principles of the descriptive approach.

2.4. The FORTRAN Listing

```
      DIMENSION IPX(128),IPY(128)
C***** READ IN DATA POINTS
      WRITE(5,100)
100   FORMAT(' ENTER UP TO 128 POINTS,
     X 1 SET/LINE, (END WITH -1,-1)')
      NP=0
      DO 1 I=1,128
      READ(5,101)IX,IY
101   FORMAT(2I)
      IF(IX.LT.0)GOTO 2
      NP=I
      IPX(I)=IX
      IPY(I)=IY
1     CONTINUE
2     NR=0
      NS=0
      NEL=0
      NUL=0
C***** TEST FOR RECTANGLES, SQUARES, EQUAL L SHAPES
C***** AND ARBITRARY L SHAPES
      DO 3 I=1,NP
      DO 3 J=1,NP
          IF(I.EQ.J)GOTO 3
          IF(IPY(I).NE.IPY(J))GOTO 3
          IHD=IPX(J)-IPX(I)
          DO 3 K=1,NP
              IF(J.EQ.K)GOTO 3
              IF(IPX(J).NE.IPX(K))GOTO 3
              IVD=IPY(K)-IPY(J)
              NUL=NUL+1
              IF(IABS(IHD).EQ.IABS(IVD))NEL=NEL+1
              DO 3 L=1,NP
                  IF(I.EQ.L)GOTO 3
                  IF(IPX(I).EQ.IPX(L))GOTO 3
                  IVD2=IPY(L)-IPY(I)
                  IF(IVD2.NE.IVD)GOTO 3
                  NR=NR+1
                  IF(IABS(IVD).EQ.IABS(IHD))NS=NS+1
3     CONTINUE
      NR=NR/4
      NS=NS/4
C***** PRINT OUT THE RESULTS
      WRITE(5,102)NR
102   FORMAT(' THERE ARE',I4,' RECTANGLES')
      WRITE(5,103)NS
103   FORMAT(' THERE ARE',I4,' SQUARES')
      WRITE(5,104)NEL
104   FORMAT(' THERE ARE',I4,' EQUAL SIDED L SHAPES')
      WRITE(5,105)NUL
105   FORMAT(' THERE ARE',I4,' ARBITRARY L SHAPES')
      STOP
      END
```

2.5. The C Listing

```
main()
{
int i,j,k,l,vd,vd2,hd,max,gridx[128],gridy[128];
int rect,square,eq_els,non_eq_els;
        max = rect = square = eq_els = non_eq_els = 0;
/* read in data points */
        printf(" ENTER UP TO 128 POINTS,");
        printf(" 1 SET PER LINE.(END WITH -1 -1)\n");
        while(1) {
                scanf("%d%d",&gridx[max],&gridy[max]);
                if((gridx[max] == -1) || (max > 128))
                        break;
                ++max;
        }
/* test for rectangles, squares, equal sided L shapes
   and arbitrary L shapes */
      for(i = 0;i < max;++i)
       for(j = 0;j < max;++j)
        if(j != i) {
         if((gridy[j] - gridy[i]) == 0) {
           hd = gridx[j] - gridx[i];
           for(k = 0;k < max;++k)
            if(k != i) {
            if((gridx[k] - gridx[i]) == 0) {
              vd = gridy[k] - gridy[i];
               ++non_eq_els;
               if((abs(vd) - abs(hd)) == 0)
                      ++eq_els;
               for(l = 0;l < max;++l)
                if(l != j) {
                 if((gridx[j] - gridx[l]) == 0) {
                    vd2 = gridy[l] - gridy[j];
                   if(vd == vd2) {
                    ++rect;
                    if(abs(vd) == abs(hd))
                        ++square;
                   }
                 }
               }
           }
           }
         }
        }
      rect =/ 4;
      square =/ 4;
/* print out the results */
   printf(" THERE ARE %d RECTANGLES\n",rect);
   printf(" THERE ARE %d SQUARES\n",square);
   printf(" THERE ARE %d EQUAL SIDED L SHAPES\n",eq_els);
   printf(" THERE ARE %d ARBITRARY L SHAPES\n",non_eq_els);
}
```

2.6. The PROLOG Listing

```
/* data points for the example */
p(0,0).        p(10,0).       p(20,0).
p(0,10).       p(10,10).      p(20,10).
p(0,20).       p(10,20).      p(20,20).

/* the program */
horizontals([]).
horizontals([Head|Tail]):- horiz(Head), horizontals(Tail).

horiz([]).
horiz([LOC]).
horiz([loc(p(X1,Y),D1),loc(p(X2,Y),D2)|Rest]):-
       p(X1,Y), p(X2,Y), X2=\=X1, D1 is X2-X1,
       horiz([loc(p(X2,Y),D2)|Rest]).

verticals([]).
verticals([Head|Tail]):- vert(Head), verticals(Tail).

vert([]).
vert([LOC]).
vert([loc(p(X,Y1),D1),loc(p(X,Y2),D2)|Rest]):-
       p(X,Y1), p(X,Y2), Y1=\=Y2, D1 is Y2-Y1,
       vert([loc(p(X,Y2),D2)|Rest]).

abs(A,A):- A>0.
abs(A,B):- A=<0, B is -A.
```

2.7. The PROLOG Listing Explained

The following explanation is intended to give the reader a simple understanding of the descriptive nature of the program, and not to teach or even introduce PROLOG as such.

(a) The following terms (or parameters) are used (note that terms beginning with capital letters are variables):

co-ordinates	X,Y	
point	P	being the complex term p(X,Y)
distance	D	
location	LOC	being the complex term loc(P,D) or loc(p(X,Y),D) where D is the distance to the next location on the line

(b) Lines are set up as lists of locations.
Shapes are seen as lists of lines in the horizontal direction and lines in the vertical direction.
Note the PROLOG semantics used for a list which is a special type of term:

```
[]                  the empty list (a list with no contents)
[Item]              a list with only one term called Item
[Head|Tail]         a list with Head as its first term
                    followed by the list called Tail
                    (which may be empty)
[First,Second|Rest] a list with terms First and Second
                    followed by the list called Rest
                    (which may be empty)
```

(c) The data consists of a series simple logic assertions that points of the form p(X,Y) exist, eg:

```
p(10,20).
```

Unlike the restrictions built into the prescriptive programs, there is no limit to the number of alternative points which can be asserted (other than the obvious restraint of machine size). Data in PROLOG is in fact no different in essence from program assertions, although it would be sensible to set them up in a separate file.

(d) The program can be "read" as follows:

```
horizontals([]).
horizontals([Head|Tail]):- horiz(Head), horizontals(Tail).
```

a list of lines is to be judged to be horizontal if in fact there are no lines in the list;
or the list of lines is horizontal provided:-
the first line is horizontal and the rest of the lines are horizontal.

```
horiz([]).
horiz([LOC]).
horiz([loc(p(X1,Y),D1),loc(p(X2,Y)|Rest]):-
        p(X1,Y), p(X2,Y), X2=\=X1, D is X2-X1,
        horiz([loc(p(X2,Y)|Rest]).
```

a line is to be judged as horizontal if there are in fact no locations in it; or only one location;
or the line is horizontal provided:-
the points of the first two locations exist,
the points have the same y co-ordinate,
the points do not have the same x co-ordinate (not coincident),
the distance in x is as specified (perhaps a variable),
and the second location with the rest of the list constitutes a horizontal line.

```
verticals    cf. horizontals
vert         cf. horiz

abs(A,A):- A>0.
abs(A,B):- A=<0, B is -A.
```

```
the absolute value of A is A provided A > 0;
or the absolute value of A is B provided:-
     A =< 0,
     and B is -A.
```

(e) Interrogations are asked by requesting the desired shape filling in as much of the detail as the user wishes. The program responds with the sets of conditions under which the request can be satisfied. For example:

To get squares omitting mirror images, request:-

```
horizontals
([[loc(P1,D),loc(P2,0)],[loc(P3,D),loc(P4,0)]]),
verticals
([[loc(P3,D),loc(P1,0)],[loc(P4,D),loc(P2,0)]]).
D>0.
```

To get L shapes including mirror images, with verticals twice as long as horizontals, request:-

```
horizontals([[loc(P1,DH),loc(P2,0)]]), abs(DH,DHA),
verticals([[loc(P1,DV),loc(P3,0)]]), abs(DV,DVA),
DVA is DHA*2.
```

To get T shapes intersecting at p(10,20), request:-

```
horizontals([[loc(P1,DH),loc(p(10,20),DH),loc(P3,0)]]),
verticals([[loc(P4,DV),loc(p(10,20)]]), DH>0, DV>0.
```

2.8. The Ultimate Program

As the shapes program was examined in depth in the preparation of these notes the surprising realisation came that the most flexible PROLOG solution was in fact no program at all! The data and basic PROLOG facilities are sufficient and in fact easier to use than the program listed earlier. As there are therefore no procedures for horizontal or vertical lines, the user is not coerced into thinking that the program is suitable for only rectilinear shapes. Questions would now be posed as follows:

To get squares omitting mirror images, request:-

```
p(X1,Y1), p(X2,Y1), D is X2-X1, D>0,
p(X1,Y2), D is Y2-Y1, p(X2,Y2).
```

To get isosceles triangles with horizontal bases, request:-

```
p(X1,Y1), p(X2,Y2), DHB is X2-X1, DHB>0,
DV is Y2-Y1, DV>0, p(X3,Y1), DHB is X3-X2.
```

Thus the example problem is, astonishingly, so trivial to PROLOG that no special procedures whatsoever are required, even though it is to all intents and purposes impossible to handle in a totally general way by prescriptive programming.

3. CONCLUSIONS

3.1. Prescriptive and Descriptive Approaches Compared

(a) Fundamentally, like is not being compared with like. The prescriptive programs, in FORTRAN and C, will only deal with the specified shapes - the descriptive program, in PROLOG, is much more general.

(Note that the only reason why the prescriptive programs appear to deal so easily with the four shapes is that L shapes and rectangles are convenient stage shapes on the way to determining a square!)

(b) The time taken to analyse the problem, code it and implement it using the descriptive approach was about half the time taken using the prescriptive approach (one hour as against two).

(c) The potential for program errors was much greater in the prescriptive approach than in the descriptive approach. Both the FORTRAN and C programmers failed on their first implementation attempt by becoming confused on horizontal and vertical checking, by checking that the x values (or was it y) were equal (or was it unequal). Such visualisation problems did not occur when coding the PROLOG solution.

(d) Typically a prescriptive program is even more restricted in use than originally envisaged, although in this simple example the limitations were known at the outset by the programmer (if not by the user). In the descriptive program, however, the final product was found to be much more powerful than originally envisaged. In addition to the facility for testing for any shape, the program can also test for mirror images, test for shapes with certain points fixed, test for shapes with certain sides of specified length, or test for shapes with several sides in a specified relationship.

(e) The descriptive solution is able to provide a range of potential solutions where only some of the possible data are provided. Such a partial solution to partial data capability is not available in prescriptive programming.

(f) Modifying the prescriptive programs would be much more onerous and error prone than modifying descriptive programs. If, for example, the prescriptive programs had been written to test for T shapes (and therefore inverted Ts), to amend the program to cater for Ts on their side would have involved awkward and finicky changes. In the case of the descriptive program the T on its side can conveniently be handled with a simple additional procedure. In detail this procedure would be:

```
bothways(Lines1,Lines2):- horizontals(Lines1),
                          verticals(Lines2).
bothways(Lines1,Lines2):- horizontals(Lines2),
                          verticals(Lines1).
```

Such is the power of PROLOG that the above could in fact be handled by a particular form of question, but the question would be fairly long and complex.

(g) The reduction in program size in the descriptive approach is primarily due to the code not having to contain precise instructions on internal machine operations. The shorter program length also reflects a shift in responsibility for the operations of an application from the programmer to the user. The more procedures that are written, the easier the PROLOG program may be to use - but the less general it may become.

(h) To those not familiar with any form of programming, the descriptive program code has proved to be much more comprehensible than the prescriptive code. However, the same people tend not to realise the limitations that conventional programming methods place on their use of computing resources.

On the other hand, people already familiar with prescriptive programming philosophy find it much harder to think in the descriptive manner than do others without such a background.

3.2. The Potential for CAAD

In assessing the value of a new software technique for Computer Aided Architectural Design consideration must be given to the way it is envisaged that future systems should function. Where possible, this projection should be free of the limitations of current software or hardware technology. The aim of EdCAAD as a group in this area has been summed up by Aart Bijl, the group's director, as follows:

> We aim to make computing more accessible to architects - to enable architects to undertake their own journeys through fields of knowledge. Present CAAD technology, offering architects turnkey systems, is equivalent to the first Stockton to Darlington railway - you have to be happy with where the track takes you. New technology needs to support architects 'driving' their own computers, selecting their own routes and destinations without a 'chauffeur' and 'mechanic' in attendance.

Descriptive programming as a technique is not yet in a suitable form for practical use on a wide scale. However, this example illustrates the potential that the technique has in the field of Computer Aided Architectural Design. With the technique comes the promise of a wider use of integrated CAAD systems, each developing under the control of its user or user organisation; the promise of a diminishing dependence on computer specialists standing between the user and computing resources.

References and Acknowledgements:

(1) "Computer Aided Architectural Design at the Scottish Special Housing Association", SSHA, Autumn 1977
(2) Backus J., "ACM Turing Award Lecture", Comm. A.C.M. 21 8, Aug. 1978
(3) Bijl A., Stone D. and Rosenthal D., "Integrated CAAD Systems", EdCAAD Studies, 1979
(4) Pereira L.M., Pereira F.C.N. and Warren D.H.D., "User's Guide to DEC System-10 PROLOG", University of Edinburgh, 1978
(5) Shawcross G., "A Dream World", Unpublished SSHA paper written following a presentation to the CAD 78 Conference by Scrivener, Edmonds and Thomas.
(6) Nash J. for his work on the prescriptive solution and the C code.
(7) Kernighan B.W. and Ritchie D.M., "The C Programming Language", Prentice-Hall, 1978

DERIVING DIFFERENT UNIFICATION ALGORITHMS FROM A SPECIFICATION IN LOGIC

by

Georg Winterstein
Manfred Dausmann
Guido Persch

Institut für Informatik II, Universität Karlsruhe,
Postfach 6380, D-7500 Karlsruhe 1, Fed. Rep. of Germany

1. INTRODUCTION

Predicate logic can be regarded as a very high level, formal language for describing problems. It also can be used as a programming language which may be interpreted on a computer. Therefore any problem description in logic can also be seen as a program. At any phase during the design of a system the execution of the specifications will detect inconsistencies and underspecifications.

In this paper we are mainly interested in the study of the description of algorithms using logic. Following Kowalski [1] an algorithm consists of a problem axiomatization and a specification of a certain control. Different axiomatizations of the same problem domain lead to different algorithms as well as changing of the control component. Also the borderline between logic and control part of a specification is not fixed, i. e. parts of the axiomatization may also be seen as a control specification and v. v.

The usefulness of predicate logic as a programming language and its implementation have been studied in detail before [2 - 5]. Our main emphasis is to show that predicate logic may also serve as a specification language for software engineering. It is formal, easy to understand, descriptive, provides abstraction facilities for data and algorithms, and allows automatic proofs of the completeness of a specification. Also contradictions may be detected by machine. On the other hand it does not impose any implementation decision or concrete realization. Logic has been used for the specification of simple concurrent problems [6 - 7]. For a rather complex problem, the evaluation of a query in a relational data base system, a complete problem specification has been given [8 - 9].

In this paper we consider as an example the unification problem in first order logic. For the syntactical representation of the specifications we use the conventional notation of logic programming. However no control for the execution of such a description is implicitly associated. If necessary the control part is described seperately.

At the beginning we analyze Robinson's first algorithm given in [10]. Then we give another more problem oriented specification. Different control strategies for executing this specification lead to different unification algorithms to compute the most general unifier. We show that parts of the control can be incorporated into the logic description.

These algorithms have been discribed in the literature before [11 - 17] and their time and space complexity have been investigated. Therefore the unification problem is a good example to show the usefulness of predicate logic as a specification language.

2. ROBINSON'S UNIFICATION ALGORITHM

In his fundamental paper: 'A Machine-Oriented Logic Based on the Resolution Principle' [10] J. A. Robinson also gave a description of of an unification algorithm.

"The following process, applicable to any finite nonempty set A of well formed expressions, is called Unification Algorithm:

Step 1: Set $s0 = \varepsilon$ and $k = 0$, and go to step 2.

Step 2: If Ask is not a singleton, go to step 3. Otherwise set $sA = sk$ and terminate.

Step 3: Let Vk be the earliest, and Uk the next earliest in the lexical ordering of the disagreement set Bk of Ask. If Vk is a variable, and does not occur in Uk set $sk+1 = sk\{Uk/Vk\}$, add 1 to k, and return to step 2. Otherwise, terminate."

...

"If A is any set of well-formed expressions, we call the set B the disagreement set of A whenever B is the set of all well-formed subexpressions of well-formed expressions in A, which begin at the first symbol position at which not all well-formed expressions in A have the same symbol."

We assume that the reader is already familiar with well-formed expressions, substitution and substitution components, instantiation (the application of a substitution to an expression) and the composition of substitutions. For a precise definition the reader is referred to [10].

Let us translate this description of an unification algorithm into a logic specification:

Logic-part:

```
UNIFY(a,sk,sa) <- APPLY(sk,a,as),
                  U*(a,as,sk,sa);

U*(a,as,sk,sa) <- IS_SINGLETON(as),
                  EQUAL(sa,sk);

U*(a,as,sk,sa) <- DISAGREE(as,uk,vk),
                  NOT_OCCUR(uk,vk),
                  COMPOSE(uk,vk,sk,sk+1);
                  UNIFY(a,sk+1,sa);
```

Control-part:

The procedure declarations are invoked in the order they are given. A goal statement is evaluated from left to right.

We have not specified the procedures APPLY, COMPOSE, DISAGREE and NOT_OCCUR. Their meaning is as described in [10].

Example:

Unification of A = {P(X,A),P(F(U),U)} where X and U are a variables.

```
<- UNIFY({P(X,A),P(F(U),U)},ε,sa)

<- APPLY(s,{P(X,A),P(F(U),U)},as),
   U*({P(X,A),P(F(U),U)},as,ε,sa)

<- U*({P(X,A),P(F(U),U)},{P(X,A),P(F(U),U)},ε,sa)

<- DISAGREE({P(X,A),P(F(U),U)},uk,vk),
   NOT_OCCUR(uk,vk),
   COMPOSE(uk,vk,ε,sk+1),
   UNIFY({P(X,A),P(F(U),U)},sk+1,sa)

<- NOT_OCCUR(X,F(U)),
   COMPOSE(X,F(U),ε,sk+1),
   UNIFY({P(X,A),P(F(U),U)},sk+1,sa)
```

```
<- COMPOSE(X,F(U),ε,sk+1),
   UNIFY({P(X,A),P(F(U),U)},sk+1,sa)

<- UNIFY({P(X,A),P(F(U),U)},{F(U)/X},sa)

<- APPLY({F(U)/X},{P(X,A),P(F(U),U)},as),
   U*({P(X,A),P(F(U),U)},as,{F(U)/X},sa)

<- U*({P(X,A),P(F(U),U)},{P(F(U),A),P(F(U),U)},{F(U)/X},sa)

<- DISAGREE({P(F(U),A),P(F(U),U)},uk,vk),
   NOT_OCCUR(uk,vk),
   COMPOSE(ul,vl,{F(U)/X},sk+1),
   UNIFY({P(X,A),P(F(U),U)},sk+1,sa)

<- NOT_OCCUR(U,A),
   COMPOSE(U,A,{F(U)/X},sk+1),
   UNIFY({P(X,A),P(F(U),U)},sk+1,sa)

<- COMPOSE(U,A,{F(U)/X},sk+1),
   UNIFY({P(X,A),P(F(U),U)},sk+1,sa)

<- UNIFY({P(X,A),P(F(U),U)},{F(A)/X,A/U},sa)

<- APPLY({F(A)/X,A/U},{P(X,A),P(F(U),U)},as),
   U*({P(X,A),P(F(U),U)},as,{F(A)/X,A/X},sa)

<- U*({P(X,A),P(F(U),U)},{P(F(A),A)},{F(A)/X,A/X},sa)

<- IS_SINGLETON({P(F(A),A)}),
   EQUAL(sa,{F(Z)/X,A/U})

<- EQUAL(sa,{F(Z)/X,A/U})

<-
```

From the description of Robinson's unification algorithm in logic we see immediately its disadvantages.

- The set A has to be kept during the execution of the whole algorithm.
- To compute the disagreement set Bk of Ask the whole set Ask has to be considered especially those parts which are already known to be identical.
- The substitution sk has to be applied explicitly to A.

The logic program above is also no good description of the unification problem because it does not describe the problem in general but only one way how to solve it. It therefore forces a special solution on an implementor. Perhaps it is due to this fact that it took more than six years before a significant improvement of the unification algorithm was achieved.

In the next section we give a more problem oriented specification of unification.

3. ANOTHER SPECIFICATION OF THE UNIFICATION PROBLEM

For the sake of simplicity we restrict ourselves to the problem of unifying two expressions e1 and e2.

Logic-part:

```
UNIFY(e1,e2,s) <- EQUAL(e1,e2);

UNIFY(e1,e2,s) <- IS_FUNCTERM(e1,head1,args1),
                  IS_FUNCTERM(e2,head2,args2),
                  EQUAL(head1,head2),
                  UNILIST(args1,args2,s);

UNILIST(args1,args2,s) <- IS_EMPTY(args1),
                          IS_EMPTY(args2);

UNILIST(args1,args2,s) <- TAKE_CORR_PAIR(args1, args2,
                                         e1,    e2,
                                         args'1,args'2),
                          UNIFY(e1,e2,s);
                          UNILIST(args'1,args'2,s);

UNIFY(e1,e2,s) <- IS_VAR(e1),
                  MEMBER(e1,t1,s),
                  NOT_OCCUR(e1,t1,s),
                  UNIFY(t1,e2,s);

UNIFY(e1,e2,s) <- UNIFY(e2,e1,s);
```

Control-part:

The procedure declarations are invoked in the order they are given. The procedure calls are elaborated from left to right. A call of NOT_OCCUR is delayed until both parameters are bound.

We have not specified the procedures EQUAL, IS_FUNCTERM, IS_VAR, NOT_OCCUR, MEMBER and TAKE_CORR_PAIR. Their meaning is obvious.

We see that the logic part of the specification only gives the description of a unifier. The specification of a most general unifier for two terms is achieved by giving an additional control specification. In this case this leads to a much more elegant and understandable specification of the most general unifier than the axiomatization of: "A unifier s of a set of terms C is a most general unifier of C if for every unifier t of C there is a substitution r so that s = rt."

Notice that nothing is said about the data structure for s, e. g. if it is organized as list or set.

Example:

```
<- UNIFY(P(X,A),P(F(U),U),s)

<- IS_FUNCTERM(P(X,A),head1,args1),
   IS_FUNCTERM(P(F(U),U),head2,args2),
   EQUAL(head1,head2),
   UNILIST(args1,args2,args'1,args'2,s)

   ...

<- UNIFY(X,F(U),s),
   UNILIST((A),(U),s)

<- IS_VAR(X),
   MEMBER(X,t1,s),
   NOT_OCCUR(X,t1,s),
   UNIFY(t1,F(U),s),
   UNILIST((A),(U),s)

<- NOT_OCCUR(X,t1,{t1/X}),
   UNIFY(t1,F(U),{t1/X}),
   UNILIST((A),(U),{t1/X})

<- NOT_OCCUR(X,F(U),{F(U)/X}),
   UNILIST((A),(U),{F(U)/X})

<- IS_VAR(U),
   UNIFY(U,A,{F(U)/X})

<- UNIFY(U,A,{F(U)/x})

<- IS_VAR(U),
   MEMBER(U,t1,{F(U)/X}),
   NOT_OCCUR(U,t1);
   UNIFY(t1,A,{F(U)/X})
```

```
<- NOT_OCCUR(U,t1,{F(t1)/X,t1/U}),
   UNIFY(t1,A,{F(t1)/X,t1/U})

<- NOT_OCCUR(U,A,{F(A)/X,A/U})

<-
```

In this example we have chosen a set representation for s.

It may seem to be more natural to specify the unification of two terms e1 and e2 by

```
UNIFY(e1,e2,s) <- APPLY(s,e1,e),
                  APPLY(s,e2,e);
```

because this would avoid to introduce the structure of the terms at the first level of the specification. Then however it must be known to give the specification of APPLY. For our purposes the above specification is most convenient but in general we cannot give a methodology for a specification, i. e. if it should be more functional or operational oriented.

From the above specification we see that the procedure MEMBER plays a central role in the specification of UNIFY. It can either be used to compose a new substitution component with the substitution computed so far, or it can be used to inspect if a variable has already been bound. An implementation of this specification using the double function of MEMBER (using a list representation for s) is the unification algorithm given by Boyer and Moore [11].

The unification algorithm given by Baxter [12] simply delays the execution of all NOT_OCCUR calls to the end of the algorithm. Then they can be implemented very elegantly by a topological sort.

All other unification algorithms use MEMBER only for inserting a new substitution component. They therefore have to assure that the variable occurring in the substitution component is not already in s. As we have seen before this is achieved in the algorithm of Robinson by applying the substitution generated so far to the original set. This guarantees that all variables which appear on the left hand side of a substitution component in s do not occur in the terms under consideration and therefore any new substitution component may be added to s without conflicts.

The unification algorithms described in [13 - 15] use another technique to achieve this goal. The execution of MEMBER is delayed and whenever there is a goal of the form

```
<- MEMBER(var,term1,s)  ...  UNIFY(var,term2,s)  ...
```

they rewrite it to the form

```
<- MEMBER(var,term1,s)  ...  UNIFY(term1,term2,s)  ...
```

By this the number of occurrences of var is reduced by one. The elaboration of MEMBER is delayed until all calls of UNIFY and UNILIST are done. Then the substitution s is generated by the evaluation of MEMBER. At the end all calls of NOT_OCCUR are processed.

Parts of the control can be incorporated into the logic-part of the specification which leads to a refined description:

Logic-part:

```
UNIFY(e1,e2,s) <- EQUAL(e1,e2);

UNIFY(e1,e2,s) <- IS_FUNCTERM(e1,head1,args1),
                  IS_FUNCTERM(e2,head2,args2),
                  EQUAL(head1,head2),
                  UNILIST(args1,args2,s);

UNILIST(args1,args2,s) <- IS_EMPTY(args1),
                          IS_EMPTY(args2);

UNILIST(args1,args2,s) <- TAKE_CORR_PAIR(args1, args2,
                                         e1,    e2,
                                         args'1,args'2),
                          UNIFY(e1,e2,s);
                          UNILIST(args'1,args'2,s);

UNIFY(e1,e2,s) <- IS_VAR(e1),
                  MEMBER(e1,e2,s),
                  NOT_OCCUR(e1,e2,s);

UNIFY(e1,e2,s) <- UNIFY(e2,e1,s);
```

Control-part:

The procedure declarations are invoked in the order they are given. A goal statements is elaborated from left to right with the exception that calls of MEMBER and NOT_OCCUR are delayed. The rewrite rule given above is applied whenever possible. The calls of NOT_OCCUR are processed after the calls of MEMBER.

Though an implementation of the algorithm above is rather efficient it still can be improved. This is done by liberating the order in which the calls of UNIFY and UNILIST are executed i. e. the expressions are not scanned from left to right anymore. It has the advantage that the NOT_OCCUR procedure is independant from s. Because if a most general unifier exists there must be a partial ordering among the variables which build the substitution. This order can be determined by allowing to process only those calls of UNIFY whose variable in the first component does not occur in another call. This may be achieved by changing the control part of the specification into

Control-part:

The procedure declarations are invoked in the order they are given. A goal statement may be executed in any order. All calls of UNIFY with a variable as first parameter are delayed. If a goal consists entirely of calls of UNIFY of that form those calls may be processed where the variable does not occur anywhere else in the goal statement. For those calls for which this is not the case the following rewrite rule is applied if possible

```
<- ... UNIFY(var,term1,s), ... UNIFY(var,term2,s), ...

--->

<- ... UNIFY(term1,term2,s), ... UNIFY(var,term1,s), ...
```

Concrete implementations of this specification use either counters to store the number of occurrences of a variable in the expressions [16], or use a data structure in which every variable is represented only once together with additional pointers [17].

Example:

Unification of {P(X,Z,Y),P(F(Y),A,F(Z))} where X, Y, and Z are variables

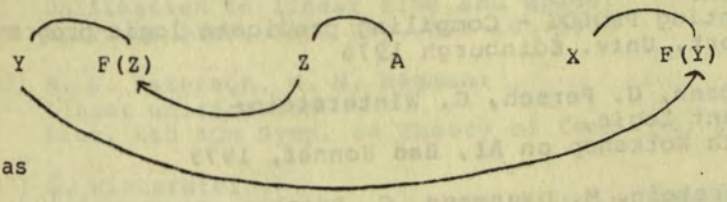
```
<- UNIFY(P(X,Z,Y),P(F(Y),A,F(Z)),s)

   ...

<- UNIFY(Y,F(Z),s), NOT_OCCUR(Y,F(Z)),
   UNIFY(Z,A,s),    NOT_OCCUR(Z,A),
   UNIFY(X,F(Y),s), NOT_OCCUR(X,F(Y))
```

```
<- UNIFY(Y,F(Z),s),
   UNIFY(Z,A,s),
   UNIPY(X,F(Y),s)

<- UNIFY(Y,F(Z),{F(Y)/X}),
   UNIFY(Z,A,{F(Y)/X})

<- UNIFY(Z,A,{F(F(Z)/X,F(Z)/Y})

<-
```

In the implementations mentioned above the goal statement

<- UNIFY(Y,F(Z),s), UNIFY(Z,A,s), UNIFY(X,F(Y),s)

would be coded as



or as

Y	F(Z)	#X = 1
Z	A	#Y = 2
X	F(Y)	#Z = 2

4. CONCLUSION

We have shown that the family of unification algorithms can be specified by a logic program togehter with different ways of controlling its execution. The logic part may be refined by incorporating parts of the control specification in it. Concrete implementations of the different unification algorithms, which are explained in detail in [21] code the control part of the specification into the data structure they use while the program structure reflects the logic part of the specification. We believe that a specification of other families of algorithms in logic will give more insight in the behaviour of algorithms and their relations to eachother. It also may help to discover more efficient implementations for certain problems. As logic also allows a formal specification of data structures and abstract data types it seems to be a useful and practicable tool for formal specification in software engineering.

5. REFERENCES

[1] R. A. Kowalski:
Algorithm = Logic + Control,
CACM 22-7-79, pp. 424 - 436

[2] R. A. Kowalski:
Logic for Problem Solving,
DCL Memo 75, Univ. of Edinburgh, 1974

[3] R. A. Kowalski:
Predicate Logic as Programming Language,
IFIP 74, pp. 596 - 574, North Holland

[4] A. Colmerauer, H. Kanoui, R. Pasero, P. Roussel:
Un systeme de communication homme-machine en francais,
Universite d'Aix-Marseille, Luminy 1972

[5] D. Warren:
Implementing PROLOG - Compiling predicate logic programs
DAI Report, Univ. Edinburgh 1976

[6] M. Dausmann, G. Persch, G. Winterstein:
Concurrent Logic,
Proc. 4th Workshop on AI, Bad Honnef, 1979

[7] G. Winterstein, M. Dausmann, G. Persch:
A new method for describing concurrent problems
based on logic,
Interner Bericht Nr. 10, Universität Karlsruhe, 1980

[8] P. Trum:
Logik als Spezifikationssprache: Auswertung einer Anfrage in einem relationalen Datenbanksystem auf der konzeptuellen Ebene,
Diplomarbeit, Universität Kaiserslautern, 1980

[9] N. Heck:
Logik als Spezifikationssprache: Auswertung einer Anfrage in einem relationalen Datenbanksystem auf der internen Ebene,
Diplomarbeit, Universität Kaiserslautern, 1980

[10] J. A. Robinson:
A machine-oriented logic based on the resolution principle,
J.ACM 12-1-65, pp. 135 - 150, 1965

[11] R. S. Boyer, J. S. Moore:
The sharing of structure in theorem proving programs,
Mach. Int. 7, pp. 101-116, 1972

[12] L. D. Baxter:
An efficient unification algorithm,
Univ. of Waterloo, Res. Rep. CS-73-23, 1973

[13] L. D. Baxter:
A practically linear unification algorithm,
Univ. of Waterloo, Res. Rep. CS-76-13, 1976

[14] J. A. Robinson:
Fast unification,
Workshop on Automated Theorem Proving,
Oberwolfach, Jan. 1976

[15] G. Huet:
Algebraic aspects of unification
Workshop on Automated Theorem Proving,
Oberwolfach, Jan. 1976

[16] A. Martelli, U. Montanari:
Unification in linear time and space,
Univ. of Pisa, Int. Rep. B76-16, 1976

[17] M. S. Paterson, M. N. Wegman:
Linear unification
Proc. 8th ACM Symp. on Theory of Comp. pp. 181 - 186, 1976

[17] G. Winterstein:
Prädikatenlogik-Programme mit evaluierbaren Funktionen
Ph. D. Thesis, Univ. of Kaiserslautern, 1978

[18] S. Tärnlund:
Logic Information Processing,
Report TRITA-IBADB-1034, Univ. of Stockholm, 1975

[19] F. G. McCabe:
Programmer's Guide to IC-PROLOG,
Imperial College, London, 1978

[20] B. M. Lichtman:
Features of Very High Level Programming with PROLOG,
M. Sc. Thesis, Imperial College, London, 1975

[21] P. Trum, G. Winterstein:
Description, implementation and practical comparison of
unification algorithms,
Univ. Kaiserslautern, Int. Ber. 6/78, 1978

CONTROLLING BACKTRACK IN HORN CLAUSES PROGRAMMING

Claudine LASSERRE - Hervé GALLAIRE
ENSAE, 10 Avenue Edouard Belin
Complexe aérospatial de Lespinet
31055 TOULOUSE CEDEX

Abstract

The purpose of this paper is to present a backtracking mechanism whose characteristics would be to erase only resolutions relevant to the conflict, and as few resolutions as possible. General algorithms are presented to perform it, i.e. to locate and analyse the conflict, and questions are raised concerning possible ways of making other choices to remedy.

Key words : Artificial Intelligence - Horn clauses programming - Derivation control - Backtracking control.

287

CONTROLLING BACKTRACK IN HORN CLAUSES PROGRAMMING

Introduction.

The problem of controlling a linear deduction on Horn clauses is that of determining an order of search space expansion, that is to say of fixing, at every step, the choice of the literal to solve and the choice of the clause to solve it. In fact, this control is shared between two processes :

(1) the forward execution process, where a first choice is set and tried, and
(2) the backtracking process, when a failure occurs in process (1), where it is to be decided what choices are to be erased and what new choices are to be made.

Of course, a reasonable control of process (1) may alleviate causes of backtracking.

In both cases there are two possible ways of controlling : either automatically, according to a general strategy, as Prolog [ROU] does and as we shall propose in the sequel, or according to specifications given by the user through metarules. We've already developed such metarules for forward control in [GAL] . Here we concentrate on the possibilities of operating a backtrack as "intelligently" as possible in the context of an interpreter that woud allow any ordering of literals to solve.

Backtracking process control.

In case of failure, the problem is to reconsider earlier choices of clauses that solved previously selected literals, that is to determine what resolutions are to be erased and what new choices are to be made to restart the forward process.

Given that derivation is a process of reducing in any order subproblems (ie literals) into other subproblems which will be dependent through the variables shared and not through the expansion order, it becomes necessary to fulfill the following requirements in order to obviate, as much as possible, unecessary backtracking to irrelevant choices :

- To have a unification algorithm permitting failure location (and analysis too), i.e. providing the resolutions accountable for a conflict of instantiation of a variable in the failing literal. It will then be sufficient to erase a part of these resolutions (i.e. to suppress a subchain of links in the chain of sucessive unifications to the irrelevant value) corresponding to a branch in the derivation tree. The informations gathered may eventually help to determine further choices.

- To have a backtrack mechanism permitting to keep the resolution of the brothers of the literal the resolution of which will be erased, in particular all the literals which have been solved afterwards in parallel branches, as far as we have chosen, in the previous step, to keep the links, if any, they could have added to the shared variables.

A proof procedure, as generally implemented (cf Prolog), actually establishes a "plan" of the derivations made, i.e. the schema (of problem reduction type) of the literals expansion, without substitutions, while in parallel it carries on the treatment of the unifications that validate this plan. Hence, to implement what preceeds, the main characteristics of these parts of the interpreter must be :

- That the unification algorithm construct a graph of the dependencies created between variables by unification, labelled by the number of the corresponding literal resolution in the graph, to enable the quick search of the resolutions responsible for a conflicting instantiation ; this graph must also be established rather easily in an on-line manner, and updated quickly too when one or several literals resolutions are suppressed.

- That the brother literals resolutions in the plan don't be linked by the chronologic expansion order, so that in case of pruning it is possible to come not to a previous situation but to the current situation without the erased branch, and so that in any case any literal can be chosen for expansion.

Cox [Cox] and Pereira [PER 1] [PER 2] give partial and sometimes identical solutions to these problems. Here are the possibilities we retain to carry on the different steps of the failure treatment.

a) Failure location.

They both establish, in the course of the derivation, dependency links between variables : Cox as a partition of subterms into unified subterms together with a graph of the subterm relation, L. Pereira as dependency lists attached to variables.

A simpler way to carry out Cox's algorithm on Horn clauses consists in generating on-line subterm partition, but labelling each link introduced between two subterms of the same class with the number (in the graph) of the literal resolution that created the link, thus avoiding much effort searching the subterm graph to find links which are in fact those already used in the partition construction.

Nevertheless it is not very efficient to work on such a partition, which is actually a set of graphs. We prefer to keep these links as linear lists of variable dependencies as Pereira does : each link created between two variables by a unification step is added, with the corresponding number of the literal being solved in the graph, in the list of these two variables, and reported in the list of the variables they are already linked to. As we choose not to keep the state of the list at every

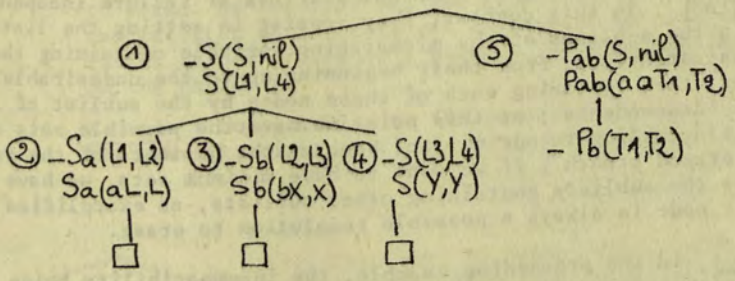
step (since we don't necessarily come back to a previous situation), it becomes necessary to store not only variable dependency but also, as Cox does, the possible instantiation of one of the variables responsible for this dependency : that is to say memorize that X is linked to aY and Y to X at node 2 instead of the links of X to Y and Y to X, in order to locate the exact point of mismatching.

Example : Let's take Pereira's example calculating the intersection of two formal grammars S and P_{ab}.
The insatisfiable set of clauses is the following :

$+S(L,L)$
$+S(L1,L4) - S_a(L1,L2) - S_b(L2,L3) - S(L3,L4)$
$+S_a(a.L,L)$
$+S_a(a.L1,L2) - S_a(L1,L2)$
$+S_b(b.X,X)$
$+S_b(b.X1,X2) - S_b(X1,X2)$
$+P_{ab}(a.a.L1,L2) - P_b(L1,L2)$
$+P_b(b.b.L1,L2) - P_{ab}(L1,L2)$
$-S(S,nil) - P_{ab}(S,nil)$

Here is the plan generated at the failure point at expansion 5.The number of a node actually corresponds to the number, according to execution order, of the resolution unifying the two literals quoted (i.e., in Cox's notation, the number of the replacement arc); we may consider that every literal is affected a number when it is selected for expansion. The failure occurs when attempting to unify the literal

- P_{ab} (S,nil) with P_{ab} (aaT1, T2) in order to solve P_{ab} (S,nil) using the reduction schema of P_{ab} (aaT1, T2) into P_b (T1, T2) ; the dependency lists will exhibit the non unifiability that invalidates this plan.



P_{ab} (S,nil) has become a node by selection for expansion, while P_b (T1,T2) is not yet a node since it is still to solve.

A possible way of implementing this graph is to update an ordered list of literals yet to solve at each literal expansion, the order depending on the selection function, while keeping for each literal, resolved or not, the number of its father (i.e. the resolution that created it). Here this list at the failing point would be : - P_b (T1,T2).

The dependency lists constituting the semantics of this plan are presented with Pereira's notations, with the modification described just before the example ; the lists for T1 and T2 are not given as they are useless for the sequel.

$S \rightarrow abnil\,[IS:[1:[IL1:[2:[aIL2:[3:[b4:UL3_1]]:UL2_2]]:UL1_1]]:[5:[aaIT1:UT1_0]]:US_2]$

$L1 \rightarrow abnil\,[IL1:[1:[IS:[5:[aaIT1:UT1_0]]:US_2]]:[2:[aIL2:[3:[b4:UL3_1]]:UL2_2]]:UL1_1]$

$L2 \rightarrow bnil\,[IL2:[2:[IL1:[1:[IS:[5:[aaIT1:UT1_0]]:US_2]]:UL1_1]]:[3:[b4:UL3_1]]:UL2_2]$

$L3 \rightarrow nil\,[4:[3:[IL2:[2:[IL1:[1:[IS:[5:[aaIT1:UT1_0]]:US_2]]:UL1_1]]:UL2_2]]:UL3_2]$

$L4 \rightarrow nil\,[1:UL4_0]$

As an example, the meaning of the list of S is : the value of S is abnil through link with L1 at node 1, itself linked to L2 at 2, itself linked to bL3 at 3, itself directly linked to nil at 4 ; it is also linked to aaT1 at node 5, which is incompatible with the preceeding value.

b) Failure analysis for one mismatching

The failure occurs when the value of an argument is incompatible with the value it must match with. Cox's calculations automatically give minimum sets of resolutions to erase in order to solve the conflict, thus avoiding further calculations of failure independent nodes [FAH] . In this context, they consist in setting the list of the nodes in the sublists of the mismatching variable containing the conflicting instantiations, from their beginning up to the undesirable instantiations, then replacing each of these nodes by the sublist of itself and its descendents ; at this point we have the possible sets of nodes to erase according to our will of keeping the brothers of the root node of the erased branch ; if we want to have minimum sets, we have to suppress the sublists containing other sublists, as examplified below. The last node is always a possible resolution to erase.

Example : in the preceeding example, the incompatibility being due to the mismatching of b in S introduced through 1,2,3, with the a introduced at 5, the list of responsible nodes is 1,2,3,5 ; as the sons of a node must be erased too, the possible sets of nodes to erase are (1,2,3,4), 2,3,5. The first set can disappear as a non minimal set, or at least be tried as a last possibility.

c) Backtracking.

In the case of a failure on several arguments, it is necessary to combine the failure analyses if we want to eliminate them all together. The process would be to look for a common set of nodes to erase, and if no such set exists, to take the set containing one set from each argument and rooted by the nearest ancestor of the roots of these two sets.

Three choices remain to make after the set of possible erasable subsets of nodes has been determined :

C_1 : the set of nodes to erase : it seems reasonable to try all the clauses for the literal which was solved the last before analysing the failure, to eliminate this trivial possibility. It is possible to order the sets as Pereira does, but in this context erasing the most recent nodes loses interest since we don't have to erase all the work done afterwards ; at least we can try a minimal set first.

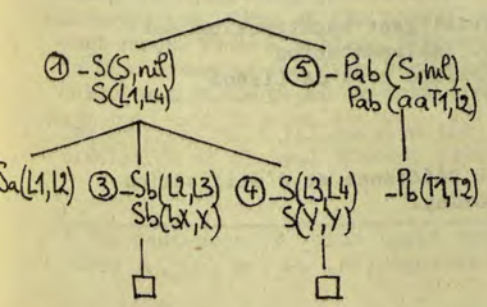
C_2 : the next literal to solve : it can be the literal that rooted the erased branch, but it could be another one too, according to forward execution strategy : for example a literal that had been given a higher priority in the producer-consumer schema.

C_3 : the next solution to this literal : further analysis of the conflict source could allow to avoid another failure : for example keeping which instantiations to avoid at this point may eliminate choices of clauses. If no other alternative is available for this literal, it is possible to choose another set of nodes to erase, or as a last chance the father of this literal, and so on.

Another possibility for all these choices is to let the user specify them by means of metarules if needed.

The problem is then to update the plan and the corresponding dependency lists. The last thing is easy since it is sufficient to suppress all the sublists beginning with the number rooting the branch to erase. As for updating the graph, it is sufficient to add to the current list of literals yet to resolve the literal of this root node (which is now erased as a node) : its place in this list will depend on the choice C_2 .

Example : Suppose that in the previous example we decided to erase 2, the new situation after this suppression would be :



$S \rightarrow aaVT1[IS:[1:[IL1:UL1_1]]:[5:[aaIT1:UT1_0]]:US_2]$

$L1 \rightarrow aaVT1[IL1:[1:[IS:[5:[aaIT1:UT1_2]]:US_2]]:UL1_1]$

$L2 \rightarrow b\,nil\,[IL2:[3:[b\,4:UL3_1]]:UL2_2]$

$L3 \rightarrow nil\ [4:[3:[IL2:UL2_2]]:UL3_1]$

$L4 \rightarrow nil\ [1:UL4_0]$

The list of literals to resolve is now - Sa (L1,L2), - Pb (T1,T2), which are kept with their father nodes, respectively 1 and 5.

Conclusion :

It seems that much work has been done on failure location. But much remains to do concerning failure analysis and treatment, i.e. to find an intelligent way of restarting the derivation after a failure ; general strategies can be applied in conjunction with further analysis of the conflict and informations memorized from the forward execution control ; they don't preclude the use of metarules which would be very effective to determine which set of nodes to erase, even possibly whether we must keep some of the parallel branches developed after the erased nodes, and what new solution try in a particular application.

Bibliography

[COX] Philip T. Cox : Deduction Plans : a graphical proof procedure for the first-order predicate calculus.
Department of Computer Science, University of Waterloo
Research Report CS-77-28 - October 1977

[FAH] Ali A. FAHMI : Contrôle de systèmes de déduction automatique fondés sur la logique.
E.N.S.A.E. 10 Av. E. Belin - 31055 TOULOUSE CEDEX
Thèse de docteur ingénieur - November 1979

[GAL] Hervé GALLAIRE - Claudine LASSERRE : Controlling Knowledge deduction in a declarative approach.
E.N.S.A.E., 10 Av. E. Belin - 31055 TOULOUSE CEDEX
IJCAI - 79 - Tokyo

[PER1] Luis M. PEREIRA - Antonio PORTO : Intelligent backtracking and side tracking in Horn clause programs - The theory
Departamento de Informatica - Universidade nova de Lisboa
Report n° 2/79 CIUNL - October 1979

[PER2] Luis M. PEREIRA - Antonio PORTO : Intelligent backtracking and sidetracking in Horn clause programs. Implementation
Departamento de Informatica - Universidade nova de Lisboa
Report n° 3/79 CIUNL - December 1979

[ROU] Philippe ROUSSEL : Prolog : manuel de référence et d'utilisation groupe d'IA - Université Marseille-Luminy.

THE RELATION BETWEEN SEMANTIC TABLEAUX AND RESOLUTION THEOREM PROVERS.

KRYSIA BRODA
Imperial College, London SW7.

Abstract.

The semantic tableau method as presented in Beth [2] can be modified by the inclusion and use of dummy variables (Prawitz [11]). With this modification, the tableau method begins to show similarities with resolution theorem provers. Moreover a procedural interpretation can be given to the tableau so that one can see a resemblance to Horn clause theorem provers (for example). This paper investigates the relation between semantic tableaux and some clausal theorem provers.

1. Introduction.

The ideas presented in this paper derived from a consideration of a theorem prover for logic programs using first order logic * (SF) based on semantic tableaux [8] . The initial motivation was to write an interpreter in ICPROLOG [7] for proving inconsistency of a set of sentences in SF.

The algorithm used in the program was based on the semantic tableau method (described in sec. 2) modified by the inclusion and use of dummy variables, to facilitate a reasonably efficient search for a proof. The program suffered from some redundancy and as an aid in overcoming this, refutation procedures for clausal sets of sentences were studied with the idea that the better aspects of each could be combined, extended to SF and then incorporated into the theorem prover. It emerged that all the procedures studied could be simulated within the semantic tableau format and could be easily and intuitively understood with reference to the tableaux.

As a natural sequel to these studies, progress has been made on completeness proofs for SL-resolution, Horn clause theorem provers and the connection graph proof procedure, obtained from consideration of certain tableaux. When the procedure for SF has been finalised (i.e. when useful features of the clausal theorem provers have been incorporated) a proof of its completeness will be attempted.

It is appreciated that work in this area has been or is being done by others (Bibel [3],[4], Andrews [1], Bowen [5]) but their work is from a different viewpoint. Bibel has presented a systematic procedure which enables the inconsistency of a set of sentences to be proved but the method is limited, at present, to sets of sentences in which only one instantiation of each sentence is required. Andrews has extended the method (although it does not necessarily derive a "most simple" proof as does Bibel's) to arbitrary sets of sentences, but he does not seem to have related it to resolution theorem provers (except connection graphs). Moreover, neither of these presentations directly relates the proof to a semantic tableau nor do they use its structure to justify the operations involved.

For the sets of sentences investigated in the aforesaid papers, a heuristic would often generate the same or a similar tableau. Logic programs are very often recursive and typically require the use of only a few sentences many times to solve a problem, and it seems that heuristic guides will be more tractable for generating useful inconsistency proofs than a systematic procedure. This is at present being investigated. Many of the operations of SL-resolution etc. can be seen as heuristic when viewed from a tableau stance. A procedural interpretation, similar in spirit to that given to top down Horn clause theorem provers can be given to the tableau method and it is hoped to make use of this in the heuristics used.

The sections that follow give the general background and terminology, illustrate simulation of clausal theorem provers and detail current work.

---

* The sentences in which logic programs are written may include the logical connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$ together with universal and existential quantifiers.

2. Preliminaries.

The semantic tableau method, augmented with dummy variables is applied to a small example, to illustrate its features and introduce some terminology. No attempt at optimisation is consciously made or indeed at generating a "most natural" tableau.

We wish to show that a given set of sentences $\mathcal{A}$ in SF logically imply ($\Longrightarrow$) some sentence in a set of sentences $\mathcal{B}$. (In logic programs $\mathcal{A}$ are the procedures and $\mathcal{B}$ (usually only one sentence) the goal). Loosely, $\mathcal{A} \Rightarrow \mathcal{B}$ iff every interpretation which makes every sentence in $\mathcal{A}$ true also makes at least one sentence in $\mathcal{B}$ true.

We make use of a semantic tableau:
Assume $\mathcal{A} \not\Rightarrow \mathcal{B}$ and attempt to derive a contradiction; i.e. assume there exists an interpretation in which all sentences of $\mathcal{A}$ are true and all in $\mathcal{B}$ are false; look for the interpretation and if it is found impossible that such an interpretation can exist, conclude $\mathcal{A} \Rightarrow \mathcal{B}$.

Each branch of a semantic tableau gives a list of sentences which, as a consequence of the assumption that all sentences in $\mathcal{A}$ are true and all in $\mathcal{B}$ are false, must also be true. (i.e. a possible interpretation is being enumerated). Thus in particular every branch will include $\mathcal{A}$ and $\neg\mathcal{B}$. If at some stage we have a sentence and its negation in a branch, we are requiring an interpretation to include (or imply) both a sentence and its negation - an impossibility. Thus the branch in which this occurs cannot represent a possible interpretation and may be terminated (closed). If all branches in the tableau are closed in this way, no interpretation is possible in which all the sentences in $\mathcal{A}$ are true and all in $\mathcal{B}$ are false. Hence we conclude $\mathcal{A} \Rightarrow \mathcal{B}$.

We illustrate the semantic tableau method in example 1 in which the problem is to show that {1} is a subset of {2,1}.

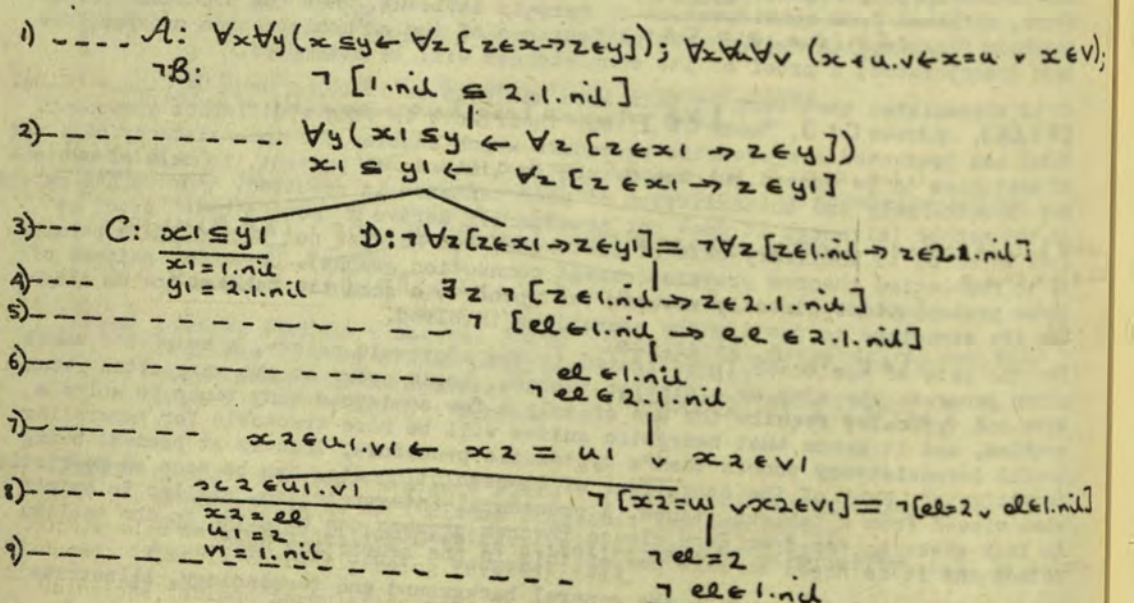


Fig. 1. (Note: "≤", "∈", " = " are predicate symbols.)

Notes for example 1. (illustrated in fig. 1.)

1) Assume each sentence of $\mathcal{A}$ true and $\neg\mathcal{B}$ true in the interpretation we are searching for (I).

2) Example of a simplification rule: If a sentence of the form $\forall x\ P(x)$ is true in I, then $P(X)$ is true in I for any individual X in the domain of I. Thus we have (possibly) many true sentences implied by $\forall x P(x)$. We are aiming to close each branch and for efficiency sake would like to do so as soon as possible. But we may not know the best individual to choose so we replace the quantified variable by a dummy variable and instantiate the dummy later, when a suitable constant is found. Dummy variables used here are xl and yl.

3) Another simplification rule: If a sentence of the form $X \leftarrow Y$ is true in I then either X or $\neg Y$ is true in I. In this example (see fig.1), I must be such that $\mathcal{A}, \neg\mathcal{B}, \mathcal{C}$ are true in it or $\mathcal{A}, \neg\mathcal{B}, \mathcal{D}$ are true in it. (We now have two possible ways of finding the interpretation I).

4) We close a branch when it represents a set of requirements which no interpretation may satisfy. Here, instantiating xl,yl to 1.nil and 2.1.nil respectively enables the branch to be closed (by "matching" with $\neg\mathcal{B}$). The substitutions are also made for xl,yl in the other branch. (Indicated by the symbol "$=$" or sometimes "$\|$").

5) Another rule: $(\neg\forall z \equiv \exists z \neg)$. $\exists z P(z)$ asserts that there exists an individual in I - let us name it 'el' - such that P(el) is true. Notice, we must choose a name that has not been used before.

6) Another rule: $\neg(X \rightarrow Y)$ true in I means both X and $\neg Y$ must be true in I.

7) Use of the "$\forall x P(x)$" rule (for x,u,v respectively).

8) Use of the "$\rightarrow$" rule and closure of a branch.

9) The branch is closed because it contains $el \in 1.nil$ and $\neg el \in 1.nil$.

There are other possible tableaux, notably the one which simulates a case analysis bottom-up from $el \in 1.nil$ (as opposed to working top down from $\neg el \in 2.1.nil$ (at 6 in fig.1)). We need extra sentences in $\mathcal{A}$ for this- namely the "only if" half of the second clause in $\mathcal{A}$ of fig.1 and $\forall x \neg x \in nil$. The details are not difficult to supply.

Some "obvious" heuristic improvements come to mind. e.g. (a) quantifiers should be moved as close to the quantified variable occurrences as possible. With this modification universally quantified parts of sentences can be repeated in a branch with different instantiations and other parts of the sentences, which only need occur once in a branch will not be repeated. (b) Sentences and their descendants may be labelled for bottom-up only or top-down only use - thereby enabling transformation rules to be used. With this modification one uses sentences only in the direction intended by the programmer and does not generate irrelevant or unusable information.

We require some rules for using the simplification rules. With the constraint that one does not pursue a branch single-mindedly one can choose any branch to develop and any sentence in that branch to simplify. (It seems obvious that one does not instantiate a sentence with the same constant more than once in any branch - for if one did repetition would occur).

Note that once all instantiations have been made we have a classical closed tableau; one of (in general) an infinite set.

3. Simulation of a Top Down Horn Clause Interpreter.

Example 1 used "human knowledge" to guide the search for a closed tableau. Certainly, if the sentences involved are Horn clauses then one can dispense with human intervention. We can simulate various features of a PROLOG theorem prover: selection of arbitrary literals in the goal clause; co-routining; backtracking; negation as failure; some loop checking. Not all of these can be properly described here, but it is hoped that the following examples will give a flavour of the ideas involved.

Example 2. This example (a very simple finite state acceptor) illustrates the top down nature of a PROLOG proof and arbitrary goal selection. (Two orders of choice are given in figs 2a and 2b). Depth first evaluation is employed (i.e. each branch is developed to its closure before developing another branch at the same or higher level) but one need not do this. Simulation of co-routining is thus possible by developing branches in a quasi-parallel way. In this example , if the goal were

←reach(s,a.b.a.a.nil) which fails, co-routining can judiciously be used to reduce the search space.e.g. compare a depth first tableau, always choosing append first and a tableau which co-routines between append and reach.

(Note the program is not the simplest that could be written to carry out the task).

The classical tableaux represented by figs 2a and 2b can be obtained by "filling-in" the instantiations. Fig 2c illustrates this for fig 2b.

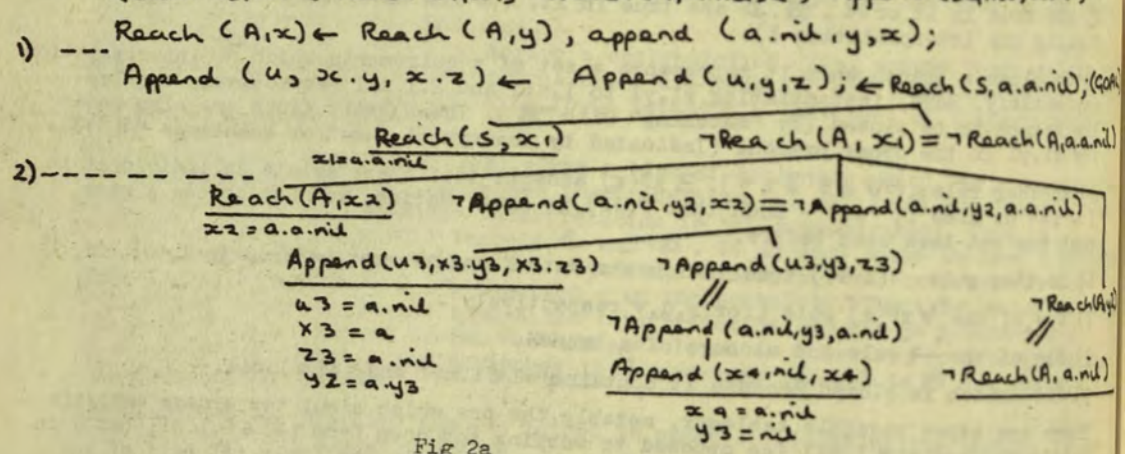


Fig 2a

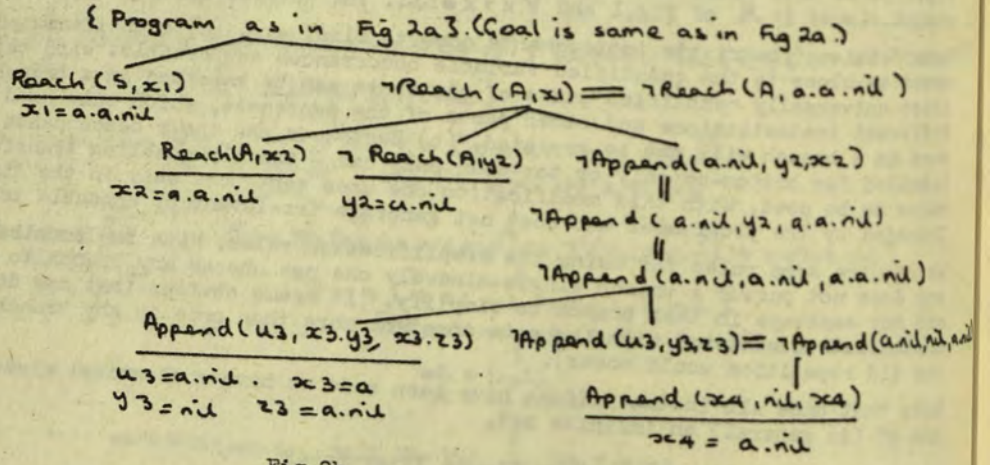


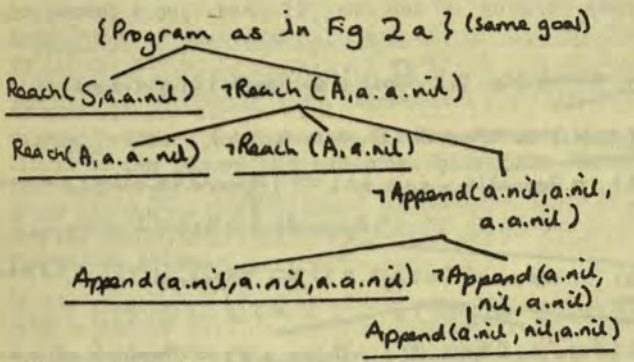
Fig 2b

Notes for example 2 (illustrated in figs 2a,2b,2c,2d)

1) Assume all clauses are appropriately universally quantified and are all true in an interpretation (say I). The same program is used in figs 2b,2c,2d and is not repeated there. The tableau is generated left to right depth first and clauses are chosen in the order given in the program.

2) Previously the "or" simplification rule dealt with the binary case only. Since "v" is associative we can write (say) $a \vee b \vee c$ as $a \vee (b \vee c)$ and hence develop the tableau pattern of fig 3a. This is equivalent to that of fig 3b as all universal quantifiers are at the left of a clause and replacement by dummy variables takes place before splitting.

3) At each stage the tips of the unclosed branches represent the resolvant (goal) clause in the LUSH proof. The LUSH proof simulated in fig 2b is given in fig 2d in a more usual notation. (The chosen literal at each stage is underlined).



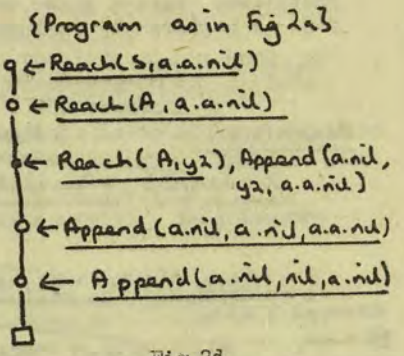


Fig 2c

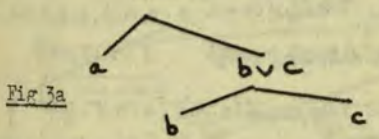
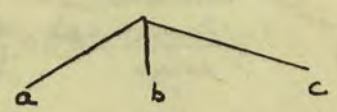


Fig 3a

Fig 3b

Fig 2d



Example 3 (illustrates simulation of backtracking)

Notes for example 3 (illustrated in fig 4a)

1) The goal, ←append(a.nil,nil,a.a.nil) fails and in PROLOG we would backtrack to the last available alternative. To simulate this we repeat the clause in which the alternative occurs, closing branches up to the failed literal in the same way as previously. (e.g. at (A) in fig 4a). We then make an alternative choice at the relevant literal. (In fig 4a, instead of "matching" ¬reach(A,y2) with reach(A,a.nil) we match it with the head of the recursive clause for reach.)

The tableau of fig 4a is redundant. Since we know that the portion below the failed append literal cannot use that literal in its closure (there would not have been a failure if it could) we can overlay the clause with the failed literal with the portion of the tableau below it. The reult (for fig 4a) is shown in fig 4b. This new tableau corresponds to a successful path through the LUSH search space.

Various other backtracking strategies might be tried, as long as care is taken not to lose the potential of using every clause in every branch, instantiated in all possible ways (unless a branch should close).

One can simulate the backtracking without the duplication of other literals in the clause to which the backtracking is made, by moving the universal quantifiers as close to their variables' occurrences as possible. Backtracking will then be to the predicate with the innermost quantified variable which still has alternatives. The result is a more natural simulation. Lack of space prevents the inclusion of an example.

Example 4 (an example using 'negation as failure' (symbol NOT) illustrated in fig.5)

(It is assumed the reader is familiar with the idea of negation as failure in logic programs (Clark [6]).

The same principle of using negation in logic programs can be applied to the tableau method: if we generate a node of the form ¬[NOT P] we replace it by ¬P and only close the branch if the tableau formed below ¬P cannot be made to close. (In the procedural interpretation to show NOT P we try to show P. If we fail we conclude NOT P). The reader should now be able to simulate such a PROLOG proof using the ideas in the previous examples. This process illustrates the use of a meta-rule which could be dispensed with by including suitable predicates and procedures in the given

set of sentences. (After all, here we are not restricted to Horn clauses). e.g. if the clause were A ← NOT B we could replace it by A,B← and we would need denials of the form ←B to solve the problem. This represents a different procedural interpretation: "assume B and see what happens" which may not give such a natural argument as negation by failure.

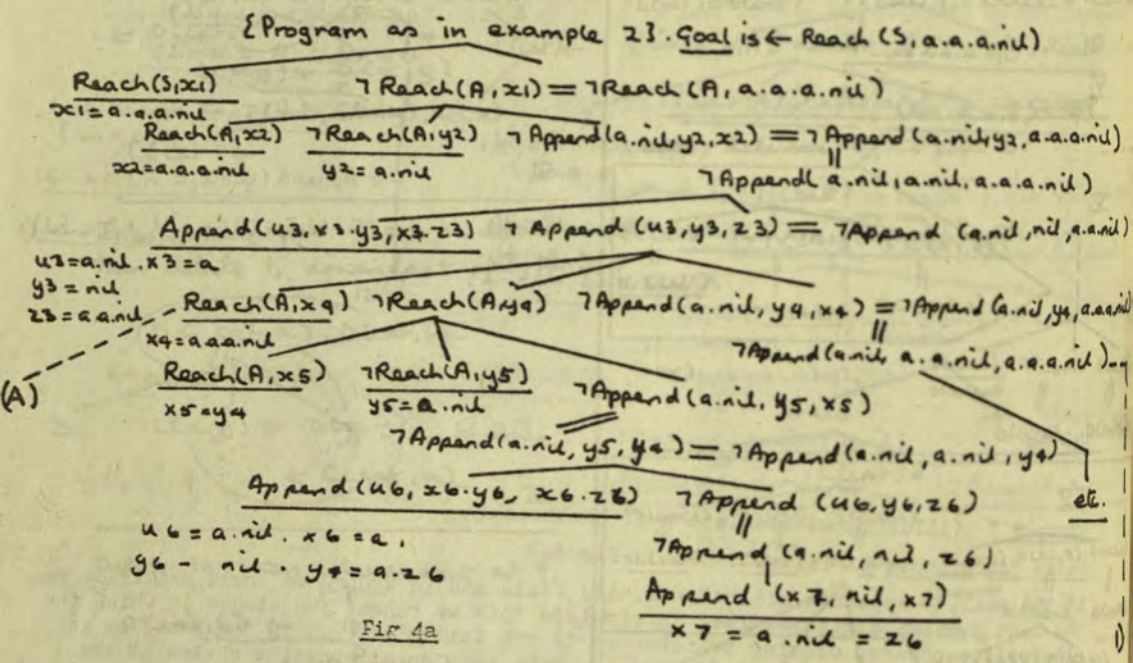


Fig 4a

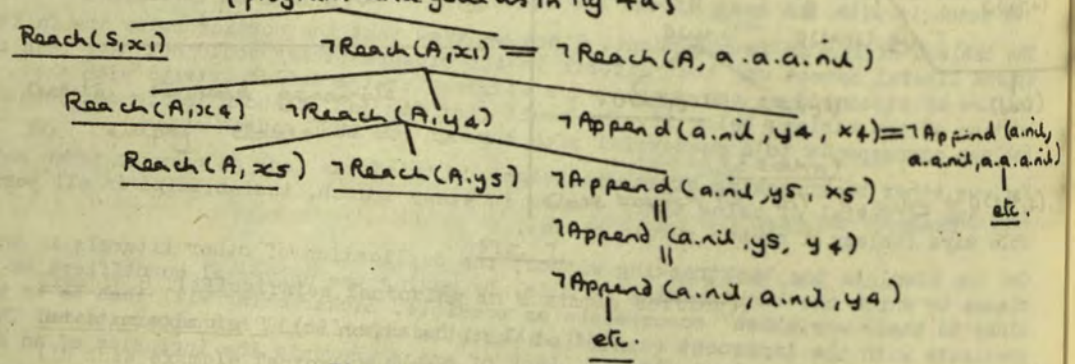


Fig 4b

Other such metarules dealing with substitution aspects of equality might be introduced. (This is currently being investigated).

Using the tableau formulation some loop checking is easy to make. Certainly, if a negative literal occurs in a branch twice, such that the second occurrence is subsumed by the first, then a potential loop may be recognised and dealt with.- by retreating to the first occurrence and trying another alternative if one exists. Other redundancy checks may or may not be possible, depending on how much of the tableau is overlayed (and hence lost) during simulation of backtracking.

In this section we have described how various aspects of PROLOG-like theorem provers may be simulated by a semantic tableau. Next we will deal with simulation of other clausal theorem provers.

4. Simulation of SL-Resolution and other Clausal Theorem Provers.

(Familiarity with SL-resolution [9], connection graphs [9] and Shostak's graph construction procedure [12] is assumed in this and the following sections.)

Example 5 (The operations of ancestor resolution, merging (factoring) and deletion of literals in SL-resolution are illustrated).

One possible SL-proof is given in tableau form in fig 5a and in a more usual notation in fig 5b. An alternative proof is given in figs. 5c and 5d. See also the explanation below.

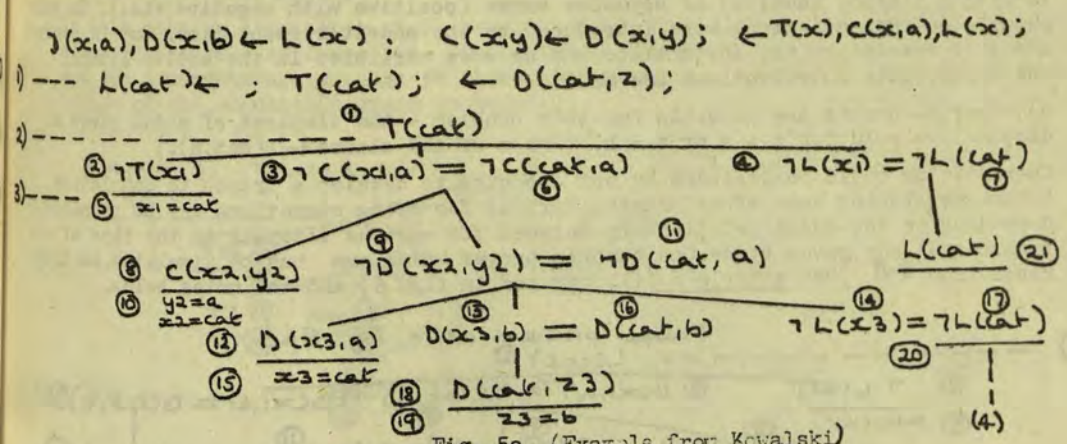


Fig. 5a (Example from Kowalski)

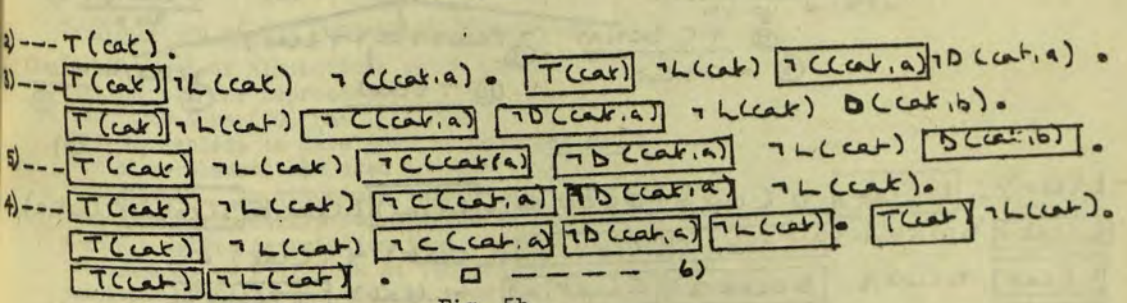


Fig. 5b

Notes on example 5 (illustrated in figs. 5a, 5b, 5c, 5d).

1) Assume suitable universal quantification of sentences. Nodes and closures are generated in ascending order and in a depth first manner.

2) The start clause is 'T(cat)←'. In fig.5b each stage is separated by a fullstop.

3) Closure of a branch corresponds to making a passive literal active. At (3) T(cat) is made active. In fig 5b an active literal is enclosed in a box. Unboxed literals represent passive literals (or tips of the unclosed branches of the tableau).

4) Merging has taken place at (20). We can close the branch at (20) since whatever closure can be developed below (7) can be developed below (17). (Any literal above (7) which is used in the closure below it is common to the branch containing (17)). In general this operation is one of factoring. As long as the literal "higher" in the tableau (7 here) is subsumed by the "lower" literal (17 here) any closure developed from the higher literal can be transferred to the lower literal (which is the more general).

In the alternative notation used in fig. 5b factoring can occur if a passive literal (say A) is seen to be the same as or to subsume a previous literal (passive). The literal A can be boxed.

5) When all branches below a literal have been closed that literal can be removed. (The SL-chain is truncated). The literal can no longer be used within allowed SL-operations and will have no passive literals following it.

6) Every literal has been truncated and we have a contradiction. (i.e. a completely closed tableau).

7) Start clause is 'L(cat)→' in figs. 5c and 5d.

8) If a literal matches a literal higher in the tableau (in the same branch) it is acceptable in the tableau method to close the branch. In SL-resolution this is called ancestor resolution. A passive literal is made active (boxed) by "matching" it with a literal (active) of opposite sense (positive with negative etc.). In this example no instantiations were introduced by the ancestor resolution. But in general ancestor resolution may instantiate one or more variables in the active literal and appropriate substitutions are made.

9) Other SL-proofs are possible for this problem - the simplest of which gives a disjunctive solution 'z = a or z = b.' (The z of the clause ← D(cat,z).)

Co-routining could be included by not choosing to develop a branch to completion before developing some other branch. Various factoring operations may be possible depending on the exact relationship between the various literals at the tips of the branches at any given stage. One theorem prover which can thus be simulated is LUSH-resolution [10] .See example 6 (illustrated in fig. 6) and the notes below.

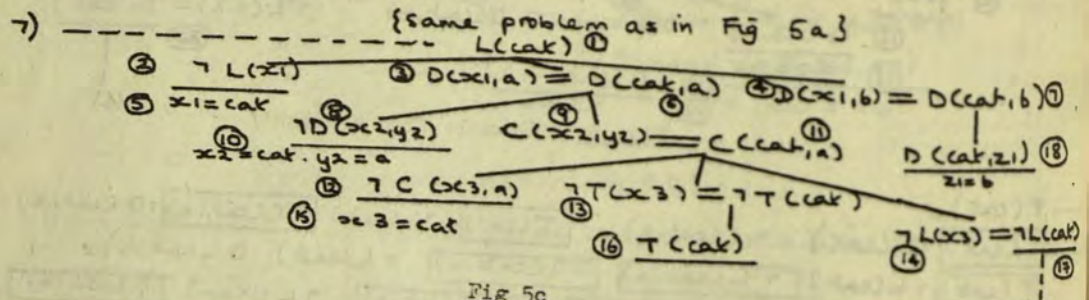


Fig 5c

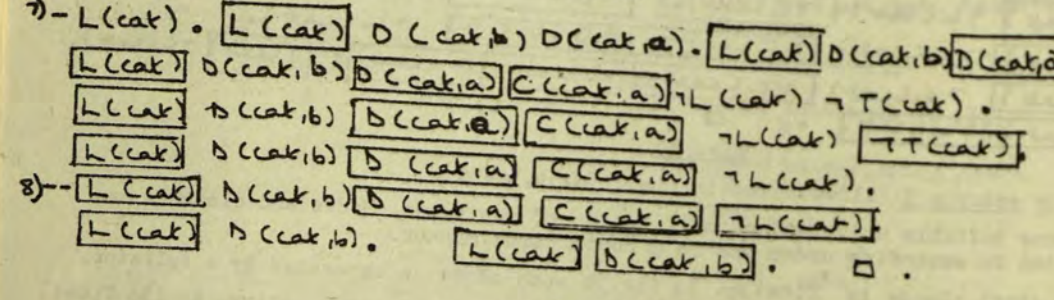


Fig 5d

Notes on example 6.

1) Notice, a propositional example for simplicity. Start clause is 'x,z←'.

2) Rather than finish developing branch from node (2) we switch to the branch from node (1) ..... and then back again. In a more general setting, where instantiations are made this is likely to be a useful strategy (as is co-routining in ICTRCLOG).

3) We have to be careful about using factoring. We cannot close the branch at 18 using the argument that the closure at (14) can be developed at (18) since the closure at (14) may use nodes not common to the branch ending at (18).

Shostak's graph construction procedure (GC) and Prawitz' splitting rule for matrices can be obtained using arguments similar to those used in this section. We briefly give the argument for GC.

In SL-resolution, when all branches below a literal have been closed, knowledge of having "proved" that literal is lost. In the special case of the closure below the literal having used only nodes below and including the literal, we know that if the same literal (or one which subsumes it) occurs elsewhere in the tableau the same closure can be used. To facilitate this we add the negation of the literal to other branches in the tableau. The easiest way to do this is to add the negated literal to the top of the tableau. In a less specialised case the closure may have involved literals (L say) in the branch above the literal (K say) being considered. We can add the negated literal to all branches which have literals L in common with the branch containing K. In the worst case this will be those branches which have the same immediate predecessor as K.

In this section simulation of SL-resolution, LUST-resolution and the graph construction procedure were described. Simulation of other theorem provers is possible and the arguments are similar to those used here. It remains to illustrate simulation of the connection graph procedure.

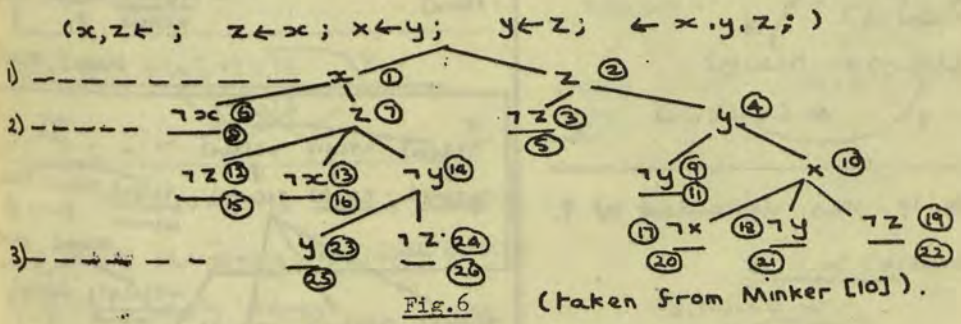


Fig.6 (taken from Minker [10]).

5. Simulation of the Connection Graph Proof Procedure.

In all previous simulations which were of a depth first nature the "latest" resolvant was the one represented by the ends of the unclosed branches. The connection graph procedure (CG) has , in general, several active resolvants and to simulate this the tableau is developed in separate pieces which are combined whenever a pair of resolvants are used as parent clauses. The book-keeping of CG still has to be performed. The structure imposed on CG by the tableau enables one to see when factoring is necessary for a proof and when it is only necessary for efficiency. A more complete description than is given here is in another paper which is in preparation. We will look at two examples. The CG format is useful for the visual recording of links and will be used as well as the tableau.

Example 7 (illustrates a simple simulation with no factoring).

Notes on example 7 (illustrated in fig. 7)

1) The operations of deleting incompatible links and deleting a clause which has a literal with no links (and the special case of pseudo-links) can be justified in terms of the tableau being generated.

2) When we have deleted link 2, the resulting tableau (the part of V in a box) represents the resultant '←C(cat,a)' and the original clause 'L(cat)←'. The latter still has a link (3) and can be used again. When we delete link 3, rather than repeat the clause 'L(cat)←' we generate a 3-dimensional tableau (indicated by the dotted line in V). We now have two tableau joined at 'L(cat)'.

3) First we delete (10) and generate the tableau of VI within the box. We now delete a link (the descendant of 9) which is between two resolvants. In this case it is an easy operation - the 3-dimensional tableau is "flattened". Notice we still have the clause 'D(cat,a),D(cat,b)←' represented in the tableau. This part could have

been deleted if the clause had a literal with no links.

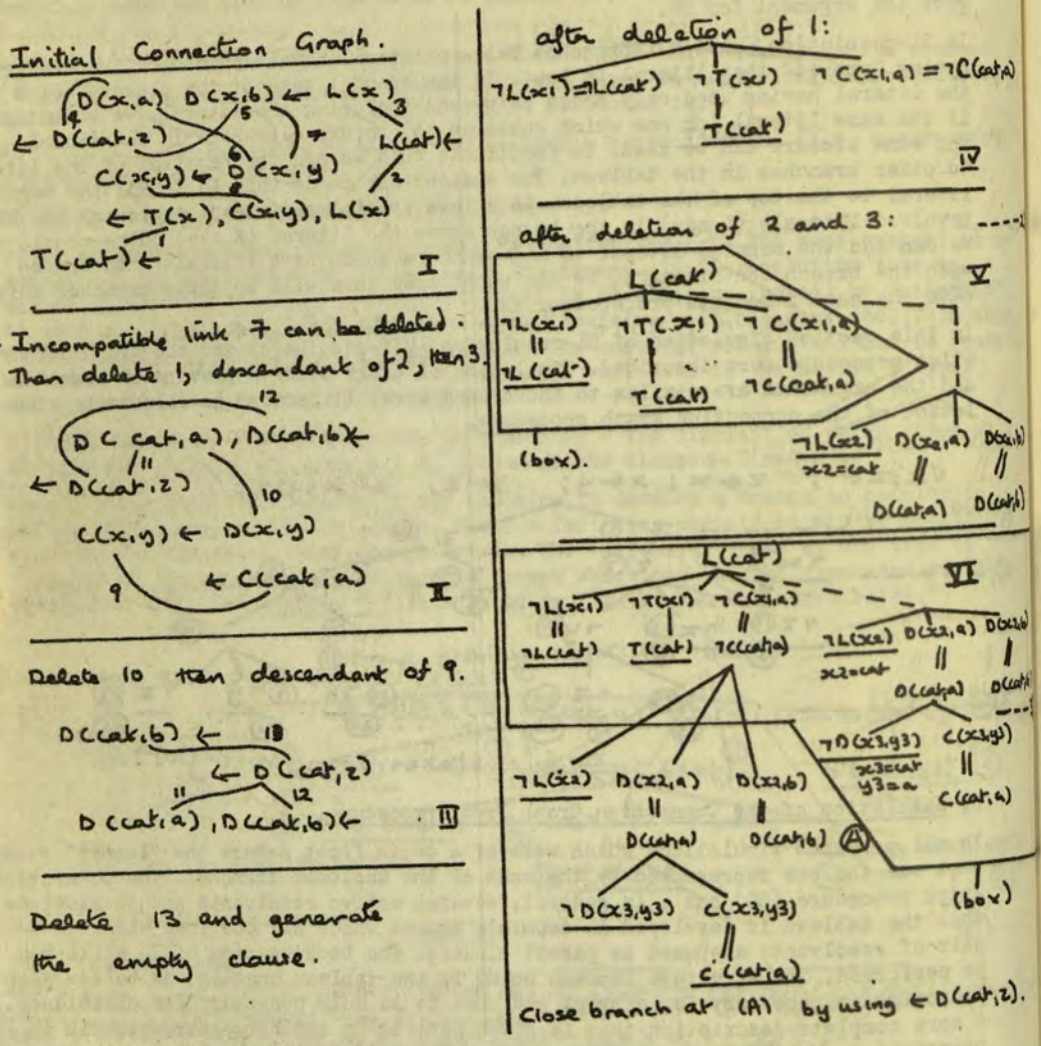


Fig. 7

Example 8 (illustrates factoring in a simple setting).

Notes on example 8 (illustrated in fig. 8).

(In this example factoring is not necessary, although it is useful for a more efficient proof. If however the clause '←P(a)' were '←P(a),P(u)' (say) factoring would be necessary). The factoring is indicated in the tableau by being able to resolve the literal '¬Q(v1)' in the first resolvant with the original clause 'P(x),Q(x)←'. The Q literal is represented in the tableau at (A). This resolution produces two copies of 'P(x1)' which effectively are factored in the tableau. Certainly, it may be the case when this phenomenon occurs that factoring is not necessary. But we can be sure that factoring is only necessary when it does occur. (Note: there are a few special cases with regard to some sorts of tautological links where factoring is necessary before deleting all tautological links. However these cases can be recognised from the tableau format and hence dealt with).

The tableau generated in fig.8 is not the only tableau corresponding to the CG proof. An alternative is shown in fig. 8VI. The difference is due to the asymmetry of the tableau with respect to the parent clauses in a resolution. This asymmetry does not occur in the normal CG format. In fig. 8VI factoring is not indicated by the tableau and hence need not have been performed in the CG proof.

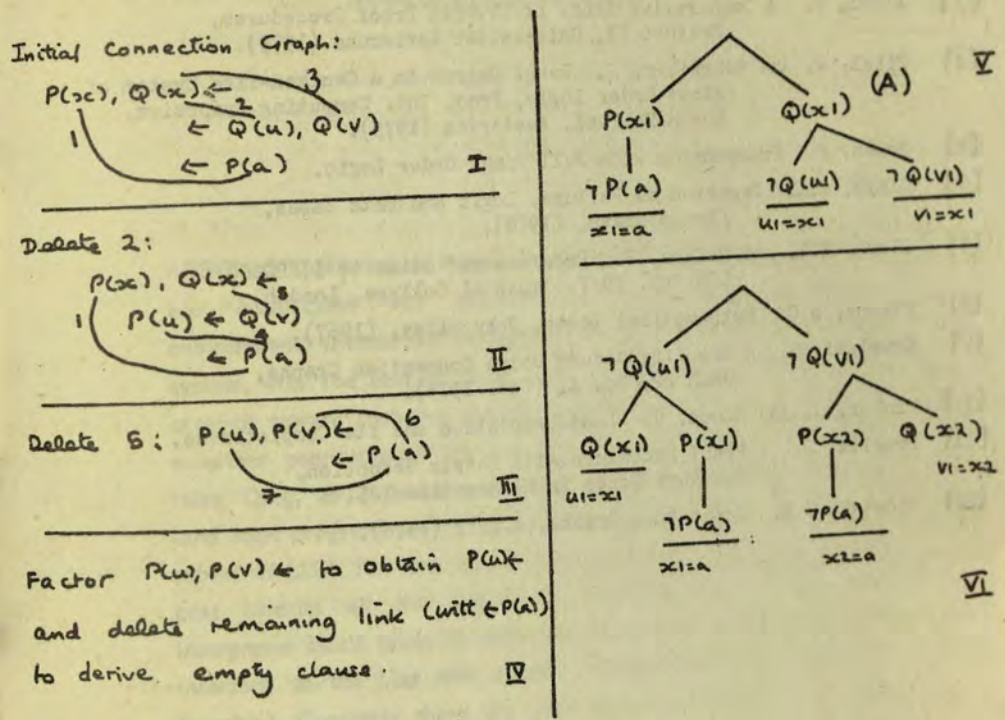


Fig.8

6. Conclusion.

It is seen that a host of resolution theorem provers can be embodied within the classical tableau method.

A paper detailing completeness proofs for various theorem provers based on the semantic tableau is in preparation. The method employed for all but connection graphs involves "pruning" operations on a fully instantiated closed tableau formed in a "systematic" way. (e.g. each clause is developed in each branch, instantiated with each constant). Arguments similar to those used elsewhere in this paper show that the pruning does not destroy closure, and the result is a closed tableau in a format consistent with the theorem prover under consideration. The final closed tableau is "lifted" to complete the proof. The method used for connection graphs is a more direct proof. The final details are currently being worked out.

Regardless of whether the tableau method always derives the "best" tableau, it has a satisfying relationship with a human problem-solving approach. It is hoped to make use of this in a revised interactive version of the original program, by allowing human intervention as necessary in the process of the mechanised prover and consequently bringing improvements.e.g. In the event of the information supplied not being sufficient to complete a proof, th new version might provide scope for a user to add a missing premiss when a branch fails to close.

Another area of exploration is regarding metarules. e.g. How much improvement, if any, is gained by using metarules as opposed to formulating them within SF?

References.

[1] Andrews, P. B. General Matings, Proc. IVth Workshop on Automated Deduction (1979).

[2] Beth, E. W. The Foundations of Mathematics.

[3] Bibel, W. A Comparative Study of Several Proof Procedures, Bericht 82, Universitat Karlsruhe (1979).

[4] Bibel, W. and Schreiber, J. Proof Search in a Gentzen-like System of First Order Logic, Proc. Int. Computing Symposium, North-Holland, Amsterdam (1975).

[5] Bowen, K. Programming with Full First Order Logic.

[6] Clark, K.L. Negation as Failure, Logic and Data Bases, Plenum Press, (1978).

[7] Clark, K.L. and McCabe, F. Programmers' Guide to ICPROLOG, CCD Rep. 79/7, Imperial College, London.

[8] Kleene, S.C. Mathematical Logic, John Wiley, (1967).

[9] Kowalski, R. A Proof Procedure using Connection Graphs, JACM 22, No. 4, (Oct. 1975).

[10] Minker, J. and Zanon, G. Lust-Resolution and its Completeness.

[11] Prawitz, D. A Proof Procedure with Matrix Reduction, Lecture Notes in Mathematics 125.

[12] Shostak, R.E. Refutation Graphs, A.I. 7 (1976).

QLOG - THE SOFTWARE FOR PROLOG AND LOGIC PROGRAMMING

by

H. Jan Komorowski

ABSTRACT: We argue that the existing Prolog implementations are insufficient wrt incremental programming, interactive environment, interactive debugging tools, integrated programming system, etc. The best of them - the Prolog DEC10system - is an attempt toward such the environment but it nevertheless provides a rather poor support for the programmer. Instead we suggest using Qlog, an interactive programming environment for Prolog (and logic programming) which has been implemented in a portable subset of LISP. The new system is very efficient and with minimal cost inherits all the support of the host Lisp system. The interpreter itself takes 10 pages of pretty printed code, while the interface to the Lisp host system occupies about 20 pages (for Interlisp). Currently there are Qlog versions in Interlisp, Fortran Lisp F3 and Stanford Lisp 1.6. Timing for Qlog in Stanford Lisp 1.6 and Prolog DEC10 on the same DEC10 (KL 10E) computer shows that code is executed slightly faster in Qlog.

Keywords and phrases: Prolog, Lisp, logic programming, interactive programming environment, program debugging and testing, embedded languages.

This work was sponsored by the Swedish Board for Technical Development under contract 77-4380b.

Informatics Laboratory
Linköping University
S-581 83 Linköping
Sweden

ACKNOWLEDGMENTS: Mats Carlsson and Martin Nilsson implemented respectively Fortran Lisp F3 and Stanford Lisp 1.6 versions of Qlog. They have contributed to debugging and cleaning up the basic implementation.

Jim Goodwin has patiently assisted during the Interlisp implementation and provided several ideas about the strategy and tactics of embedding languages in Interlisp so as to obtain maximum support at minimum cost: what we have begun to call "the law of maximal embedding".

Sten-Åke Tärnlund discussed with me several implementation issues and invited me to write this paper for the Logic Programming Workshop in Budapest.

Professor Daniel Chester of the University of Texas at Austin reviewed an early version of the report, and his criticisms were of great help to me in revising it.

Finally, I am very indebted to Erik Sandewall whose constant supervising of my research provided several fruitful ideas.

CONTENT

1.0 INTRODUCTION

The best and most well known implementation of Prolog < 1 > is the Prolog DEC10 system < 2 >. That implementation has been a pioneer work toward an interactive environment for logic programming. The task to implement it was very big but unfortunately only partially succeeded. The system is interactive but for several reasons uninvitingly hard to use. We had a rather big experience with Interlisp < 3 > and would like to provide a similar environment for Prolog.

There were two possible approaches: use the existing Prolog implementation and design Interlisp inspired facilities in it, or take Interlisp (or any other modern Lisp environment) and embed Prolog in it.

The first approach was possible but would require an enormous effort. The code for Interlisp packages is several hundred pages. Working from scratch would mean to repeat it for Prolog.

The second approach attracted us very much. By the use of the "law of maximal embedding" < 4 > we expected to design an interactive environment for Prolog with minimal cost. The embedding law says: implement special facilities for a given language only when necessary, and embed in Lisp otherwise. This minimizes the costs of the implementation, while maximizing the inheritance of Lisp language features (like input/output routines) and Lisp system facilities (like the editor and file librarian). The embedded approach usually results in very few pages of code. In our case the functions of the interpreter itself consist of 330 lines of pretty printed code (the text files have about 10 pages of pretty printed code). The interface to Lisp facilities usually takes more code than the interpreter. The figure for Interlisp is 20 pages of pretty printed code.

2.0 WHY PROLOG DEC10 IS INSUFFICIENT

A user who switches from "batch" oriented computation as in Prolog Marseille < 5 > is certainly very pleased by the new environment of Prolog DEC10. However the system may only satisfy an unexperienced programmer who never tasted a flavor of a fully interactive programming environment. The MACLISP < 6 > and INTERLISP are good examples what are such the environments.

2.1 The requirements on a language for an interactive programming system

Let us characterize what should be the requirements on a programming language for integrated, interactive programming system. With some modifications we quote from Sandewall < 7 > and immediately relate to Prolog DEC10. (We shall write PD10 for Prolog DEC10.)

BOOTSTRAPPING. The system should be implemented itself in the language it supports; PD10 - yes.

INCREMENTALITY. To achieve real interaction, the basic cycle of the programming system should be to read an expression from the user, execute it, and print out the result while preserving global side effects in its database; PD10 - the basic cycle: yes, although there are no tools for examining side effects in the database of programs.

PROCEDURE ORIENTATION. Obvious reasons; PD10 - yes.

INTERNAL REPRESENTATION OF PROGRAMS. Since most of the operations required by the programming system are operations on programs, the language should make it as easy as possible to operate on programs; PD10 - yes.

FULL CHECKING CAPABILITY. All possible input from the user must result in rational response from the system; PD10 - NO!

DATA STRUCTURES AND DATABASE. The system must minimally have data structures that are able to represent programs as tree structures and a database facility where one can conveniently store and retrieve properties of items. For example, what are the procedures currently defined by the user; PD10 - data structures: yes; database facility: no.

DEFINED I/O FOR DATA STRUCTURES. In order to test a procedure interactively, one wants to be able to type in a call to the procedure and obtain back the result. Since the arguments and/or the result may be data structures, I/O for data structures must be defined in the system. Since programs are internally stored as data structures, this I/O may also be used as parser and program-printer; PD10 - the I/O is very poor and it forces the user to define his/her own routines.

HANDLES AND INTERACTIVE CONTROL. The actions taken by the system in specific situations should be controllable by the user in such a way that a user-defined procedure (a "handle") can be inserted instead of the original procedure provided by the system. For example, such handles are useful for the operation applied to expressions input by the user, and reactions to errors and exceptional conditions during the execution of a procedure.
Also, the system must allow for an assortment of different control signals that may be typed-in by the user at arbitrary times to control the ongoing computation. The "killer" interrupt, which terminates the interactive session and returns to the operating system, is exactly what the user does NOT want. The response to control signals should also be user-controllable through handles.

PD10 - the system has a possibility to interrupt the execution without returning to the operating system, but the interrupt is not programmable. The worst is the elimination of errors in PD10 which leads to bizarre, hard to analyze computations.

In summary, the Prolog DEC10 language fulfills only partially the requirements on a programming language for an interactive environment. It (and any Prolog) has however one very important property: programs are represented as Prolog data. This property makes it possible implement Prolog in Lisp.

2.2 The requirements on an interactive programming system

Let us now consider what are the requirements on an interactive programming system. The view of advanced program development and maintenance tools < 8 > in such the environment is centered around a few main concepts:

1. Program development is an incremental process.

2. Programs are stored in a database not as strings of characters but in some structured form.

3. The programmer must have available several advanced tools when developing and maintaining programs. This includes tools for:
    a. program specification
    b. program debugging and testing
    c. program presentation and modification
    d. program analysis and transformations
    e. program verification
    f. program generation
    g. program documentation

Here we focus our attention on tools named in points 3.b, 3.c, and 3.g.

3.0 PROGRAM DEBUGGING AND TESTING

The interactive programming in a system based on a language with pattern-directed invocation and backtracking requires several properties for a programming system. First of all calls to undefined functions must be properly handled.

1. If the definition of the function FOO contains a call to the function FIE, but FIE is not actually called for a certain argument vector x to FOO, then the programming system should be able to operate and to compute FOO(x) even if FIE has not yet been defined.

2. If FOO calls FIE as in the previous case and FIE is undefined, but computation of FOO(y) leads to a call of FIE, then the programming system should make a "soft landing". In other words, it should not print an error message and abort, but rather preserve the current environment and allow the programmer to inspect the situation, decide on a suitable assignments that FIE could have accomplished, type it into the programming system, and let the computation continue.

The first property is of course available in every interpreter. The second one is not provided by any other Prolog implementation than Qlog. The "soft landing" is in fact the point of the interactive testing and debugging. After it happens, the user has a spectrum of tools. He/she can provide a definition for this yet undefined function. He can also decide that the function fails and simulate this case by typing FAIL to the system. The decisions are supported by several facilities. Namely, the user may examine the current environment by looking at the formal arguments' binding, the AND/OR tree structure of the computation, the stack, the procedures about to be computed, and the pattern which has just succeeded in the unification. It is also possible to change actual bindings of variables.

An intended (programmed) interrupt followed by the "soft landing" is a modification of the second case. In Lisp terminology it is called "break". The user simply informs the system that when a function FOO is called he wants to examine the environment prior to the execution of FOO.

A simplified variant of breaking is tracing. In the trace the function (which the user wants to trace) is broken, the current bindings are printed, and the computation is automatically continued without any further action from the user.

The Prolog DEC10 system offers quite different philosophy. Since the semantic is totally defined, there are no errors nor programmed interrupts (breaks). The execution of a program might be halted in random, but since no tools are given for examining the environment these halts are nearly useless.

4.0 PROGRAM PRESENTATION AND MODIFICATION

The collection of procedures which are currently in the database must be presentable to the user. He wants to know what functions are currently in the program and he may want to examine their text. The natural requirement is a pretty printer which displays a selected procedure in some system defined format.

During the debugging process a lot of changes are done in the text of procedures. For these purposes an intelligent editor must be provided, i.e. an editor which understands the structure of programs and works on their tree representation, not on the strings of characters. Usually the editor is in core and is written in the same language. Another solution is the editor in parallel job. It requires however a sophisticated operating system.

5.0 PROGRAM DOCUMENTATION

An interactive system needs a file librarian. Among other things the librarian knows what functions have been introduced during the interactive session, and which have been changed. He asks the user where the new ones should go, and takes care of updating the existing files, creating the new ones, producing the compiled files, etc. He also should be able to answer questions like: "Where is the function FOO?". In programs with several hundred functions distributed over several files this is an important facility.

6.0 THE IMPLEMENTATION OF QLOG

As we have earlier pointed out Prolog has equality between data and programs. This property makes implementation easier. As an implementation principle we applied the "law of maximal embedding", i.e. implemented special facilities for Qlog when necessary, and embedded in Lisp otherwise < 9 >. This minimizes the costs of implementation drastically. The inherited services are far larger than Qlog itself, far better than anything one could have afforded to build from scratch, and far better than the facilities built from scratch in other Prolog implementations.

6.1 Data types

No special finesse is required to embed Prolog data in Lisp data. They are similar enough that Qlog can just use an appropriate subset of S-expressions directly. Thus Qlog obtains for free allocators, a garbage collector, READ and PRINT, a list structure editor and in general all utilities which are defined over arbitrary S-expressions.

6.2 Prolog control structure and variable binding

Since the Prolog control structure and variable binding mechanism are quite different from that of Lisp, they involve harder design decisions. The use of FEXPRs, i.e. functions which do not evaluate their arguments, and the introduction of a special stack for the non-recursive Prolog control structure allow representing Qlog functions directly as Lisp functions. Thus Qlog forms and functions are simply a subset of those of Lisp. This allows us to bind Qlog variables arbitrarily (through the introduction of a special stack), while still inheriting most of Lisp's numerous form and function oriented programming tools.

6.3 The inherited programming tools

Thanks to the implementation method we inherited almost for free a lot of Lisp programming tools. We have the testing and debugging tools of Lisp: trace, break, error handling, visible stacks, current binding environment, etc. Some of them are inherited directly; a few need some cosmetics like variable binding display since it is rather different from the Lisp one.
The file librarian, I/O routines, and the structure editor are taken as they stand.
Our worst cases are the pretty printing routine and the compiler. Because the Lisp pretty print formats are quite different from that of Qlog there is a special pretty printer consisting of 36 lines of code.
The regular Lisp compiler may be used on Qlog forms. The gain is however relatively small. The Lisp compiler does not know anything about the Qlog structure; in general case a true pattern matcher compiler is required. (Of course one can use the Prolog DEC10 compiler if accessible). A Lisp pattern matcher compiler is also a sine qua non for correct comparisons of Lisp and Prolog DEC10 compiled code. In < 10 > the designers of Prolog DEC10 compared Prolog to Lisp in a very unfavorable case for Lisp. The program for symbolic differentiation is strictly a problem of pattern matching. The Lisp program is not compiled with the

respect to this pattern matching whereas the Prolog one is.
Instead of a Qlog compiler we have developed an incremental indexing of functions entries by their first arguments. The method is similar to Prolog DEC10 one, but is superior in that it works for the interpreted code and the new assertions might be added arbitrarily.
In addition, the Interlisp system provides several more advanced facilities like the history package and the spelling corrector which work fairly well for Qlog functions. They needed a little more work to interface with the regular Qlog.

Totally, the interface in Interlisp case takes only 20 pages of pretty printed code. As a rough estimate, the Lisp code for the parts of the Interlisp programming environment which we use is about 50 times larger! We guess that implementing them in Prolog DEC10 would result in approximately the same size.

7.0 CONCLUSIONS

We have shown that that the possibility of partially embedding Prolog in Lisp is very powerful system design technique. The point of maximal embedding is that the designer does not have to spend most of his time (re)programming the environment facilities for Prolog. Instead, he makes a set of design decisions and inherits major parts of the Lisp system. This method is very important since it drastically cuts the costs and time expenses required for the development of an interactive system.

The volume of the code is several times smaller than any other implementation of Prolog. This factor is very important for obvious reasons, e.g. debugging the interpreter, introducing modifications, working on a minicomputer version of Lisp, etc.

Since by embedding Prolog in Lisp we did not depart from the implementation environments (cf. Interlisp, Lisp 1.6 < 11 >, and Fortran Lisp F3 < 12 >), the Qlog programming system may be extremely easily used for experiments with logic programming

while retaining the same interactive programming system. Some examples are: new search strategies, other pattern matcher, parallel processing, and many others.

7.1 Comparison to the Prolog DEC10 system

The speed of execution was not our primary concern; it was more important to get a very good programming environment. To our pleasant surprise the resulting interpreter runs slightly faster than Prolog DEC10, both on the same DEC10 computer (KL 10E). The average times for benchmarks programs in Qlog Lisp 1.6 and Prolog DEC10 show that the Prolog average time was 798 milliseconds and the Qlog was 760 milliseconds. The test program resulted in 10x27=270 function calls and the cost of timing was subtracted from the results shown by the Prolog timing function (about 18 ms).

The comparison of interactive environments is very favorable for Qlog. The Prolog DEC10 is interactive and incremental, but the actual data base of functions is virtually impossible to control or display. The i/o routines are very low level, allowing only input of characters (READC) or Prolog terms. RATOM and READLINE must be programmed by each user. The trace feature is of the wallpaper type, i.e. one can trace all functions at once or none!

Instead of error handling, Prolog DEC10 offers the philosophy that errors are impossible. This means that the system treats all errors, from misspellings to undefined functions, as pattern matching failure. As < 13 > promises, there are no "incomprehensible error messages". Indeed there are no messages at all; the program does not even stop. We can hardly agree, however, that "this totally defined semantics ensures that programming errors do not result in bizarre program behaviour." The discussion about consistency between theoretical basis for Prolog and the implementation of the language is just not a relevant response to the user who has lost hours of work finding

typographical mistakes which can be routinely detected by the interpreter.

Prolog DEC10 has no facilities like break, pretty printer, file librarian, nor any support for editing either in core or in parallel jobs. The compiler exists as a separate package but is rather inflexible and uninvitingly hard to use.

8.0 SUMMARY

Qlog is a portable Lisp implementation of Prolog. It already exists in Fortran Lisp F3, Stanford Lisp 1.6 and Interlisp < 14 >.

Qlog inherits most of the major components of the Lisp programming environment at very low cost, and obtains a high quality programming environment. Interfacing the existing Interlisp facilities to the new language required 30 to 50 times less code than the Lisp facilities themselves require.

Brand new development-time facilities for languages with pattern directed invocation of functions were invented.

Lisp itself has been complemented with pattern directed invocation of functions, a unification pattern matcher, and an associative data base with richer structure than property lists, all for less then 30 pages of pretty printed Lisp code.

The Prolog language has inherited a high quality interactive programming environment for a very low price. If this new implementation attracts serious interest, Prolog can reach a much broader audience, the whole Lisp community. In fact this is already happening. The Qlog system is installed at MIT Artificial Intelligence Laboratory and there is an implementation of Intermission < 15 > (an actor system in Prolog) in Qlog. At the same time Prolog can use Lisp machine < 16 > what we think is of a very big advantage.

9.0 REFERENCES

< 1 > Kowalski, Robert, A., "Algorithm = Logic + Control" Comm. ACM, July 1979, Vol. 22, No. 7.

< 2 > Pereira, L.M. et al., User's Guide to DECsystem-10 Prolog (Provisional Version) Divisao de Informatica Laboratorio Nacional de Engenharia Civil, Lisbon, April 1978.

< 3 > Teitelman, Warren, Interlisp Reference Manual, Xerox-Palo Alto, Calif., 1979.

< 4 > Goodwin, James W., Komorowski, H. Jan, "System Design by Embedded Sublanguages: A Lisp Craft Tradition", to appear.

< 5 > Roussel, P. "Prolog: Manuel de reference et d'utilisation", Groupe d'Intelligence Artificielle, Marseille-Luminy, Sept. 1975.

< 6 > Moon, David, "Maclisp Reference Manual", MIT AI Laboratory, 1975.

< 7 > Sandewall, Erik, "Programming in an Interactive Environment: the "Lisp" Experience", Computing Surveys, Vol. 10, No. 1, March 1978.

< 8 > Wilander, Jerker, Haraldsson, Anders, "Proposal for Programming Environment Laboratory" in: "Software Systems Research Center", research proposal, Linköping University, March 1980.

< 9 > Komorowski, H. Jan, Goodwin, James W., "Embedding Prolog in Lisp: An Example of a Lisp Craft Tradition", to appear.

< 10 > Warren, D.H.D., et al., "Prolog - The Language and its Implementation Compared with Lisp", in: Proceedings of the symposium on Artificial Intelligence and Programming Languages, SIGPLAN/SIGART Special issue, pp. 109-115, Rochester 1977.

< 11 > Quam, Lynn H. and Diffie, Whitfield, "Stanford Lisp 1.6 Manual" Stanford AI Laboratory, Operating Note 28.7

< 12 > Nordström, Mats, "Lisp F3 Users Guide", Datalogilaboratoriet, Uppsala University, June 1978.

< 13 > "How to Solve It with Prolog", A tutorial, Laboratorio Nacional de Engenharia Civil, Lisboa, Agosto de 1979.

< 14 > Komorowski, H. Jan, "Qlog Interactive Environment - The Experience from Embedding a Generalized Prolog in Interlisp", Informatics Laboratory, Linköping University, August 1979.

< 15 > Kahn, Kenneth M., "Intermission - Actors in Prolog" Stockholm University, April 1, 1980.

< 16 > Weinreb, Daniel, Moon, David, Lisp Machine Manual, MIT AI Laboratory, November 1978.

Intelligent backtracking and sidetracking in Horn clause programs - implementation

Luis Moniz Pereira

António Porto

Departamento de Informática
Universidade Nova de Lisboa
1899 Lisbon, Portugal

December 1979

Abstract

This is the second report of our work on intelligent backtracking and sidetracking strategies in Horn clause programs.

This second part is a description of three interpreters (written in Prolog) for Prolog programs, working on the basis of the theory described in the first part [Pereira et al. 1979]. They offer practical information about the features of the new strategies.

The first is a general-purpose interpreter which uses intelligent backtracking instead of the standard blind one.

The second is a specialization of the first, for database query only, which uses intelligent backtracking in a much more restricted way, although sufficient for Prolog relational databases satisfying a reasonable set of assumptions.

The third is an interpreter working through sidetracking, which by its very nature is not extensive to the full language without some predefinitions. The use of cut, notably, does not make sense without a fixed order of execution of goals. In a future paper, however, some control constructs will be presented to achieve an effect similar to the cut.

Structured Programming Techniques in Prolog

Simon B. Jones
Computing Laboratory,
University of Newcastle upon Tyne

In this paper I intend to compare the programming facilities available in predicate logic and functional programming languages. I will examine a simple dialect of Prolog (Kowalski and others [1,2,3]), directing my attention to the capability for structured and advanced programming techniques.

The work arises from a background of practical programming with a higher order, functional language (Lispkit [4]). Lispkit is very much like ISWIM [5]. The capability of having variables and function definitions which are local to expressions, and the support of higher order functions (functions which accept functions as parameters and/or return functions as results) combine to give the language great expressive power in the representation and structuring of textually large and computationally complex programs.

Prolog provides a fascinating alternative formalism for the representation of programs. Key features contributing to the programming power of Prolog are computation directed by pattern matching, built in nondeterminism, multi-directional computation (the input and output roles of parameters may be altered without modification of the program), and the facility to create data structures whose component values may be unknown at the time of creation.

However, these features are concerned solely with the detailed specification of a computation "in the small". Prolog provides no assistance with the organisation of programming "in the large".

Conventional programming disciplines require a programmer to abstract and modularise his program with the objective of improving its clarity and manageability. Common disciplines include hiding the implementation details of complex computations, abstracting common expressions (giving rise to loca variables), and abstracting common patterns of program construction (giving rise to local function definitions and higher order functions).

These techniques are facilitated in Iswim by the devices of "where clauses" which introduce local names for subexpressions and functions, and by higher order working enabling functions to be represented by a construction, behaving like a λ-expression, which can be treated subsequently as a "first class citizen" just like any other value in the program. "Where clauses" give programs a natural nested structure, and scope rules limiting the use of variable and function names define the modularisation of a program.

Prolog, on the other hand, does not permit a nested program structure, since all the program clauses are defined at the "same level" and each predicate name is of global scope. Local variables may be introduced within program clauses to represent intermediate data structures, and indeed they seem indispensable in this role, but local program clauses cannot be defined, and clauses (or groups of clauses) cannot be treated as first class citizens.

The purpose of this paper, then, is to put forward some proposals for structured and higher order extensions to the basic Prolog language. The extensions, motivated by aesthetic and practical programming reasons, are designed to bring Prolog up to par with higher order functional languages.

A Basic Dialect of Prolog

I wish to illustrate and discuss my extensions to Prolog, and their properties, in terms of some concrete language representation in order to avoid any unnecessary vagueness. To this end I will introduce a simple dialect of Prolog, unadorned by any fancy features, which I will subsequently refer to as "basic Prolog". The language should be very familiar, it is just that described by Kowalski in [1].

A basic Prolog program consists of a goal clause (line 1 in the example below), and a set of program clauses (lines 2, 3, 4, 5 below). Here is a common example, a program which may be used to reverse a list built using the constructor cons:

1. ← reverse(cons(1,cons(2,...cons(..,nil)...)),Result)

2. reverse(nil,nil) ←

3. reverse(cons(X,L1),L2)←reverse(L1,L3),append(L3,cons(X,nil),L2)

4. append(nil,L,L)←

5. append(cons(X,L1),L2,cons(X,L3))←append(L1,L2,L3)

The goal clause is a set of subgoals (or conditions, or procedure calls) - this is represented as a list of subgoals in the text. Each subgoal specifies a parenthesised list of argument terms, and the name of a predicate (or procedure) which is to be satisfied by the values of those arguments. There is only one condition in the goal clause on line 1 of the example; the procedure name (entirely in lower case) is "reverse", and there are two arguments.

An argument term (or simply term) is either a variable (initial letter in upper case), a constant (underlined), or a constructed term "cons(t1,t2)" where "cons" is the constructor (or function name, or functor) and t1,t2 are themselves terms. Hence the arguments of the single condition in line 1 above are, firstly, the constructed term "cons(1,cons(2,..cons(..,nil)..))" which is supposed to represent some particular list (1,2,...), and, secondly, a variable "Result".

The computational resources of a program are contained in the set of program clauses (or predicate definitions, or procedure definitions). Each program clause (or simply clause) specifies a particular relation on its argument terms, defined by an implication of the form "consequent←antecedent". The consequent contains the procedure name of the relation being defined, and a parenthesised list of argument terms which are to be related. The antecedent may be empty, as in lines 2 and 4 of the example, or it may contain a set of conditions (represented textually as a list) which must be satisfied in order that the arguments are in the defined relation, for example lines 3 and 5.

I now require a semi-formal tool with which to describe the semantics of basic Prolog. An appropriate choice here is an "abstract interpreter" scheme following the "procedural semantics" of basic Prolog given by Kowalski [1]. The abstract interpreter described below will assist greatly in defining and understanding the Prolog extensions later on.

A basic Prolog program is separated into its goal clause gc (a non-empty set of conditions), and its program clauses pc (a non-empty set of program clauses). From gc construct gc_0 by subscripting each variable in gc with a zero. Each variable in pc must be unsubscripted.

The result of interpreting such a program is zero or more substitution instances gc_{res} of gc_0, in which the variables of gc_0 have been replaced by new terms (or may have remained unchanged). The meaning of each such gc_{res} is intended to be that the resulting argument values satisfy the conditions of the goal clause as defined by the program clauses pc.

Each result gc_{res} is related to gc_0 as follows:

The abstract interpreter constructs (or discovers otherwise) a sequence of goal clauses starting with gc_0

$$gc_0,\ gc_1,\ gc_2,\ \cdots\ gc_n$$

and a sequence of substitutions (each associating terms with subscripted variables)

$$\theta_1,\ \theta_2,\ \cdots\ \theta_n$$

with the following four properties:

1) $gc_0,\ldots,gc_{n-1}$ are not empty, they contain at least one condition to be satisfied.

2) gc_n is empty, it could be read as "Halt".

3) Each substitution θ_m relates goal clauses gc_{m-1} and gc_m: For some condition c in gc_{m-1} and clause p from pc, with the predicate name in c matching that in the consequent of p,

$$c \in gc_{m-1} \quad p \in pc \quad \text{predicate of } c = \text{predicate of } p$$

generate a variant p' of p by subscripting each variable in p with m,

$$p' = \text{variant } m \text{ of } p$$

and obtain θ_m by <u>unifying</u> the argument list of c with the argument list of the consequent of p',

$$\theta_m = \text{unify (arguments of c, arguments of p')}.$$

Unification, which is described in detail elsewhere by Robinson, Clark and others [ ], produces a substitution which makes the corresponding arguments of the two lists equal.

4) gc_m is obtained by removing c (from 3) from gc_{m-1}, adding the set of conditions which is the antecedent of p' (from 3), and substituting

through by θ_m,

$$gc_m = [gc_{m-1} - \{c\} + \text{antecedent of } p'] \theta_m .$$

The result substitution instance gc_{res} is found from gc_0 by first substituting with θ_1, then θ_2 and so on until θ_n,

$$gc_{res} = [gc_0] \theta_1 \theta_2 \cdots \theta_n \quad .$$

Note that in property 3 I am assuming a judicious choice of p for each c, in order to avoid a failed unification (if at all possible). In other words, I am only interested here in describing "successful executions" of the interpreter.

Local Clause Definitions

The first step that I will take in extending basic Prolog is a simple one, but it will significantly alter the abstract interpreter. The extension gives Prolog the essential nesting structure in which I am interested, and eases the transition to a more useful variation of the idea which I will introduce in the next section.

In a nutshell, the general idea is to allow any set of conditions (the main goal clause of a program and the antecedents of program clauses) to have a local, privately accessible set of program clause definitions. Syntactically I will delimit a set of conditions and local definitions by curly brackets, with the keyword where prefixing the set of definitions. If there are no local definitions then the syntax will be as before. An antecedent containing local definitions I will call a compound antecedent.

Consider the example program clauses containing four local definitions in two compound antecedents:

```
1.   p1(...)←{ q1( ), q2( ), p2( )
2.              where  q1( )←....
3.                     q2( )←.... }
4.   p2(...)←{ q1( ), p1( )
5.              where  q1( )←....
6.                     p1( )←.... }
```

My intention with respect to scope rules for procedure names is that they should be similar to the rules in conventional block structured languages. The procedure names in a condition set are to be resolved within the set of local definitions (if there are any), otherwise, if a procedure name has no match in the local set (or there are no local definitions), then the name is resolved in successively more enclosing sets. Hence an inner clause definition with the same procedure name as any outer definitions makes the outer definitions inaccessible, and clause definitions in the same set may be recursive or mutually recursive.

Thus in the example, where the two clause definitions are assumed to appear in the same local definition set, the following scope resolutions hold: In line 1, q1 matches q1 in line 2, q2 matches q2 in line 3, and p2 matches the p2 definition in lines 4, 5, 6. In line 4, q1 matches q1 in line 5, and p1 matches p1 in line 6 (and not the p1 in lines 1,2,3).

With this extension a Prolog program can be seen to have the structure of a single goal clause qualified by a set of local definitions.

The practical utility of the modified Prolog is that it enables groups of predicate definitions which are interdependent, or which contribute to the description of a particular definition, to be collected together and isolated (both textually and semantically) from the remainder of the program within a compound antecedent. In terms of programming discipline this amounts to hiding implementation details.

For example, when coding the efficient form of the reverse clauses it may be desirable to hide the extra argument which is required:

```
reverse(L1,L2)←{ rev(L1,nil,L2)
        where rev(nil,L,L)←
              rev(cons(X,L1),L2,L3)←rev(L1,cons(X,L2),L3)}
```

Now I must provide a more precise meaning for the extended basic Prolog. I will do this by describing a transformation into basic Prolog, and by showing how the abstract interpreter can be modified to perform this transformation "dynamically".

To transform an extended basic Prolog program into basic Prolog it is necessary to bring all the nested program clause definitions up through levels of definition until they are local to only the main goal clause of the program. During this process the accidental introduction of program clauses with the same name, but which should be distinct, must be avoided. The easiest way to ensure this is to add subscripts to predicate names in a controlled manner. In the description of the transformation I will assume an understanding of the predicate name scope rules by using a phrase such as "All occurrences of the predicate name p" to mean only those occurrences associated with the definition of p by the scope rules.

The transformation proceeds in n steps, where n is the total number of local definition sets in the program.

In step 0 the goal clause gc and its local definitions pc are separated. Each predicate name in gc is subscripted with 0. Each predicate name defined at the outermost level of pc, and all occurrences of these names throughout pc, are given the subscript 0.

In each of steps 1 to n-1 one level of nesting is removed from one definition in pc. At step i the program clause definitions which are immediately local to the compound antecedent of some outermost level definition in pc are themselves brought to the outermost level, with their defined predicate names and all their occurrences subscripted with i. Following step n there should be no nested definitions left in pc.

Transformation of the program

```
← { p( )
  where p( )←
        p( )←{ r( ), s( ), p( )
               where r( )←
                     s( )←  }
  }
```

gives the goal clause

$$\leftarrow p_0( )$$

and the following sequence of program clause sets:

Step 0

$p_0( )\leftarrow$

$p_0( )\leftarrow$ { r( ), s( ), $p_0( )$
where r( )←
s( )← }

Step 1

$p_0( )\leftarrow$

$p_0( )\leftarrow r_1( ), s_1( ), p_0( )$

$r_1( )\leftarrow$

$s_1( )\leftarrow$

The goal clause and the program clauses of step 1 form the desired basic Prolog program.

The abstract interpreter is easily modified to handle extended basic Prolog. Again nested program clause definitions will be raised from their nested locations, but only as required. When a program clause with a compound antecedent is selected for unification with a condition from the goal clause, its local definitions are instantiated and are included as separate definitions in the program clause set. Only instantiated program clauses may be selected for unification.

Each predicate name and variable in the program's goal clause is given the subscript 0, giving the initial goal clause gc_0.

The program clauses which are immediately local to the goal clause have their predicate names, and all occurrences of these names, subscripted with 0. This gives the initial program clause set pc_0.

The abstract interpreter constructs a sequence of goal clauses

$$gc_0, gc_1, gc_2, \cdots gc_n$$

a sequence of program clause sets

$$pc_0, pc_1, pc_2, \cdots pc_n$$

and a sequence of substitutions

$$\theta_1, \theta_2, \cdots \theta_n$$

As before $gc_0, \ldots gc_{n-1}$ are not empty, and gc_n is empty.

θ_m, gc_{m-1} and gc_m are related as before, selecting a condition c from gc_{m-1} and a program clause p, this time from pc_{m-1}:

$c \in gc_{m-1}$ $\quad p \in pc_{m-1}$ predicate of c = predicate of p

p' = variant m of p

θ_m = unify (arguments of c, arguments of p')

$gc_m = [gc_{m-1} - \{c\} + \text{conditions of antecedent of } p'] \theta_m$

Generating variant m of p is more complex than earlier. Each variable in the arguments and conditions (if any) of p is given subscript m. Variables within the local definitions in p are not altered. However, each local definition has its predicate name given subscript m, and all occurrences of these names throughout p are similarly subscripted. The local program clauses are thus given new predicate names, ready for instantiation by the addition of these clauses to pc_{m-1}:

$$pc_m = pc_{m-1} + \text{local definitions of antecedent of } p'$$

The resulting substitution instance of the goal clause is found as before:

$$gc_{res} = [gc_0]\theta_1 \, \theta_2 \cdots \theta_n \ .$$

Relaxing Scope Rules for Variables. Structured Prolog

In basic Prolog each variable in a program clause is strictly local to that clause and may be used only within the consequent and antecedent of the clause. If the same variable name occurs in more than one clause then each clause has its own distinct variable - this is ensured by the subscripting strategy of the abstract interpreter.

In the extended basic Prolog presented in the previous section, variables are still local to the clause in which they appear, but the rule is stricter because the scope of variables does not extend into any local clause definitions which may be present. The consequent and conditions of a program clause contain only its local variables. Again, the abstract interpreter enforces these properties.

It is often desirable to be able to relax these strict scope rules. For example, consider the following program clause for performing some complex operation on a table:

```
tableop(Table,Result)←
        { ..... p(Table,V,W), .....
        where  p(Table,V,W)← ... lookup(Z,Table,V),...
               ....
               lookup(X,T,Y)← .... T ....
               ....
        }
```

The clause p(Table,V,W) is designed to check that V and W are related, somehow, by Table. However, p is not interested in the structure or contents of Table, it merely passes Table on to lookup. It would be convenient to be able to drop Table from the arguments of p. This makes sense since the programmer probably has in mind that Table is of global significance within the tableop clause.

I will permit such simplifications to be made by the inclusion of explicit information in a program to indicate that the scope of a variable may cover a wider range than simply its local consequent and conditions. This is in contrast to the implicit "local scope" rule which both basic and extended basic Prolog have. Such explicit information will take the form of an import list of variable names appearing before any program clause. The meaning of each variable name in an import list is that occurrences of that variable name immediately outside and immediately inside the clause are to be treated as occurrences of same variable.

For example in the program clause

```
p(...X)← { ...X...
           where ...
                 (X):q( )← ...X...
           }
```

the variable X is explicitly imported into q from the surrounding environment, nd all the occurrences of "X" shown represent the same variable.

Variables may be explicitly imported through several levels of local clause definitions.

The scope rule for predicate names remains the same as that in extended basic Prolog.

With the above modification to the treatment of variables, the coding for the tableop example becomes:

```
tableop(Table,Result)←
      {... p(V,W) ...
      where      ...
            (Table) : p(V,W)← ... lookup(Z,Table,V)...
                      ...
                      lookup(X,T,T)←...
                      ...
      }
```

This new dialect of Prolog I will call "structured Prolog".

Before proceeding to give a more formal description of the meaning of a structured Prolog program, I would like to show an unusual and interesting example clause:

```
pairs(L1,L2)←{  firsts(L1), seconds(L2)
             where  firsts(nil)←
                    (X):firsts(cons(cons(X,Y),L1))←firsts(L1)
                       seconds(nil)←
                    (X):seconds(cons(cons(Y,X),L2))←seconds(L2)
             }
```

The predicate pairsis satisfied by two lists of pairs, L1 and L2, if the first members of each pair in L1 are equal, and are also equal to the second members of each pair in L2. The identity of the value which is to be the same in both lists is "communicated" between the local definitions of firsts and seconds via the variable X, which, unusually, does not appear outside the definitions of firsts and seconds (and it is not imported from a more global environment). However, this is valid structured Prolog, and I must be able to give a correct interpretation for programs of this form.

It is possible to transform a structured Prolog program into a basic Prolog program, but the details are a little tedious and I will not give them here. During the transformation variables which are imported into program clause definitions are added to the argument lists of those clauses, and conditions which call on the clauses have their argument lists similarly expanded. In the general case of mutually recursive clauses it may be necessary to add more variables to the import lists of some or all of the clauses. Care is needed to get the scheme correct. Once the dependence on global variables has been removed, the import lists can be dropped, leaving a program in extended basic Prolog which can be transformed to basic Prolog as I described earlier.

The abstract interpreter for extended basic Prolog may be modified in a straightforward fashion to handle structured Prolog programs. The new abstract interpreter depends on the presence of instantiated (subscripted) variables within the sets of program clauses - previously instantiated variables were only allowed in the goal clauses.

For the initial goal clause gc_0 take the goal clause of the program, and give each variable and predicate name the subscript 0. For the initial set of program clauses pc_0 take the local definitions associated with the goal clause, give each locally defined predicate name (and all occurrences) the subscript 0, give all occurrences (as defined by the implicit scope rule and explicit import lists) of each variable in the outer import lists (if any) the subscript 0, drop the import lists from outer program clauses, and delete all subscripted variables from all other import lists. This has instantiated the main program clauses, and all the variables belonging to the outermost environment.

The interpreter constructs the three sequences as before:

$$gc_0,\ gc_1,\ gc_2 \ldots . gc_n$$

$$pc_0,\ pc_1,\ pc_2 \ldots . pc_n$$

$$\theta_1,\ \theta_2 \ldots . \theta_n$$

To relate θ_m, gc_{m-1}, gc_m, pc_{m-1}, pc_m select condition c and clause p,

$$c \in gc_{m-1},\ p \in pc_{m-1} \text{ predicate of } c = \text{predicate of } p,$$

and generate variant m of p,

$$p' = \text{variant m of p}\ .$$

Generating the variant is again more complex. As before, each of the local definitions within p has its predicate name subscripted m, and all occurrences of these names are similarly subscripted. The variables immediately local to p are those appearing in the consequent and conditions of p which do not already have subscripts, and also those in the import lists of the local definitions in p. The local variables, and all their occurrences in p (as defined by the implicit scope rule and the import lists) are given subscript m, and such occurrences in import lists are deleted. The import lists for the local definitions of p are now empty and are deleted. This completes the generation of p' from p, and substitution θ_m may be found by unification:

$$\theta_m = \text{unify (arguments of c, arguments of p')} .$$

gc_m is constructed by replacing c:

$$gc_m = [gc_{m-1} - \{c\} + \text{conditions of antecedent of } p']\theta_m$$

and pc_m is constructed by adding newly instantiated local definitions, and by substituting through with θ_m since instantiated variables within pc_{m-1} and the new definitions may have been bound by θ_m:

$$pc_m = [pc_{m-1} + \text{local definitions of antecedent of } p']\theta_m$$

The result is, as usual,

$$gc_{res} = [gc_0]\theta_1\ \theta_2 \cdots \theta_n .$$

As an illustration of the working of the abstract interpreter I will consider the following structured Prolog program, which uses the pairs clause given earlier:

```
←{ pairs(cons(cons(1,2),nil),cons(cons(3,X),nil))
 where pairs(L1,L2)←.......
 }
```

The initial goal clause, gc_0 is

$$\leftarrow pairs_0(cons(cons(1,2),nil),cons(cons(3,X_0),nil))$$

and the initial program clause set, pc_0, contains just:

$$pairs_0(L1,L2)\leftarrow \ .....$$

To construct θ_1, gc_1 and pc_1, there is no choice of condition and clause. Variant 1 of the only available program clause is:

```
pairs0(L11,L21)←{ firsts1(L11),seconds1(L21)
                  where firsts1(nil)←
                        firsts1(cons(cons(X1,Y),L1))←firsts1(L1)
                        seconds1(nil)←
                        seconds1(cons(cons(Y,X1),L2))←seconds1(L2)
                  }
```

and unification, giving θ_1, binds

$L1_1$ to cons(cons(1,2),nil)

and

$L2_1$ to cons(cons(3,X_0),nil)

Hence gc_1:

← $firsts_1$(cons(cons(1,2),nil)),seconds(cons(cons(3,X_0),nil))

and pc_1 contains

$pairs_0$(L1,L2)←....
$firsts_1$(nil)←
$firsts_1$(cons(cons(X_1,Y),L1))←$firsts_1$(L1)
$seconds_1$(nil)←
$seconds_1$(cons(cons(Y,X_1),L2))←$seconds_1$(L2)

To construct θ_2,gc_2 and pc_2 I will select the first condition from gc_1 and the third clause from pc_1. The variant is simply

$firsts_1$(cons(cons(X_1,Y_2),$L1_2$))←$firsts_1$($L1_2$)

which gives θ_2 binding

X_1 to 1, Y_2 to 2, and $L1_2$ to nil.

The new goal clause gc_2 is

← $firsts_1$(nil),$seconds_1$(cons(cons(3,X_0,nil))

and pc_2 is found by substituting θ_2 through pc_1:

$pairs_0$(L1,L2)←....
$firsts_1$(nil)←
$firsts_1$(cons(cons(1,Y),L1))← $firsts_1$(L1)
$seconds_1$(nil)←
$seconds_1$(cons(cons(Y,1),L2))← $seconds_1$(L2)

Note that the value 1 of X_1, common to both the $firsts_1$ and $seconds_1$ clauses, has been substituted into these clauses.

The remaining substitutions, goal clauses and program clause sets are constructed in a straightforward manner, with the result that the original unknown, X_0, is bound to the constant 1.

Hence the result, gc_{res}:

← $pairs_0$(cons(cons(1,2),nil),cons(cons(3,1),nil))

Higher Order Extensions and Further Work

An important technique in the design of programs concerns the discovery of common patterns of program construction, and the abstraction of each pattern into a clear description independent of its particular applications. Many languages provide some way of achieving this, for example procedure parameters in Algol type languages. In functional languages the facility takes the form of λ expressions, whose values are functions which can be applied to arguments, and higher order functions, which may take function values as arguments, and may return function values as results.

That this technique may be of some use in Prolog can be seen from the pairs example in the previous section. There are two locally defined predicates, firsts and seconds, which each have the effect of applying a test to every member of a list. This program structure will be quite common, and a method for representing the pattern as a separate predicate is desirable. The predicate would require two arguments, the list whose members are to be tested, and the predicate which implements the tests. Hence the pattern would be represented by the clauses of a higher order predicate, and the predicate valued argument would be represented in particular instances by a new program construct, a "predicate expression", containing one or more program clauses.

Investigation is currently proceeding of such a dialect of "higher order Prolog". The modified language is not simply transformed into basic Prolog, however basic, extended basic, and structured Prolog programs can be transformed into special cases of higher order programs. It cannot be expected that the full power of higher order logic programs will be captured in an implementation of higher order Prolog, and indeed such power would not be exploited in practical programming. The problem then is to develop a semantic description of higher order Prolog which provides adequate programming power, but which is sufficiently constrained to be implementable.

I am interested in the development and implementation of various forms of Prolog, in particular higher order, in the context of a broad investigation of the performance characteristics of very high level language systems. The study is designed to cover methodologies for the assessment and comparison of systems, to obtain quantitative measurements of the influence of implementation decisions, and in the long term to attempt a quantitative ranking of systems representing a range of programming styles. Structured and higher order Prolog will play their part in this investigation, since they will be representing logic programming in a form which is at least equal to functional programming in expressive power.

References

[1] R. Kowalski. Predicate Logic as a Programming Language. Proc. IFIP Congress, 1974.

[2] K. Clark and F. McCabe. IC-Prolog Reference Manual. CCD Report. Imperial College.

[3] D. Warren. Implementing Prolog - Compiling Predicate Logic Programs. DAI Res. Reports 39, 40. Edinburgh University, 1977.

[4] P. Henderson. Functional Programming - Application and Implementation. Prentice Hall Int. Series in Computer Science, 1980.

[5] P.J. Landin. The Next 700 Programming Languages. Comm. ACM, Vol. 9, No. 3, March 1966.

and unification, giving θ_1, binds

$L1_1$ to cons(cons(1,2),nil)

and

$L2_1$ to cons(cons(3,X_0),nil)

Hence gc_1:

← $firsts_1$(cons(cons(1,2),nil)),seconds(cons(cons(3,X_0),nil))

and pc_1 contains

$pairs_0$(L1,L2)←....

$firsts_1$(nil)←

$firsts_1$(cons(cons(X_1,Y),L1))←$firsts_1$(L1)

$seconds_1$(nil)←

$seconds_1$(cons(cons(Y,X_1),L2))←$seconds_1$(L2)

To construct θ_2,gc_2 and pc_2 I will select the first condition from gc_1 and the third clause from pc_1. The variant is simply

$firsts_1$(cons(cons(X_1,Y_2),$L1_2$))←$firsts_1$($L1_2$)

which gives θ_2 binding

X_1 to 1, Y_2 to 2, and $L1_2$ to nil.

The new goal clause gc_2 is

← $firsts_1$(nil),$seconds_1$(cons(cons(3,X_0,nil))

and pc_2 is found by substituting θ_2 through pc_1:

$pairs_0$(L1,L2)←....

$firsts_1$(nil)←

$firsts_1$(cons(cons(1,Y),L1))← $firsts_1$(L1)

$seconds_1$(nil)←

$seconds_1$(cons(cons(Y,1),L2))← $seconds_1$(L2)

Note that the value 1 of X_1, common to both the $firsts_1$ and $seconds_1$ clauses, has been substituted into these clauses.

The remaining substitutions, goal clauses and program clause sets are constructed in a straightforward manner, with the result that the original unknown, X_0, is bound to the constant 1.

Hence the result, gc_{res}:

← $pairs_0$(cons(cons(1,2),nil),cons(cons(3,1),nil))

Higher Order Extensions and Further Work

An important technique in the design of programs concerns the discovery of common patterns of program construction, and the abstraction of each pattern into a clear description independent of its particular applications. Many languages provide some way of achieving this, for example procedure parameters in Algol type languages. In functional languages the facility takes the form of λ expressions, whose values are functions which can be applied to arguments, and higher order functions, which may take function values as arguments, and may return function values as results.

That this technique may be of some use in Prolog can be seen from the pairs example in the previous section. There are two locally defined predicates, firsts and seconds, which each have the effect of applying a test to every member of a list. This program structure will be quite common, and a method for representing the pattern as a separate predicate is desirable. The predicate would require two arguments, the list whose members are to be tested, and the predicate which implements the tests. Hence the pattern would be represented by the clauses of a higher order predicate, and the predicate valued argument would be represented in particular instances by a new program construct, a "predicate expression", containing one or more program clauses.

Investigation is currently proceeding of such a dialect of "higher order Prolog". The modified language is not simply transformed into basic Prolog, however basic, extended basic, and structured Prolog programs can be transformed into special cases of higher order programs. It cannot be expected that the full power of higher order logic programs will be captured in an implementation of higher order Prolog, and indeed such power would not be exploited in practical programming. The problem then is to develop a semantic description of higher order Prolog which provides adequate programming power, but which is sufficiently constrained to be implementable.

I am interested in the development and implementation of various forms of Prolog, in particular higher order, in the context of a broad investigation of the performance characteristics of very high level language systems. The study is designed to cover methodologies for the assessment and comparison of systems, to obtain quantitative measurements of the influence of implementation decisions, and in the long term to attempt a quantitative ranking of systems representing a range of programming styles. Structured and higher order Prolog will play their part in this investigation, since they will be representing logic programming in a form which is at least equal to functional programming in expressive power.

References

[1] R. Kowalski. Predicate Logic as a Programming Language. Proc. IFIP Congress, 1974.

[2] K. Clark and F. McCabe. IC-Prolog Reference Manual. CCD Report. Imperial College.

[3] D. Warren. Implementing Prolog - Compiling Predicate Logic Programs. DAI Res. Reports 39, 40. Edinburgh University, 1977.

[4] P. Henderson. Functional Programming - Application and Implementation. Prentice Hall Int. Series in Computer Science, 1980.

[5] P.J. Landin. The Next 700 Programming Languages. Comm. ACM, Vol. 9, No. 3, March 1966.

A FUNCTIONAL PLUS PREDICATE LOGIC PROGRAMMING LANGUAGE

Marco Bellia(+), Pierpaolo Degano(*), Giorgio Levi(+,*)

1. Introduction.

In the last few years, languages based on first order logic became very popular, due to several reasons that make them good candidates not only as specification languages, but also as practical programming languages.

The main features of such languages are:

i) They have a clear mathematical basis, which allows to define a straightforward formal semantics and which provides a natural environment for proving properties of programs;

ii) They are nice examples of applicative languages, whose semantics is not based on state transitions, and they lead to a hierarchically structured non von Neumann programming style /1/;

iii) Last but not least, today's technology allows to design efficient implementations /2-6/.

Predicate logic programming languages can be classified according to the kind of procedures they define. In the first class (relational languages) procedures are defined as relations. The first example of a relational language is PLANNER /7/. Kowalski's language /8/ is a milestone within this family, because of the formal definition of procedures as sets of Horn clauses, and its clean mathematical semantics /9/. On Kowalski's footsteps, PROLOG /2-6, 10-11/ and other similar languages /12-14/ have been proposed. In the second class of languages (functional languages) procedures are defined by sets of functional equations. Languages within such a class have been motivated by several different problems, namely proving program properties in formal systems /15-19/, and abstract data type specification /20-23/.

There are no definite arguments in favour of one class against the other, yet each class has its own appealing features. Namely, a uniform evaluation rule can more easily be defined for functional languages, while relational languages lead to non-deterministic interpreters. Properties of programs (i.e. lemmas and theorems to be used in symbolic simplifications) are more expressively defined within the functional approach. On the other hand, relational languages are exactly what is needed to describe procedures with more than one output.

The language described in this paper is based on an attempt to combine relational and functional languages in a unified environment, which provides the best features of both approaches.

Our goal was to design a first order logic language, which allows to define both functions and procedures. Our language is a proper extension of functional languages enriched with somewhat constrained Horn clauses. The constraints are concerned with distinguishing between input and output parameters and sequencing of literals. In the resulting language, predicates play the role of standard programming language procedures. Moreover, it is possible to define an efficient deterministic interpreter.

(+) Istituto di Elaborazione dell'Informazione - C.N.R., Via S.Maria, 46 - I56100 Pisa (ITALY)

(*) Istituto di Scienze dell'Informazione - Università di Pisa - I56100 Pisa (ITALY)

2. The Syntax of FPL.

The Functional plus Predicate Logic (FPL) programming language is a strongly typed first order language, whose programs are equations defined according to first order logic over the alphabet $A=\{S,C,D,V,F,R\}$, where:

S is a set of identifiers. Given S, we define a sort s which is:
i) simple if $s \in S$, ii) functional if $s \in S^* \rightarrow S$, iii) relational if $s \in S^* \rightarrow S^*$.
C is a family of sets of constant symbols indexed by simple sorts.
D is a family of sets of data constructor symbols indexed by functional sorts.
V is a family of denumerable sets of variable symbols indexed by simple sorts.
F is a family of sets of function symbols indexed by functional sorts.
R is a family of sets of predicate symbols indexed by relational sorts.

Families are defined in the language by declarations, which assign a specific simple or functional or relational sort to each object.

Examples are:

0:→NAT; succ: NAT→NAT
nil:→NLIST; cons: NAT x NLIST→NLIST
+: NAT x NAT→NAT
eqn: NAT x NAT→BOOL
ndiv: NAT x NAT→NAT x NAT.

A FPL program is a set of declarations and equations. Each symbol occurring in an equation must be declared.

The syntax of equations is based on the standard concepts of term and atomic formula.

A term is either a data term or a functional term.

A data term of sort s ($s \in S$) is
i) a constant symbol of sort s,
ii) a variable symbol of sort s,
iii) a data constructor application $d(t_1,\ldots,t_n)$ such that $t_1,\ldots,t_n$ are data terms of sort $s_1,\ldots,s_n$ and $d \in D$ has sort $s_1 \times \ldots \times s_n \rightarrow s$.

A functional term of sort s ($s \in S$) is a function application $f(t_1,\ldots,t_n)$, such that $t_1,\ldots,t_n$ are data terms of sorts $s_1,\ldots,s_n$ and $f \in F$ has sort $s_1 \times \ldots \times s_n \rightarrow s$.

An atomic formula is either
i) a functional atomic formula of the form t=d, where d is a data term of sort s and t is a term of the same sort, or
ii) a relational atomic formula of the form $r(\underline{in}:t_1,\ldots,t_m;\ \underline{out}:t_{m+1},\ldots,t_n)$, such that $t_1,\ldots,t_m,t_{m+1},\ldots,t_n$ are data terms of sorts $s_1,\ldots,s_m,s_{m+1},\ldots,s_n$ and $r \in R$ has sort $s_1 \times \ldots \times s_m \rightarrow s_{m+1} \times \ldots \times s_n$.

A constraint is either
i) an atomic formula, or
ii) a formula of the form c_1,c_2 such that c_1 is an atomic formula and c_2 is a constraint.

Constraints are used to combine functional terms (function calls) and atomic formulas (procedure calls) in a program. Constraints define a local environment which is shared by (and allows the interaction among) its components. Constraints can be used within function and procedure definitions, according to the following syntax of equations.

Equations are formulas of the following form $l \leftarrow r$, where l is the left part and r is the right part, such that its left part l is an atomic formula possibly followed by a constraint and its right part r is either empty or a constraint.

The equation is functional or relational, according to the type of its atomic formula.

Example:

1. true:→BOOL
2. false:→BOOL
3. 0: → NAT
4. s: NAT→NAT
5. minus: NATxNAT→NAT
6. lt: NATxNAT→BOOL
7. ndiv: NATxNAT→NATxNAT

e8. minus(x,0)=x ←
e9. minus(s(x),s(y))=z←minus(x,y)=z
e10. lt(0,s(x))=true←
e11. lt(x,0)=false←
e12. lt(s(x),s(y))=z←lt(x,y)=z
e13. ndiv(in:x,y:out:0,x),lt(x,y)=true←
e14. ndiv(in:x,y;out:s(q),r),lt(x,y)=false←ndiv(in:z,y;out:q,r),minus(x,y)=z
e15. isfact(x,y)=false,ndiv(in:y,x;out:z,s(r))←
e16. isfact(x,y)=true,ndiv(in:y,x;out:z,0)←

Declarations 1-3, 4, 5-6, 7 are constant, data constructor, function and relation declarations, respectively. The example is completed with the functional equations e8-e12, e15-e16 and the relational equations e13-e14.

The above definition of equation is inadequate, since context-dependent conditions on variable occurrences are needed to guarantee proper nesting of constraints and binding of local variables. Some more definitions are needed to introduce the conditions. In order to give some insight into the meaning of the conditions, we will informally use operational arguments.

A definition contains atomic formulas of the form $r(in:x_1,\ldots,x_n;out:y_1,\ldots,y_m)$, or $f(x_1,\ldots,x_n)=y$. Let us define, for each atomic formula a the multisets of input and output variable occurrences. Namely,

$M_{in}(a)$ is the multiset of the variable occurrences in terms $x_1,\ldots,x_n$, while $M_{out}(a)$ is the multiset of the variable occurrences in terms $y_1,\ldots,y_m$ or y.

Each definition has a header, consisting of the leftmost atomic formula, and a set of invocations, whose element are the other atomic formulas. Let H and $I=\{I_i\}$ be the header and the set of invocations of an equation e.

Condition 1. The multisets $M_{in}(H)$ and $M_{out}(I)=\bigcup_i M_{out}(I_i)$ must be sets.

The absence of multiple occurrences of a variable in the header corresponds to the left-linearity, while the absence of multiple output occurrences of a variable in the set of invocations rules aliasing out.

Examples of equations not satisfying condition 1 are:

eq(x,x)=true ← (since it would impose a specific relation on input values),

r(in:x;out:y,z) ← g(w)=y,f(x)=w,q(in:x;out:w,z) (since variable w is (output) constrained (i.e. could be computed) by two different constraints).

Condition 2. $M_{in}(H) \cap M_{out}(I)=\phi$.

Disjointness of sets of header input variables and invocations output variables in an equation is connected with the non invertibility of programs. As an example, the equation p(in:x,y;out:z) ← r(in:y,z;out:x),f(y)=z,is ruled out, because it imposes a constraint on the variable x (i.e. it may invert with respect to x).

Condition 3.

3.1. All variable symbols occurring in $M_{out}(H)$ and $M_{in}(I_i)$, must belong either to $M_{in}(H)$ or to $M_{out}(I_k)$, where I_k is an inner invocation (the innermost invocations possibly being in the left part constraint).

3.2. For each invocation I_k in a right part constraint, $M_{out}(I_k)$ must contain at least one variable symbol belonging either to $M_{out}(H)$ or to $M_{in}(I_i)$, where I_k in an inner invocation.

Example of equations which do not satisfy condition 3 are:

r(in:x,y;out:z,w),f(x,y)=t←h(t)=w (since the output z cannot be computed),

p(in:x,y;out:z)←g(t,w)=z,f(x,y)=w (since intermediate variable t cannot be computed),

f(x,y)=z,k(x,y,t)=z ← h(x)=t (since the left part constraint could not be computed before the right part constraint),

h(x)=t←g(x,z)=t,f(x,t)=z (since there exists a circular precedence relation between invocations),

r(in:s(x),y;out:s(z)) ← r(in:x,y;out:z),f(x,y)=w (since the invocation f(x,y)=w never needs to be computed).

r(in:x,y;out:z) ← h(x)=z,f(x,y)=false (since false is a constant symbol occurring as output of an invocation which will never be computed).

Thus far we have defined well-formed equations. A set of equations should denote sets of procedures. Since our aim is to restrict sets of equations so as to define (deterministic) procedures by disjunct cases, we are forced to introduce more definitions and conditions.

Conditions on a set of equations are concerned with the non superposition property on the equations left parts and relies on (first order) unification.

An equation left part consists of a header and a (possibly empty) set of invocations. Let c be any header or invocation,

i) n(c) be the function or relation symbol in c,

ii) $D_{in}(c)$ the n-tuple of input data terms in c,

iii) $D_{out}(c)$ the n-tuple of output data terms in c.

Given a set of equations $E=\{e_i\}$, the set has the non superposition property if for any pair of equations $l_i \leftarrow r_i$, $l_j \leftarrow r_j$, the left parts l_i and l_j are non overlapping.

Condition 4. Two left parts l_i and l_j are non overlapping if one of the following properties holds:

1) $n(h_i) \neq n(h_j)$, where h_i and h_j are the header of l_i and l_j.

2) $D_{in}(h_i)$ and $D_{in}(h_j)$ are non-unifiable.

3) $D_{in}(h_i)$ and $D_{in}(h_j)$ are unifiable with most general unifier λ, l_i and l_j constraints k_i and k_j, and $[k_i]_\lambda$, $[k_j]_\lambda$ are syntactically disjoint.

Condition 5. Two constraints k_i and k_j are syntactically disjoint if one of the following properties holds.

1) k_i and k_j are invocations, $n(k_i)=n(k_j)$, $D_{in}(k_i)=D_{in}(k_j)$ and $D_{out}(k_i)$, $D_{out}(k_j)$ are non-unifiable.

2) k_i and k_j have the form c_{i1}, k_{i2} and c_{j1}, k_{j2} respectively, and either

2.1 c_{i1} and c_{j1} are syntactically disjoint, or

2.2 $n(c_{i1})=n(c_{j1})$, $D_{in}(c_{i1})=D_{in}(c_{j1})$, $D_{out}(c_{i1})$ and $D_{out}(c_{j1})$ are unifiable with most general unifier λ, and $[k_{i2}]_\lambda$, $[k_{j2}]_\lambda$ are syntactically disjoint. The following are sets of overlapping equations:

{+(x,0)=x←,
+(0,x)=x←},

{plus(in:x,y;out:y),eq(x,0)=true←,
plus(in:x,y;out:z)←plus(in:y,x;out:z),
plus(in:x,y;out:z),eq(y,0)=true←,
plus(in:x,y;out:0),+(x,y)=0←,
plus(in:x,y;out:z),+(x,y)=z←}

Let us finally introduce the syntactic construct program. A program has the same form of an equation right part, namely it is a constraint. Hence a program consists of a set of invocations $I=\{I_i\}$, whose variables must obey the following conditions.

Condition 6. $M_{out}(I)=\bigcup_i M_{out}(I_i)$ must be a set.

Condition 7.

7.1. For each I_i in a program, each variable belonging to $M_{in}(I_i)$ must belong to $M_{out}(I_k)$, where I_k is a inner invocation.

7.2. For each I_k in a program, $M_{out}(I_k)$ must contain at least one variable symbol which belongs to $M_{in}(I_i)$, where I_k is an inner invocation.

Conditions 6 and 7 ensure that a program is closed.

In section 3 we will introduce FPL operational semantics, which allows to define a computation from given program and set of equations. It is worth noting that our lengthy and tedious definition of the FPL syntax (typically, the conditions for well-formedness of equations, sets of equations and programs), was mainly concerned with semantic properties, which can be incorporated into the syntax and statically checked. The possibility of defining a deterministic FPL interpreter relies exactly on such conditions.

Let us finally note that the syntax we have defined does not allow function composition. However, our syntax has to be seen as the abstract FPL syntax. The concrete syntax will allow to use standard function composition. Namely, a general term obtained by function composition can replace a functional term every where in an equation, but in an equation header.

The functional and relational aspects of FPL can be distinguished leading to two different subsets of the language.

The language obtained ruling out relational atomic formulas and left part constraints, is a subset of the functional language TEL /15/, since it does not allow to express properties.

Ruling out functional atomic formulas and left part constraints, we obtain a specific class of Horn clauses, characterized by input-output separation and ordering of the right part atomic formulas. The above constraint forbids program invertibility, yet leads to a deterministic interpreter.

FPL can be extended by releasing some of the above conditions in order to allow to express properties of programs as well. Such an extension, however, is outside the scope of this paper.

3. Operational Semantics.

The operational semantics will be defined by describing the FPL interpreter. The interpreter consists of a set of mutually recursive procedure (EVAL,MATCH,UNIFY) which operate on abstract representations of programs and constraints (closure structures), that will be defined in the following.

A set of invocations $I=\{I_i\}$ can be represented as a closure set, which contains a closure for each invocation I_i. The closure corresponding to invocation I_i is the pair $c=<I_i,env(I_i)>$, where $env(I_i)$ is a set of bindings for all the input variables of I_i (which are also input variables of closure c).

A binding possibly associates an input variable v to the closures which correspond to those invocations in I which have v among their output variables.

A closure structure is a set of closures $C=\{c_i\}$, such that:

i) For each closure c_i in the set and for each input variable v in c_i, v is bound to exactly one closure in C.

ii) The multiset of output variables of all the closures of C is a set.

Let Γ be a closure structure. If we associate a labeled node to each closure in Γ and a directed arc from node labeled c_i to node labeled c_j, if some input variable of c_j is bound to c_i. Then, a closure structure is a directed graph.

Let Γ be a closure structure and c_i be a closure in Γ. The substructure of Γ rooted at c_i is the closure structure Γ/c_i defined as follows;

i) $c_i \in \Gamma/c_i$

ii) If closure c_k belongs to Γ/c_i, then Γ/c_i contains all the closures of Γ whose output variables are input variables of c_k.

A substitution is a closure structure λ, such that for each closure $c \in \lambda$ and for each output variable v in c, there exists no closure belonging to the substructure λ/c which has v among its input variables.

A root is any closure c_i of λ, such that there exists no closure in λ having an input variable bound to c_i. Hence a substitution is a directed acyclic graph. Note that each substructure of a substitutions is itself a substitution.

The composition $\lambda.\mu$ of a substitution λ with a substitution μ is the closure structure containing the following closures.

i) All the closures of μ.

ii) Only those closures of λ whose output variables are different from the output variables of closures of μ.

The closure structure $\lambda.\mu$ is itself a substitution, because it is acyclic. In fact, the presence of a cycle would require the existence of a closure c_i, such that $c_i \in \mu$ and $c_i \in \lambda.\mu$, which has as input variable a variable v which is bound to some closure c_j, such that $c_j \in \lambda$ and $c_j \in \lambda.\mu$. Even if such a c_j belonging to λ may exist, c_j cannot belong to $\lambda.\mu$ by definition of composition, since variable v must also be an output variable of μ.

A set of closures $C=\{c_i\}$ can be appended to a substitution λ, only if:

i) For each closure c_i and for each input variable v of c_i, v is an output variable of some closure in λ.

ii) The multiset of output variables of C is a set.

iii) The sets of output variables of C and λ are disjoint.

The result $C \| \lambda$ of appending a legal set of closures C to a substitution λ is a substitution.

A FPL program, as defined in Section 2, is a single-rooted substitution (i.e. a directed single-rooted acyclic graph). A program is a closure structure, because

i) Each input variable in an invocation is bound to at least one invocation (condition 7.1) and such an invocation happens to be unique (condition 6).

ii) The multiset of output variables of its invocations is a set (condition 6).

Moreover, a program is a substitution, i.e. it is acyclic, because each invocation input variable is bound to an inner invocation (condition 7.1). Finally, it is single-rooted because condition 7.2 ensures that there exists only one invocation which does not occur in any binding.

The interpreter procedure EVAL will operate on a program, giving a new program as output. In order to allow single-rootness to be preserved by EVAL, the substitution corresponding to a program will be "topped" with a virtual closure (which models the external environment) which contains an empty invocation and has as input variables all the output variables of the program.

It is worth noting that each substructure of a program is a program.

A set of closures $C=\{c_i\}$ is a schematic closure structure if

i) For each closure c_i and for each variable v in c_i, either v is bound to a unique closure of C, or v is free.

ii) The multiset of all the output variables of closures in C is a set.

Hence, a schematic closure structure is different from a closure structure only because some input variables can be free. Schematic substructures and schematic substitutions can easily be defined following the definitions given for the closure structure case. In particular, a schematic substitution G is an acyclic schematic closure structure.

Let free(G) the set of free input variables in G. A schematic substitution G can be instantiated by a substitution λ, if

i) For each variable v in free(G), there exists a closure in λ having v among its outputs.

ii) The sets of output variables of G and λ are disjoint.

The instantiation $[G]_\lambda$ contains all the closures of G and only those closures of λ which belong to a λ/c, such that c has some variable in free(G) among its outputs. $[G]_\lambda$ is a substitution, because all its inputs are bound, all its outputs are different, and there are no cycles since each input of a closure of λ cannot be an output of a closure of G.

A FPL equation e is a triple $<H(e),G_l(e),G_r(e)>$, where:

i) H(e) is the header.

ii) $G_l(e)$ is the left part constraint.

iii) $G_r(e)$ is the right part constraint.

It is possible to prove that both $G_l(e)$ and $G_r(e)$ are schematic substitutions. In fact, for each closure c corresponding to an invocation of either $G_l(e)$ or $G_r(e)$, and for each variable v in c, v is either free, or bound to at least one closure (condition 3.1), which is unique (condition 1). Moreover, the multiset of output variables is a set (condition 1), and there are no cycles, since v can only be bound to an inner constraint (condition 3.1).

We are now able to describe the interpreter procedures.

UNIFY (X:n-tuple of terms, D:n-tuple of terms, λ:substitution);

returns <failure/success, μ:substitution>

X is a n-tuple of data terms $(x_1,\ldots,x_n)$, which contain free variables not occurring in any closure of λ, with no multiple occurrences of the same variable.

D is a n-tuple of data terms $(d_1,\ldots,d_n)$, whose only variables are bound to some closure of λ.

UNIFY is basically first order unification, which returns failure or, in case of success, a set of associations of the form t=v, where v is a variable and t is a data term. In our framework, each association is a closure, having the association as the invocation, variable v as output, and all the variables occurring in t as inputs. As soon as a new association is generated, the corresponding closure is inserted in the (initially empty) set of closures MGU.

Unification proceeds like standard first order unification comparing terms of X to terms of D (possibly) associating variables occurring in X to terms occurring in D. The difference has to do with bound variables occurring in D, which cannot be instantiated, just because they are bound. If unification reaches the point where a bound variable b_i is matched against a non-variable data term t_k (which occurs in X), the following actions are taken.

Step 1. If b_i is bound to closure c whose invocation has the form $t_j=b_i$ and t_j is a data term, then unification proceeds with b_i replaced by t_j.

Step 2. Otherwise, standard unification is suspended and a call is made to EVAL, passing the closure c (to which b_i is bound) and the substitution λ, as parameters. If EVAL returns failure, UNIFY returns failure. Otherwise, EVAL returns a new substitution λ', such that the closure of λ' which has b_i among its outputs is different from c. Step 1 is taken one more time, possibly leading to a further evaluation.

Eventually, unless some EVAL process does not terminate, unification will end up with failure or with a set of closures MGU and a substitution λ^*.

REMARK. The output variables of MGU are exactly the variables occurring in the n-tuple X, while its input variables are all bound to some closure of λ^*. From the conditions imposed on variables occurring in X (which also prevent circularity in most general unifiers), it follows that the set of closures MGU can be appended to the substitution λ^*.

UNIFY returns the substitution μ=MGU $\parallel \lambda^*$.

MATCH (e:equation, a:atomic formula, λ:substitution);

returns <failure/success, μ:substitution>

a is an atomic formula, whose only variables are bound to closures of λ.

e is an equation, with header H(e) and (possibly empty) left part constraint $G_1(e)$.

Step 1. If the function (or predicate) symbols occurring in a and H(e) are different, returns failure.

Step 2. Otherwise, let X be the n-tuple of input data terms in H(e) and let D be the n-tuple of input data terms in a.

REMARK. When MATCH is called, e is a renaming of a FPL equation. Hence all the variables in X do not occur in any closure of λ. Moreover, because of condition 1, no variable can have multiple occurrences in X. Finally, all the input variables of D are bound to some closure of λ. Therefore, UNIFY can be applied to parameters X, D and λ.

Call UNIFY(X,D,λ). If UNIFY returns failure, return failure. Otherwise, let λ' be the substitution returned by UNIFY. If G_1 is empty, return $\mu = \lambda'$.

Step 3. Otherwise, let $\lambda'' = [G_1(e)]_{\lambda'}$ be the instantiation of the schematic substitution $G_1(e)$ by the substitution λ'.

REMARK. Each variable in free($G_1(e)$) is bound to some closure in λ', because a free input variable in the left part constraint can only be an input variable of H(e) (condition 3.1), and because all the input variables of H(e) are output variables of λ' (by definition of UNIFY). Hence λ' can be used to instantiate $G_1(e)$.

Let $C=\{c_i\}$, $1 \leq i \leq k$, be the k-tuple of closures, such that each c_i is a root of λ''. Set i=1 and $\lambda_1 = \lambda''$.

Step 4. Call EVAL(λ_i,c_i). If EVAL returns failure, return failure. Otherwise, if i=k return the substitution $\mu = \lambda' . \lambda_{i+1}$, which is the composition of the output substitution of UNIFY and the output substitution λ_{i+1} of the last EVAL.

Step 5. If i≠k, increase i by 1, and iterate Step 4.

REMARK. If eventually, MATCH returns success, its output substitution μ has among its output variables all the input variables of H(e) and all the output variables of $G_1(e)$ (if any).

EVAL (λ:substitution, c:closure);

returns <failure/success, μ:substitution>, c is any closure of λ.

Step 1. Let I be the invocation associated with the closure c. According to the form of I, one of the following actions is taken.

1.1 If I is empty (top closure of a program), let $C=\{c_i\}$, $1 \leq i \leq k$, the k-tuple

of closures to which the input variables of I are bound. Set i=1 and $\lambda_i = \lambda/c_i$.

1.1.1 Call EVAL(λ_i,c_i). If EVAL returns failure return failure. Otherwise let λ'_i be the substitution returned by EVAL. If i=k, let $\lambda' = \lambda'_i$ and go to step 2, otherwise increase i by 1, set $\lambda_{i+1} = \lambda/c_{i+1} \cdot \lambda'_i$ and iterate step 1.1.1.

1.2 If I has the form d=v, where d is a data term and v is a variable, then

1.2.1 If d is not a variable, then return λ.

1.2.2 If d is an (input) variable, let c' be the (unique) closure in λ to which d is bound. Call EVAL(λ/c',c'). If EVAL returns failure, return failure. Otherwise, let λ' be the output substitution of EVAL and go to step 2.

1.3 If I is an atomic formula, for each equation e_i in the global set of equations E, a nondeterministic call to MATCH is performed, MATCH(e_i,I, λ^+), where e_i is a new consistent renaming of equation e_i, and λ^+ is the substructure of λ rooted at c, c non included.

1.3.1 If no MATCH succeeds, return failure. Otherwise, let e'_k and λ_k be one successful equation and the output substitution of the corresponding MATCH. If $G_r(e'_i)$ is empty, set $v' = \lambda_k$ and go to 1.3.3.

REMARK. Because of the non superposition condition (conditions 4 and 5) on sets of equations, a unique MATCH can terminate successfully. However, we are not allowed to handle the different equations sequentially since MATCH could be nonterminating.

1.3.2 Let v be the instantiation $[G_r(e'_k)]_{\lambda_k}$, of the schematic substitution, associated to the right part constraint of the successful equation by the output substitution of the successful MATCH, and $v' = \lambda_k . v$.

REMARK. λ_k can be used to instantiate $G_r(e'_k)$, because each variable in free($G_r(e'_k)$) is either an input variable of $H(e'_k)$ or an output variable of $G_1(e'_k)$ (because of condition 3.1), and λ_k has all such variables as output variables (see the last remark to MATCH). Moreover, for each output variable v of $G_r(e'_k)$, v cannot be an output variable of λ_k. In fact, because of equation renaming, for v to be an output variable of λ_k, v must be either an input variable of $H(e'_k)$ (contradictory because of condition 2) or an output variable of $G_1(e'_k)$ (contradictory because of condition 1).

1.3.3 Let X be the n-tuple of output data terms of closure c, and D be the n-tuple of the output data terms of $H(e'_k)$.

REMARK. We want to show that X, D and v' are legal parameters for UNIFY. We must prove that

i) There are no multiple occurrences of a variable in X (by condition 6, i.e. absence of aliasing in a procedure call).

ii) All the variables in D are bound to some closure in v'. Each variable in D is an output variable of $H(e'_k)$. By condition 3.1 it must also be either an input variable of $H(e'_k)$ or an output variable of $G_1(e'_k)$ or $G_r(e'_k)$. On the other hand, all the output variables of $G_r(e'_k)$ are outputs of v, while all the input variables of $H(e'_k)$ and the output variables of $G_1(e'_k)$ are output variables of λ_k. Hence, they are all output variables of $v' = \lambda_k . v$.

iii) Each variable v in X is not an output variable of any closure in v'. Initially, c is the only closure in λ having v as output variable. It is rather easy to prove that the only new output variables directly generated by MATCH and UNIFY are variables coming from renamed equations (and therefore different from v). We only need to show that each recursive call to EVAL (via MATCH and UNIFY) has the following property.

EVAL property. Let μ be the output of EVAL(λ,c); for each closure c' such that:
a) $c' \in \mu$ b) $c' \notin \lambda$ c) c' has an output variable which is also an output variable of some closure c" in λ,
the closure c" belongs to λ/c.

We will assume here the property to hold.

1.3.4 Call UNIFY(X,D, ν'). If UNIFY fails, returns failure. Otherwise, let λ^* be the output substitution of UNIFY.

REMARK. λ^* has all the output variables of c as outputs.

Let λ' be the structure which contains only those closure of λ^* which belong to substructures of λ^* rooted at closures which have as output variable an output variable of c.

REMARK. λ' is a substitution.

Step 2. Return $\mu = \lambda . \lambda'$.

REMARK. The EVAL property follows directly from the above construction.

A FPL program Π is evaluated by calling EVAL with substitution Π and with the unique root of Π as closure.

EVAL is clearly based on an external rule. Since our language has no builtin data types, and since "constructors are not evaluated", the FPL rule is a call-by-need, whose behaviour can be summarized as follows. "An atomic formula is evaluated so much as it is needed to allow unification".

The above defined abstract interpreter suggests an efficient implementation, which does not require equation renaming, and which modifies programs and substitutions by side effects, through the use of structure sharing. The same technique was successfully used in several predicate logic language implementations and theorem provers. In fact, with structure sharing, different instances of the same atomic formula are identified, thus avoiding multiple evaluations of atomic formulas which typically arise in call-by-name interpreters.

Even if the language is deterministic, the above described interpreter is nondeterministic. The EVAL Step, in which a program is nondeterministically MATCHED against all the equations left parts, could be implemented by backtracking, provided that the following property holds.

Backtracking property. Let I be an invocation whose input variables are bound in a substitution λ, and let $E=\{e_1,\ldots,e_n\}$ be a set of equations, such that for each equation e_i in E, $H(e_i)$ contains the function or predicate symbols occurring in I. If MATCH(e_k,I, λ), for some $e_k \in E$, diverges, then MATCH(e_i,I, λ) diverges for all e_j belonging to E.

The above property holds if one more simple condition is imposed on sets equations. For the sake of brevity, the condition will not be described here. Let us only remark that if we take equations satisfying such a condition (which, roughly speaking, are simply good recursive definitions), the call-by-need and structure sharing implementation is in a sense "optimal", because all the evaluations which could have been performed within a failing MATCH are transmitted to the next MATCH, and would have been, in any case, performed by the successful MATCH, if any.

We will now give an example showing our use of the left part constraint, which allows recursive by cases definitions (without built-in conditional), with cases being defined by general atomic formulas. The example shows the evaluation of the program isfact(s(s(0)),s(0)) with equations e8,...,e16 in Section 2 (i(c) denotes the invocation of closure c).

```
c0: isfact(s(s(0)),s(0))=x
λ0= c0
EVAL(λ0,c0)
  λ1 ={}
  MATCH(e^2_15,i(c0),λ1)
    UNIFY((x1,y1),(s(s(0)),s(0)),λ1)
    c1: s(s(0))=x1
    c2: s(0)=y1
    λ2={c1,c2}
    c3: ndiv(in:y1,x1;cut:z1,s(c1))
    λ3={c1,c2,c3}
    EVAL(λ3,c3)
      λ4= c1,c2
      MATCH(e^2_13,i(c3)
        UNIFY((x2,y2),(y1,x1),λ4)
        c4: s(0)=x2
        c5: s(s(0))=y2
        λ5={c1,c2,c4,c5}
        c6: lt(x2,y2)=true
        λ6={c4,c5,c6}
        EVAL(λ6,c6)
          λ7={c4,c5}
      MATCH(e^3_10,i(c6),λ7)
        UNIFY((0,s(x3)),(x2,y2),λ7)
        failure
      failure
      MATCH(e^4_11,i(c6),λ7)
        UNIFY((x4,0)(x2,y2),λ7)
        failure
      failure
      MATCH(e^5_12,i(c6),λ7)
        UNIFY((s(x5),s(y5)),(x2,y2),λ7)
        c7: 0=x5
        c8: s(0)=y5
        λ8= c4,c5,c7,c8
      λ8
      c9: lt(x5,y5)=z5
      λ9={c7,c8,c9}
      λ10=λ8.λ9={c4,c5,c7,c8,c9}
      UNIFY((true),(z5),λ10)
        EVAL(λ10,c9)
          λ11={c4,c5,c7,c8}
          MATCH(e^6_10,i(c9),λ11)
            UNIFY((0,s(x6)),(x5,y5),λ11)
            c10: 0=x6
            λ12={c4,c5,c7,c8,c10}
          λ12
          UNIFY(z5),(true),λ12)
          c11: true=z5
          λ13={c4,c5,c7,c8,c10,c11}
        λ13={c4,c5,c7,c8,c11}
      λ14
    λ15={c4,c5,c6}
λ16=λ5.λ15 {c1,c2,c4,c5,c6}
                UNIFY((z1,s(c1)),(0,x2),λ16)
                c12: 0z1
                c13: 0=c1
                λ17={c1,c2,c4,c5,c6,c12,c13}
              λ18={c1,c2,c12,c13}
            λ18
            UNIFY((x),(false,λ18)
            c14: false=x
            λ19={c1,c2,c12,c13,c14}
          λ20={c14}
```

4. Fixed-point Semantics.

In this Section, we will describe the fixed-point semantics of a set of equations $E=\{e_i\}$. For the fixed-point semantics, each equation e can be seen as a pair $<H(e_i), G(e_i)>$, such that $G(e_i)$ is the set of all the invocations occurring both in the left part and in the right part of e_i. It is worth noting that generally two equations

1. P , Q ← R , S 2. P ← R , S , Q

which differ only because one invocation occurs in the left part and in the right part, are different both from the operational and the mathematical viewpoint. The difference is only operational if invocation Q satisfies condition 3.2 (i.e. it has at least one output variable, which is an input to S or to R or an output of P). If this is the case equation 2 is a legal equation. The operational difference is concerned with nondeterminism. With equation 1, as soon as a MATCH succeeds, the other nondeterministic attempts in EVAL can be killed (since we are guaranteed from conditions 4 and 5 that any other MATCH would fail). Failing in the evaluation of Q, within MATCH, would just kill the current attempt. With equation 2, a failure in the evaluation of Q could only be detected after the successful MATCH. This would require to backtrack to a choice-point which had already succeeded (nonrecursive backtracking). This situation corresponds to the fact that equation 2 could possibly have a superposition with other equations. In such a case we are not guaranteed that when a match is successful, no other successful MATCHing is possible.

On the other hand, if 2 does not satisfy condition 3.2, equation 2 is not a legal equation. As a matter of fact, equations 1 and 2 would have a completely different semantics if the evaluation of Q diverges or fails. In fact, in such a case, Q would not be evaluated by equation 2.

The fixed-point semantics gives to equation e_i a semantics which is equivalent to the operational semantics of e_i, only if all the invocations of $G(e_i)$ which do not satisfy condition 3.2 occur in the left part of e_i (i.e. if e_i is a legal equation). The fixed-point semantics of E is a model of E, obtained as the fixed-point of a transformation φ_E on interpretations. Our fixed-point semantics is very close to the semantics defined in /9/. Our semantics however, is a call-by-name semantics. Therefore our domain will contain an undefined object ω_s, for each simple sort s.

Interpretations are defined on an abstract domain A, which is a family of sets A_s, each set beeing indexed by a sort s occurring in E. Each A_s is defined as follows:

i) ω_s belongs to A_s;
ii) All the constant symbols of sort s, occurring in E, are in A_s;
iii) For each data constructor symbol d of sort $s_1 \times \dots \times s_n \rightarrow s$, A_s contains all the terms $d(t_1,\dots,t_n)$, such that $t_1,\dots,t_n$ belong to $A_{s1},\dots,A_{sn}$, respectively. A term belonging to a family A is undefined if it contains ω_s, for some sort s which indexes a set A_s in A.

An interpretation ξ_i is any subset of the interpretation base B. The interpretation base B is a set of atomic formulas defined as follows:

i) For each function symbol f (occurring in E) of sort $s_1 \times \dots \times s_n \rightarrow s$, B contains all the formulas $f(t_1,\dots,t_n)=t$ such that $t_1,\dots,t_n$ and t have sorts $s_1,\dots,s_n,s$ respectively, and term t is not undefined.
ii) For each predicate symbol P(occurring in E) of sort $s_1 \times \dots \times s_m \rightarrow s_{m+1} \times \dots \times s_n$, B contains all the formulas $P(\underline{in}:t_1,\dots,t_m;\underline{out}:t_{m+1},\dots,t_n)$, such that $t_1,\dots,t_m, t_{m+1},\dots,t_n$ have sorts $s_1,s_m,s_{m+1},\dots s_n$ respectively, and terms $t_{m+1},\dots,t_n$ are not undefined.

Roughly speaking, an interpretation assigns output values to applications of functions and relations to ground input values. All the other applications have some undefined output. An interpretation ξ_i is "more defined" than interpretation ξ_j if $\xi_j \subseteq \xi_i$, where $\subseteq$ is set inclusion. Note that the partial ordering relation $\subseteq$ on interpretations corresponds to an intuitive notion of better approximation. In fact, if $\xi_j \subseteq \xi_i$, ξ_i assigns output values to some applications that in ξ_j had some undefined output.

Transformation φ_E maps interpretations on interpretations and is defined as follows.

Let ξ_i be any interpretation and $e_k = \langle H(e_k), G(e_k) \rangle$ be an equation of E. Equation e_k defines a transformation φ^k which maps ξ_i onto the interpretation $\xi_i^\kappa = \varphi^k(\xi_i)$, such that

1)) All the atomic formulas of ξ_i are in ξ_i^κ.

2) For each instantiation λ of variables to terms such that, for each invocation I_j in $G(e_k)$ either

2.1) $[I_j]_\lambda$ is in ξ_i, or

2.2) An output variable v of I_j, which is not an output variable of $H(e_k)$, is instantiated to an undefined term by λ,

the formula $[H(e_k)]_\lambda$ is in ξ_i^κ.

Note that λ must instantiate a variable v of sort s to a term belonging to A_s, and that if $G(e_k)$ is empty, condition 2.2 is satisfied for any instantation λ.

The transformation φ_E is the transformation defined by all the equations of E according to the above definition, i.e. $\varphi_E(\xi_i) = \bigcup_{e_k \in E} \varphi^k(\xi_i)$.

It can be proved that transformation φ_E on the set of interpretations partially ordered by set inclusion is monotonic and continuous. Hence, there exists the least fixed-point interpretation ξ^* such that $\xi^* = \varphi_E(\xi^*)$, which can be obtained by iteratively applying φ_E, starting with the empty subset of B, which is the bottom element of the partially ordered set of interpretations.

5. Conclusion

We have described a new first order logic language, which combines the functional and the relational approach. We have defined the fixed-point semantics and we have shown an interesting operational model which is both formal and close to efficient implementations. Both the fixed-point and the operational semantics are based on theorems that have been either assumed or informally proved in this paper. A complete formalization was certainly outside of the scope of the present paper.

We have some nice examples of FPL programs, that would be innatural and awkward, in a predicate language without left part constraints or in a functional language. The improved expressive power of the language is due to the presence of both the function and the procedure constructs and to the left part constraints which provide the full power of a built-in conditional, while saving the first order logic axiomatic flavour. One more interesting feature of FPL is its ability to describe non-strict functions and relations. Non strict functions, as the if-then-else, can easily and naturally be defined in FPL, just because of its call by need evaluation rule.

We have almost completed an experimental FPL interpreter, whose architecture is strictly related to the operational model of Section 3. The interpreter (written in LISP) is based on structure sharing and relies on LISP garbage collector.

Future work on FPL will include its extension to allow the definition of theorems and parallel programs. Our final goal is creating an FPL environment providing tools for program proving also.

REFERENCES

1. Backus,J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. C.ACM 21,8 (1978), 613-641.
2. Warren, D. Implementing PROLOG - compiling predicate logic programs. Report 39, Dept. of AI, Edinburgh, 1977.
3. Roussel,P. PROLOG: Manuel de Réference et d'utilisation, Groupe d'Intelligence Artificielle. Université d'Aix-Marseille.
4. McCabe,F.G. Programmer's guide to IC-PROLOG. Dept. of Computation and Control, Imperial College, London, 1978.
5. Roberts,G.M. An implementation of PROLOG. M.Sc. TH., Dept. of Computer Science, Univ. of Waterloo, 1977.
6. Szeredi,P. PROLOG- a very high level language based on predicate logic. 2nd Hungarian Computer Science Conf., Budapest, 1977.
7. Hewitt,C. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. AI Memo 231, MIT Project MAC, 1972.
8. Kowalski,R.A. Predicate logic as a programming language. Information Processing 74, North Holland (1974), 556-574.
9. vanHemden,M.H., and Kowalski,R.A. The semantics of predicate logic as a programming language. J.ACM 23,4 (1976), 733-742.
10. Warren,D., Pereira,L.M., and Pereira,F. PROLOG - the language and its implementation compared with LISP. Proc. ACM Symp. on AI and PL, Rochester, 1977.
11. Colmerauer,A. Le grammaires de metamorphose. Group d'Intelligence Artificielle. Université d'Aix Marseille, 1975.
12. Tärnlund,S-Å. Horn clause computability. BIT 17 (1977), 215-226.
13. Hansonn,Å. and Tärnlund,S-Å. A natural programming calculus. Proc. 6th IJCAI, Tokyo 1979.
14. Stickel,S. Invertibility of logic programs. Proc. 4th Workshop on Automated Deduction, Austin, 1979, 103-109.
15. Levi,G. and Sirovich,F. Proving program properties, symbolic evaluation and logical procedural semantics. Proc. MFCS '75. Lecture notes in Computer Science, Springer Verlag (1975), 294-301.
16. Burstall,R.M. Recursive programs: Proof, transformation and synthesis. Rivista di Informatica 7, 2 (1976), 25-42.
17. Aubin,R. Strategies for mechanizing structural induction. Proc. 5th IJCAI,Cambridge 1977, 363-369.
18. Boyer,R.S. and Moore,J S. A lemma driven automatic theorem prover for recursive function theory. Proc. 5th IJCAI, Cambridge, 1977, 511-519.
19. Cartwright,R. and McCarthy,J. First order programming logic. Proc. of 6th POPL, San Antonio, 1979, 68-80.
20. Burstall,R.M. and Goguen,J.A. Putting theories toghether to make specifications. Proc. 5th IJCAI, Cambridge, 1977, 1045-1058.
21. Goguen,J.A. and Tardo,J. OBJ-0 Preliminary user manual. Semantics and theory of computation Report, UCLA, 1977.
22. Guttag,J.V., Horowitz,E. and Musser,D.P. Abstract data types and software validation. C.ACM 21, 12 (1978), 1048-1063.
23. Musser,D.R. Abstract data type specification in the AFFIRM system. Proc. Specif a-tion of Reliable Software Conf., Boston, 1979.

HORN CLAUSE PROGRAMS SUGGESTED BY RECURSIVE FUNCTIONS

Jan ŠEBELÍK Petr ŠTĚPÁNEK

Department of Cybernetics and Operational Research
CHARLES UNIVERSITY, Praha, Czechoslovakia

1. Introduction

We shall construct a program in Horn logic for every partially recursive function by induction on the complexity of its definition. We shall show that these programs can be transformed into programs consisting only of binary Horn clauses. This gives a new proof of a result due to Tärnlund [4] , who used a binary Horn clause program to simulate the behaviour of a Universal Turing machine. Our proof gives additional information about the length of computations. We can show that for every partially recursive function, the computations of the original program and of the corresponding binary Horn clause program have the same number of steps on every input.

Reviewing the structure of programs suggested by induction on recursive functions, we shall see that these programs can be stratified in a natural way. We shall call stratifiable every program admitting stratification. The above result shows that every stratifiable program to compute a recursive function can be transformed in a binary Horn clause program. We shall show that every binary Horn clause program can be transformed in an inductive program computing the same function. At the end of paper, we shall formulate some open problems.

Throughout the paper, we shall use standard concepts and notation of Horn logic and of Resolution logic. We refer the reader to [2] , [3] and [5] for a more detailed exposition. We shall mostly deal with first-order languages without equality containing only two function symbols 0 and S , where 0 is a constant interpreted as zero and unary function symbol S is interpreted as the successor function $s(x) = x + 1$. Hence, the terms 0, $S(0)$, $SS(0)$, ... can be identified with natural numbers $0, 1, 2, \ldots$. Every expression $P(t_1, t_2, \ldots, t_p)$, where P is a p-ary predicate symbol and $t_1, \ldots, t_p$ are

terms, is called an atomic formula or an atom. The atomic formulas and the negations of atomic formulas are called literals; the atoms and the negations of atoms are called positive literals and negative literals respectively. A clause is a disjunction of literals, a Horn clause is a clause with at most one positive literal. A binary Horn clause is a Horn clause with at most one negative literal. Every conjunction of (Horn) clauses is called a (Horn) sentence. The clauses are usually represented as lists of literals and sentences are identified with sets of clauses. We shall sometimes speak about unions of sentences etc.

It is useful to express Horn clauses in the following way

(1) $$A \longleftarrow B_1, B_2, \cdots, B_n$$

where A is a positive literal (if any) and $B_1, \cdots, B_n$ are negative literals (if $n > 0$).

We shall use special names for the following types of Horn clauses:

(i) A Horn clause without a positive literal is called a goal statement of a goal and it is called an empty clause or a halt statement if it has no negative literals. Empty clause is denoted by $\square$.

(ii) A Horn clause with exactly one positive literal is called regular. A regular clause without negative literals is called an assertion or an axiom , otherwise it is called a procedure or a procedure declaration, the literal A in (1) being called the name and literals $B_1, \cdots, B_n$ being called the body of procedure (1).

(iii) Every sentence which is a conjunction of regular clauses is called regular sentence.

The Resolution principle with unification as a pattern matching algorithm is the only Inference Rule of so called Horn Logic. We assume that the reader is familiar with these concepts and we shall illustrate them by the following example.

Example 1 . The clauses

(2) PLUS(x, 0, x) ⟵

(3) PLUS(x, Sy, Sz) ⟵ PLUS(x, y, z)

state respectively that for every natural number x , the sum of x and zero is equal to x and that the sum of x and Sy (the successor of y) is Sz whenever z is the sum of x and y. The regular sentence consisting of regular Horn clauses (2) and (3) can be used as a program computing addition of natural numbers. Suppose e.g. that we want to compute the sum 1 + 1 . We start by writing a goal statement which denies that there is a number z which is the sum of SO and SO. The following goal statement is the first step in computation that yields the desired sum. The assignments of variables are the effect of unification.

(4) ⟵ PLUS(SO, SO, z)

$x \longleftarrow SO$, $y \longleftarrow 0$, $z \longleftarrow Sz'$

(5) ⟵ PLUS(SO, 0, z′) by (3), (4)

$z' \longleftarrow SO$

by (5), (2).

We obtain $z \longleftarrow SSO$ by composition of both assignments.

We have seen that regular sentences can be used as programs and we shall call every regular sentence a Horn clause program. In general, we shall proceed as follows. If S is a Horn clause program and C_o is a goal statement, the sequence $C_o, C_1, \dots, C_n$ of goal statements is said to be a deduction (computation) from S and C_o provided that for every i , C_{i+1} is the resolvent of C_i and a clause D from S such that the leftmost atom of C_i and the only positive literal of D are the literals the clauses C_i and D are resolved upon. If the last clause of a deduction is empty, we say that the deduction refutes S and C_o . It follows from the properties of Linear Ordered Resolution and from the fact that every regular sentence is satisfiable that $S \cup \{C_o\}$ is not satisfiable iff there is a deduction starting with C_o and refuting S and C_o . To find such a refutation, one has to use a complete search strategy. We shall adopt the following notation, we shall write

$$C_i \vdash_S C_j$$

to say that the goal C_j , $j > i$ is deducible from C_i and S. If P is a ground atom (i.e. a variable-free atomic formula), we shall write

$$S \vdash P$$

instead of

$$\leftarrow P \vdash_S \square \ .$$

If S is the program from Example 1 then $S \vdash PLUS(SO,SO,SSO)$ expresses the result of computation, namely, that the term SSO corresponds to the sum $1 + 1$.

The following easy lemma is used quite often in proofs.

Lemma 1. Let S be a Horn clause program, P a ground atom. Let there be only one regular clause

$$P' \leftarrow Q_1, Q_2, \dots, Q_n$$

in S such that P' can be unified with P by a most general unifier σ .

Then $S \vdash P$ iff there is a substitution η such that

$S \vdash Q_i\sigma\eta$ holds for every $i \leq n$,

all $Q_i\sigma\eta$ being ground atoms.

2. Programs for recursive functions

We shall describe programs for all partially recursive functions by induction on the complexity of their definitions. We start with basic functions and then we shall construct programs for functions obtained by composition, recursion and minimization. The corresponding programs are naturally motivated and, as we shall see later, they admit a certain stratification. Now, we shall prove the following

Theorem 1. For every partially recursive function f of n variables, there is an $(n + 1)$-ary predicate symbol F_f and a Horn clause program P such that for every sequence of natural numbers $a_1, \dots, a_n, c$,

(6) $f(a_1, \dots, a_n)$ is defined and equal to c iff

$F_f(a_1, \dots, a_n, c)$ is provable from P .

Proof. We shall omit the subscript f by the predicate F. If f is one of the basic functions, then P consists of one assertion, namely, P is

(i) $F(x, 0) \longleftarrow$

provided that f is the zero function, $f(x) = 0$ for every x,

(ii) $F(x, Sx) \longleftarrow$

whenever f is the successor function $f(x) = x + 1$, and

(iii) $F(x_1, \dots, x_n, x_i) \longleftarrow$

whenever f is the projection on the i-th coordinate.

(iv) Let f be obtained by composition, let $f(x) = h(g_1(x), \dots, g_k(x))$, where g_i for $i = 1, 2, \dots, k$ is a function of n variables. Let $H, G_1, \dots, G_k$ be the predicate symbols corresponding to $h, g_1, \dots, g_k$ and let P_o be a program computing h and P_i for $i \leq k$ be programs computing g_i respectively. By induction hypothesis, we have

$$g_i(a) = b_i \quad \text{iff} \quad P_i \vdash G_i(a, b_i) \quad \text{for} \quad i \leq k,$$

(7)

$$h(b) = c \quad \text{iff} \quad P_o \vdash H(b, c),$$

for every n-tuple a of natural numbers and natural numbers $b_1, \dots, b_k, c$, where $b = (b_1, \dots, b_k)$. We may assume that every two programs P_i, P_j have no predicate symbol in common.

Let F be a new $(n + 1)$-ary predicate symbol and $x_1, \dots, x_n, y_1, \dots, y_k, z$ be new variables. If we add the clause

(8) $F(x, z) \longleftarrow G_1(x, y_1), G_2(x, y_2), \dots, G_k(x, y_k), H(y, z)$

where x and y are appropriate tuples of variables, to the union of sentences $P_o, P_1, \dots, P_k$, we obtain a program P satisfying (6).

(v) Let f be obtained from functions g and h by primitive recursion, i.e.

$$f(0, x_2, \dots, x_n) = g(x_2, \dots, x_n),$$

$$f(Sx_1, x_2, \dots, x_n) = h(x_1, \dots, x_n, f(x_1, \dots, x_n))$$

holds for every $x_1, \dots, x_n$. Let G, H be predicate symbols corresponding to g and h and let P_1, P_2 be the programs for g and h. We assume that both programs do not share any predicate symbol. Let F be a new $(n + 1)$-ary predicate symbol. It is easy to see that the Horn clause program P which is obtained by adding the clauses

(9) $F(0,x_2, \ldots ,x_n, z) \leftarrow G(x_2,\ldots,x_n, z)$
$F(Sx_1,x_2,\ldots,x_n,z) \leftarrow F(x_1,\ldots,x_n,y),\ H(x_1,\ldots,x_n,y,z)$

to the union of P_1, P_2 satisfies (6).

(vi) The last case to be considered is when f is obtained by minimization of a computable function g, i.e.

$$f(x) = \mu y[g(x,y) = 0] \quad ,$$

where x is an n-tuple of variables. Let G be the $(n+2)$-ary predicate symbol corresponding to g and P_1 be a program for g. Let R and F be new $(n+1)$-ary predicate symbols and let P is obtained from P_1 by adding the following clauses

(10)
$$R(x,0) \leftarrow ,$$
$$R(x,Sz) \leftarrow R(x,z),\ G(x,z,sv) \quad ,$$
$$F(x,z) \leftarrow R(x,z),\ G(x,z,0) \quad ,$$

where $x = (x_1,\ldots,x_n)$.

To show that P has the desired property, it suffices to prove that for every natural number c and every n-tuple a of natural numbers, we have

(11) $P \vdash R(a,c)$ iff $(\forall x < c)(g(a,x)$ is defined and nonzero).

This can be proved by induction on c and concludes the proof of Theorem 1.

The result stated in Theorem 1 was first proved by Tärnlund [4] who showed that every Turing computable function is computable by a binary Horn clause program. Independently, Andreka and Nemeti [1] showed that no auxiliary symbols are needed in a Horn clause program for a computable function provided that the language contains individual constants for all natural numbers (see also [5]).

We would like to turn attention to specific properties of programs constructed in the proof of Theorem 1. The step by step construction of recursive functions from basic functions to functions that are obtainable by recursive operations give rise to a natural hierarchy. Every partially recursive function can be assigned a rank according to the level of the hierarchy on which

it appears. Our programs reflect the hierarchy of recursive functions in the following way: the predicates corresponding to recursive functions can be uniformly stratified in every program by a mapping that assigns a natural number to every predicate symbol. It is convenient to assign zero to the predicate symbol occuring in the goal statement and to other predicate symbols assign numbers in the inverse order with respect to the ranks of corresponding functions. This motivates the following definition.

Let P be a Horn clause program and A be a predicate symbol that occurs in P. We say that P admits stratification with respect to A if there is a mapping s that assigns a natural number to every predicate symbol in P such that the following conditions hold

(i) $s(A) = 0$

(ii) if the clause $Q(\ldots) \leftarrow R_1(\ldots),\ldots,R_k(\ldots)$ belongs to P, then every $i = 2,3,\ldots,k$, we have

(12) $s(R_i) = s(Q) + 1$,

and for $i = 1$ either R_i is Q or (12) holds.

We say that a program P is stratifiable is it admits stratification with respect to one of its predicate symbols.

It should be stressed that the mapping s assigns natural numbers to predicate symbols, not to the atoms containing them. It is not hard to check that the programs constructed in the proof of Theorem 1 are all stratifiable, admitting stratification with respect to predicate symbol F_f. Thus we can reformulate the result as follows:

Theorem 1'. Every partially recursive function is computable by a stratifiable Horn clause program.

3. Binary Horn clause programs

We shall show that every stratifiable program constructed in previous Section can be converted into a binary Horn clause program. This gives a new proof of the above mentioned result of Tärnlund. Our proof gives additional information about the length

of computations. In fact, we can show that for every recursive function f the computations of the stratifiable program for f and of the corresponding binary program have the same number of steps for every input. Thus the binary Horn clause programs do not increase the complexity of computation in comparison with the "natural" stratifiable programs.

Theorem 2. Every partially recursive function is computable by a binary Horn clause program.

Moreover, for every such function, it is possible to transform the corresponding stratifiable program from Theorem 1 in a binary Horn clause program such that the lengths of computations of both programs are the same for every input.

Sketch of proof. We shall proceed by induction on recursive functions and describe the steps to transform the corresponding stratifiable program in a binary program.

If f is one of the basic functions, then the programs described in (i) - (iii) of the proof of Theorem 1 are already binary. Let f be obtained by composition. As in the case (iv) of the previous proof, we shall assume that G_i , $i = 1,\ldots,k$ are $(n+1)$-ary predicate symbols corresponding to functions g_i and that T_i is a binary program for g_i . Let H be the $(k+1)$-ary predicate symbol corresponding to h and let T_o be a binary program for h . We may assume that any two programs have no predicate symbol in common. To avoid the clause (8) which is not binary, we have to modify programs T_i .

Let $x_1,\ldots,x_n,y_1,\ldots,y_k,z$ be new variables which do not occur in any program T_i . We shall add these variables to all atoms in programs T_i for $i = 1,2,\ldots,k$ and we shall convert all assertions of these programs into suitable procedure declarations to establish ties between the resulting programs. For every $i = 1,2,\ldots,k$, let P_i be a binary program obtained from T_i as follows :

(13) Every clause $R(t_1,\ldots,t_p) \longleftarrow Q(u_1,\ldots,u_q)$ of T_i is replaced by

$$\hat{R}(t_1,\ldots,t_p,x_1,\ldots,x_n,y_1,\ldots,y_k,z) \longleftarrow \hat{Q}(u_1,\ldots,u_q,x_1,\ldots,x_n,y_1,\ldots,y_k,z) \ ,$$

where $\hat{R}$, $\hat{Q}$ are new predicate symbols.

(14) Every assertion $R(t_1,\ldots,t_r) \longleftarrow$ of T_i is replaced by

$$\hat{R}(t_1,\ldots,t_r,x,y,z) \longleftarrow \hat{G}_{i+1}(x,y_{i+1},x,y,z) ,$$

where $\hat{R}$ is as above and $\hat{G}_{i+1}$ is a new $(n+1+n+k+1)$-ary predicate symbol corresponding to G_{i+1}.

For $i = k$, we apply (13) and every assertion $R(\ldots) \longleftarrow$ is replaced by $\hat{R}(\ldots,x,y,z) \longleftarrow H(y,z)$.

Let P_i be the resulting programs. Now, we can state the following fact. For every ground atom $P(\ldots)$ and appropriate tuples of natural numbers a, b, c we have

$$T_i \vdash P(\ldots) \quad \text{iff} \quad \longleftarrow \hat{P}(\ldots,a,b,c) \;\vdash_{P_i}\; \longleftarrow \hat{G}_{i+1}(a,b_{i+1},a,b,c)$$

(15) holds for every $i = k-1$, and

$$T_k \vdash P(\ldots) \quad \text{iff} \quad \longleftarrow \hat{P}(\ldots,a,b,c) \;\vdash_{P_k}\; \longleftarrow H(b,c) .$$

The lengths of deductions on both sides are the same.
This fact can be proved by induction on the length of deduction.
If we add the clause

(16) $F(x,z) \longleftarrow \hat{G}_1(x,y_1,x,y,z)$,

where x and y are appropriate tuples of variables, to the union of programs $P_1,\ldots,P_k,T_o$, we obtain a binary program P for f. If T is the program obtained by adding (8) to the union of T_i's, it is not hard to prove from (15) and Lemma 1 that the computations of T and P have the same length for every input.

The cases, where f is obtained by primitive recursion or minimization are treated similarly. One has to deal with recursive calls of procedures. To give the idea, we shall illustrate the case of recursion by the following example.

Example 2. Let f be defined by primitive recursion as follows $f(1) = 1$, $f(x+1) = f(x) + (x+1)$. A stratifiable program for f consists of clauses

(17) $F(0,S0) \longleftarrow$

(18) $F(Sx,z) \longleftarrow F(x,y), PLUS(y,Sx,z)$

and of the clauses from example 1, namely,

$PLUS(x,0,x) \longleftarrow$

$PLUS(x,Sy,Sz) \longleftarrow PLUS(x,y,z)$.

If v,w are new variables, we can write the following binary program P for f.

(19) $F(0,S0) \leftarrow$

(20) $F(Sx,z) \leftarrow P(y,sx,z,x,y)$

(21) $P(x,0,x,v,w) \leftarrow F(v,w)$

(22) $P(x,Sy,Sz,v,w) \leftarrow P(x,y,z,v,w)$.

If we denote the original program by T , we can compare the computations of T and P starting with the goal $\leftarrow F(SS0,z)$.

$\leftarrow F(SS0,z)$
$\leftarrow F(S0,y),PLUS(y,SS0,z)$
$\leftarrow F(0,y'),PLUS(y',S0,y),PLUS(y,SS0,z)$
$\leftarrow PLUS(S0,S0,y),PLUS(y,SS0,z)$
$y \leftarrow Sv$
$\leftarrow PLUS(S0,0,v),PLUS(Sv,SS0,z)$
$\leftarrow PLUS(SS0,SS0,z)$
$\leftarrow PLUS(SS0,S0,z')$
$\leftarrow PLUS(SS0,0,z'')$
$\square$

$\leftarrow F(SS0,z)$
$\leftarrow P(y,SS0,z,S0,y)$
$\leftarrow P(y,S0,z',S0,y)$ $z \leftarrow Sz'$
$\leftarrow P(y,0,z,S0,y)$ $z' \leftarrow Sz$
$\leftarrow F(S0,z'')$ $y \leftarrow z'$
$\leftarrow P(y,S0,z'',0,y)$ $z'' \leftarrow Sz$
$z \leftarrow Sz'$ $\leftarrow P(y,0,z''',0,y)$ $y \leftarrow z'''$
$z' \leftarrow Sz''$ $\leftarrow F(0,z''')$
$z'' \leftarrow SS0$ $\square$ $z''' \leftarrow S0$

Both programs are deterministic : at every step of computation there is only one procedure which can be unified with the goal statement. In general case, it is possible that T is deterministic in the above sense but P is not. In other words, it may happen that the length of successful computations of both programs are the same, but the search for a successful computation of P may be more complex due to nondeterminism. It should be noted that the program P from Example 2 is not stratifiable.

4. Stratifiable and binary Horn clause programs

We have seen that the stratifiable programs motivated by recursive functions can be transformed into binary programs computing the same function. Surprisingly, both programs have the same length of computations on every input. On the other hand, it can be shown that every binary program cen be transformed in a stratifiable program. We have

Theorem 3. Let T be a binary Horn clause program, A a predicate symbol occuring in T . It is possible to transform T into a stratifiable Horn clause program P such that for every ground atom A' containing A , we have

$T \vdash A'$ iff $P \vdash A'$.

The proof of Theorem 3 uses induction on the height of a computation tree of T . The main problem is to deal with multiple recursive calls of procedures. We shall not give the proof here. In comparison with the two previous theorems, the proofs of which depend strongly on the language of Resolution Arithmetic, in particular on the representation of natural numbers by terms $S^n(0)$, Theorem 3 can be proved for every first-order language. In general case however, the resulting stratifiable program need not be binary.

The above results indicate that there is a strong relationship between binary and stratifiable program, at least in the case of Resolution Arithmetic. There are some open problems left. We have shown that certain stratifiable programs in the language of (successor) arithmetic can be transformed into binary programs.

Problem 1.

Is it possible to transform every stratifiable Horn clause program into a binary program ?

Note that the proof of Theorem 2 used special features of programs motivated by recursive operations. Another problem is motivated by Theorem 3. It is clear that every Horn clause program computing a partially recursive function has a stratifiable couterpart computing the same function. This follows from Theorem 1'. Generalizing Theorem 3 to the case of arbitrary Horn clause programs, one has to deal with AND/OR trees instead of simple trees for binary programs. One of the possible ways to handle the problem may be coding information about branching in natural numbers.

Problem 2.

Is it possible to transform every Horn clause program into a stratifiable program computing the same function ? What are the necessary conditions for the language ?

REFERENCES

[1] H.Andreka, I.Nemeti, The generalized Completness of Horn Predicate Logic as a programming language, D.A.I. Ressearch Rep. 21, Dept. AI, University of Edinburgh 1976

[2] C.L.Chang, R.T.C. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, NY 1971

[3] R. Kowalski, Logic for problem solving Memo 75, Department of Computational Logic University of Edinburgh 1975

[4] S.Å. Tärnlund, Logic Information processing, TRITA - IBADB - 1029 1975-11-24, dept. Comp. Sci., Royal Institute of Tech. Stockholm 1975

[5] S.Å. Tärnlund, Horn Clause Computability BIT 17 (1977), 215 - 226

A decision method for process logic

Olga Štěpánková
Institute for Computation techniques
of the ČVUT, Horská 3, Praha 2,
Czechoslovakia

Process logic was proposed in [1] as a tool for reasoning about on going processes. There are introduced modal constructs after $\lrcorner$, throught $\sqcup$, during $\perp$ and preserves $\lceil$, expressing important properties of programs. Their intuitive meaning is

$a\lrcorner p$ - any halting computation of a ends in a state satisfying p
$a\sqcup p$ - in every state of computation of a holds p
$a\perp p$ - sometimes during any computation of a occures a state satisfying p
$a\lceil p$ - ones p is true in a computation of a, it holds on in all the following states.

The modal concepts $\lrcorner$, $\sqcup$, $\lceil$ are completely axiomatized in [1]. The axiomatization of these modalities combined with $\perp$ is formulated in [1] as an open problem. The modality $\perp$ is defined in [2] differently. There is given axiomatization of $\lrcorner$, $\sqcup$, $\lceil$ and of this new version of $\perp$ - no claim is mode about its completeness. We propose a complete decision method for $\lrcorner$, $\sqcup$, $\lceil$ and $\perp$.

Syntax of the language

Let a be a constant differing from all the propositional variables. Let formulas be the elements of the minimal class including propositional variables and closed under the following rule:

"if p, q are formulas then

$p \wedge q$, $\neg p$, $p \vee q$, $a\lrcorner p$, $a\sqcup p$, $a\lceil p$ and $a\perp p$

are formulas as well".

Semantics

Let $\mathcal{W}$ be a set of states: a trajectory $s = (s_0, s_1, \ldots, s_{m_s})$ is a sequence of states drawn from $\mathcal{W}$; m_s is the length of s (when m_s is infinite, then s has no final state).

A failing computation is indicated by a trajectory whose final state is Λ, a distinguished "limbo" state used only for this purpose this case covers deadlock, abort and fail situations.

Any occurrence of Λ in a trajectory must be as the final state.
An interpretation of the language is given by a structure $\langle \mathcal{W}, \mathcal{P}, \mathcal{T} \rangle$, where $\mathcal{W}$ is a set of states, $\mathcal{P}$ a set of "facts" about states (for any propositional variable p and $w \in \mathcal{W}$ either $\langle p, w \rangle \in \mathcal{P}$ or $\langle \neg p, w \rangle \in \mathcal{P}$, not both), $\mathcal{T}$ a set of trajectories from $\mathcal{W}$.

The set $\mathcal{T}$ describes a program a in this sense: if $s = s_0, s_1, \dots, s_{m_s} \in \mathcal{T}$, then the program a started in s_0 can bring about the trajectory s.

Validity of formulas of the considered language in a given state w of this interpretation is defined as follows:

if p is a propositional variable then

$$w \models p \quad \text{iff} \quad \langle p, w \rangle \in \mathcal{P}$$

otherwise

$$w \models \neg p \quad \text{iff} \quad w \not\models p$$

$$w \models p \wedge q \quad \text{iff} \quad w \models p \text{ and } w \models q$$

$$w \models a \lrcorner p \quad \text{iff} \quad (\forall s \in \mathcal{T})(s_0 = w \wedge s_{m_s} \neq \Lambda \supset s_{m_s} \models p)$$

$$w \models a \sqcup p \quad \text{iff} \quad (\forall s \in \mathcal{T})(s_0 = w \supset \forall s_z \neq \Lambda\, (s_z \models p))$$

$$w \models a \lfloor p \quad \text{iff} \quad (\forall s \in \mathcal{T})(s_0 = w \supset \forall s_y s_z \neq \Lambda (y \leq z \supset (s_y \models p \supset s_z \models p)))$$

$$w \models a \sqcap p \quad \text{iff} \quad (\forall s \in \mathcal{T})(s_0 = w \supset \exists s_z \neq \Lambda\, (s_z \models p))$$

A formula p with at least one occurrence of a is satisfiable iff there exists a structure $\langle \mathcal{W}, \mathcal{P}, \mathcal{T} \rangle$ and a state $w \neq \Lambda$ of $\mathcal{W}$ such that there is at least one sequence $s \in \mathcal{T}$ beginning in w (i.e. $s_0 = w$) and $w \models p$. A finite set of formulas (a theory) is satisfiable iff the conjunction of all its formulas is satisfiable. A formula p is valid iff $\neg p$ is unsatisfiable.

A decision method

We follow [1] in the use of Gentzen systems manipulating sets of formulas.

Let $\mathcal{P}, \mathcal{R}$ be finite sets of formulas:
A sequent corresponding to a theory $\mathcal{P} \cup \{\neg r : r \in \mathcal{R}\}$ is denoted $\mathcal{P} \to \mathcal{R}$.

The theory $P \cup \{\neg r : r \in R\}$ is unsatisfiable iff the formula $\wedge P \to \vee R$ is valid. A sequent $P \to R$ is valid iff the theory $P \cup \{\neg r : r \in R\}$ is unsatisfiable (iff $\wedge P \to \vee R$ is valid).

All properties of propositional calculus can be derived by the use of

$$P1\ \frac{}{p \to p} \qquad P2\ \frac{p \to}{\to \neg p} \qquad P3\ \frac{p, q \to}{p \wedge q \to} \qquad P4\ \frac{\to p \quad \to q}{\to p \wedge q}$$

It is proved in 1 that if the rules

$$\lfloor 1\ \frac{p, a \lfloor p \to \quad . \quad a \lfloor p \to p}{a \lfloor p \to}$$

$$\lfloor 2\ \frac{\{p_1, \dots, p_j, q_{j+1}, \dots, q_k \to r\}}{a \lfloor p_1, \dots, a \lfloor p_j, a \sqcup q_{j+1}, \dots, a \sqcup q_k \to a \lfloor r}$$

$$\lfloor 3\ \frac{\{p_1, \dots, p_j \to q \equiv f(p_{j+1}, \dots, p_k)\}\ (f \text{ monotonic})}{a \sqcup p_1, \dots, a \sqcup p_j, a \lfloor p_{j+1}, \dots, a \lfloor p_k \to a \lfloor q}$$

are added, the resulting system is a complete axiomatisation of $\sqcup$, $\lfloor$ and $\bot$.

Let us use the following conventions - $\mathfrak{i}, \mathfrak{j}, \mathfrak{k}$ are either natural numbers or 0. The set $\mathfrak{N}_{\mathfrak{k}}$ includes just all the permutations of the numbers $\{1, -, \mathfrak{k}\}$, i.e. all 1 - 1 functions of this set on itself. If R is a set of formulas, then $\neg R$ is an abbreviation for $\{\neg r : r \in R\}$. If P_1, P_2 are sets of formulas and s, p are formulas, then

$$P_1, P_2, s \to p$$

denotes just the sequent $P_1 \cup P_2 \cup \{s\} \to \{p\}$. We write <u>false</u> instead of notoriously invalid formula $p \wedge \neg p$ and <u>true</u> instead of its negation.

$$\bot 0\ \frac{\{p \to a \bot \underline{false}\}}{\{p \to\}}$$

$$\bot 1\ \frac{(\forall P_1 \subseteq P_2 \subseteq \dots \subseteq P_{\mathfrak{j}} = \{p_j\}_{j<\mathfrak{j}})(\forall \pi \in \mathfrak{N}_{\mathfrak{k}})(\exists k_0 < \mathfrak{k})\ \{\{s_i\}_{i<\mathfrak{i}}, P_{k_0+1}, \neg(P_{\mathfrak{j}} - P_{k_0}), q_{\pi(k_0)} \to \sigma\}}{\{a \sqcup s_i\}_{i<\mathfrak{i}}, \{a \lfloor p_j\}_{j<\mathfrak{j}}, \{a \bot q_k\}_{k<\mathfrak{k}} \to a \bot \sigma}$$

$$\perp 2 \quad \frac{(\forall \beta_1 \subseteq \dots \subseteq \beta_\varkappa \subseteq \beta_0 = \{p_j\}_{j<\gamma})(\forall \pi \in \mathfrak{P}_\varkappa)(\exists k_0 < \varkappa)}{(\{s_i\}_{i<\gamma}, \beta_{k_0}, \neg(\beta_0 - \beta_{k_0}), q_{\pi(k_0)} \to \} \text{ or } \{\{s_i\}_{i<\gamma}, \beta_\varkappa, e' \to e\})}$$
$$a \perp e', \{a \amalg s_i\}_{i<\gamma}, \{a \sqcap p_j\}_{j<\gamma}, \{a \perp q_k\}_{k<\varkappa} \to a \perp e$$

$$\perp 3 \quad \frac{(\forall \beta_1 \subseteq \dots \subseteq \beta_{\varkappa+2} \subseteq \beta_0 = \{p_j\}_{j<\gamma})(\forall \pi \in \mathfrak{P}_{\varkappa+2})(\exists k_0 < \varkappa+2)}{(q_{\varkappa+1} = r \wedge q_{\varkappa+2} = r \Rightarrow (\{\{s_i\}_{i<\gamma}, \beta_{k_0}, \neg(\beta_0 - \beta_{k_0}), q_{\pi(k_0)} \to\} \text{ or } \pi^{-1}(\varkappa+1) > \pi^{-1}(\varkappa+2)))}$$
$$\{a \amalg s_i\}_{i<\gamma}, \{a \sqcap p_j\}_{j<\gamma}, \{a \perp q_k\}_{k<\varkappa} \to a \sqcap r$$

The intended reading of $\perp 1$ is :

"Let $\{s_i\}_{i<\gamma}, \{p_j\}_{j<\gamma}, \{q_k\}_{k<\varkappa}, \{\alpha\}$ be the sets of formulas of process logic. Denote $\beta_0 = \{p_j\}_{j<\gamma}$. If for any $\beta_1 \subseteq \dots \subseteq \beta_\varkappa \subseteq \beta_0$ and any 1-1 function π of the set $\{1, \dots, \varkappa\}$ on itself there exists $k_0 < \varkappa$ such that

$$\{s_i\}_{i<\gamma}, \beta_{k_0}, \neg(\beta_0 - \beta_{k_0}), q_{\pi(k_0)} \to \alpha$$

is a valid sequent, then

$$\{a \amalg s_i\}_{i<\gamma}, \{a \sqcap p_j\}_{j<\gamma}, \{a \perp q_k\}_{k<\varkappa} \to a \perp \alpha$$

is valid too".

Similarly for the rules $\perp 2$, $\perp 3$.

The rule $\perp 0$ is added to cover the case of sequents of the form

$$\{\{a \amalg s_i\}_{i<\gamma}, \{a \sqcap p_j\}_{j<\gamma}, \{a \perp q_k\}_{k<\varkappa} \to\}.$$

In other words, it eliminates the possibility that a structure with a base state w, in which no sequence starts, would be considered to be a model for e.g. $\{a \amalg p \to\}$.

Theorem 1 :

The rules $\perp 0, \perp 1, \perp 2, \perp 3$ are sound, i.e. if the premises of a rule are met, then the sequent resulting from this rule is valid.

Proof :

Suppose, that the sequent resulting from e.g. $\perp 1$ is not valid. Thus there exists $\langle W, \mathcal{P}, \mathcal{C} \rangle$ and $w \in W$ such that

$$w \models \{ a \sqcup s_i \}_{i \in I} \cup \{ a \sqcap p_j \}_{j \in J} \cup \{ a \perp q_k \}_{k \in K} \cup \{ \neg a \perp \alpha \}$$

Let us considere the trajectory $v \in \mathcal{C}$ such that $v_1 = w$ and $\forall w_i \neq \Lambda \; (w_i \models \neg \alpha)$. It can be shown, that the study of this trajectory leads to the specification of $\mathcal{P}_1 \subseteq \dots \subseteq \mathcal{P}_K \subseteq \mathcal{P}_0 = \{p_i\}_{i \in I}$ and of the permutation $\pi \in \mathfrak{P}_K$ such that they contradict the assumption of the rule $\perp 1$.

Similarly for $\perp 2, \perp 3$.

Considering the rule $\perp 0$, let us distinguish the cases.

a/ there is no occurrence of a in p

b/ there is an occurrence of a in p .

a/ If there is a state v satisfying p , denote $\mathcal{P} = \{ q : q \text{ propositional variable}, v \models q \}$. Then the structure $\langle \{w, \Lambda\}, \mathcal{P}, \{ \langle w, \Lambda \rangle \} \rangle$ with the base state v is a model of $p \rightarrow a \perp \underline{\text{false}}$.

b/ The formula $\neg a \perp \underline{\text{false}}$ is true in the base state of any structure satisfying p .

A proof is a rooted tree whose verteces are labelled with sequents such that every sequent follows from its predecessors by some rule. Given a proof we say that the sequent labelling the root is proved from the assumptions, which are just all the sequents on the leaves of that proof.

Let $\varphi \cdot \varphi_1 \to \varphi_2$ be a sequent. The corresponding theory $\mathcal{T}_\varphi$ is $\varphi_1 \cup \{\neg s : s \in \varphi_2\}$. We say that φ is a <u>complete sequent</u> iff the corresponding theory $\mathcal{T}_\varphi$ meets all the following conditions:

1. $\alpha \lceil p \in \mathcal{T}_\varphi$ then $p \in \mathcal{T}_\varphi$ or $\neg p \in \mathcal{T}_\varphi$
2. $p \wedge q \in \mathcal{T}_\varphi$ then $p \in \mathcal{T}_\varphi$ and $q \in \mathcal{T}_\varphi$
3. $p \vee q \in \mathcal{T}_\varphi$ then $p \in \mathcal{T}_\varphi$ or $q \in \mathcal{T}_\varphi$
4. $\neg(p \wedge q) \in \mathcal{T}_\varphi$ then $\neg p \in \mathcal{T}_\varphi$ or $\neg q \in \mathcal{T}_p$
5. $\neg(p \vee q) \in \mathcal{T}_\varphi$ then $\neg p \in \mathcal{T}_\varphi$ and $\neg q \in \mathcal{T}_\varphi$

<u>Lemma 1</u> :

Let φ be a sequent. There can be found a finite set $\mathfrak{B}_\varphi$ of complete sequents such that

a/ φ can be proved from the assumptions $\mathfrak{B}_\varphi$ using only the rules P1 - P4 and $\lceil$ 1.

b/ if φ is valid, then any $\mathcal{B} \in \mathfrak{B}_\varphi$ is valid as well.

<u>Proof</u> :

Let $\mathcal{U}$ be a set of all theories associated with complete sequents of process logic. Let us construct a sequence of sets of theories :

We start with $\mathfrak{B}_0 = \{\mathcal{T}_\varphi\}$ and given $\mathfrak{B}_i$ the element $\mathfrak{B}_{i+1}$ is composed from $\mathfrak{B}_i \cap \mathcal{U}$ and from all the theories obtained from the members of $\mathfrak{B}_i$ by the use of conditions on completness of sequents as follows :

If there is $\mathcal{R} \in \mathfrak{B}_i$ and p such that $\alpha \lceil p \in \mathcal{R}$ but $p \notin \mathcal{R}$ and $\neg p \notin \mathcal{R}$, then both $\mathcal{R}_1 = \mathcal{R} \cup \{p\}$ and $\mathcal{R}_2 = \mathcal{R} \cup \{\neg p\}$ are elements of $\mathfrak{B}_{i+1}$.

Similarly for the conditions 2, -, 5.

Obviously there is m_0 such that $\mathfrak{B}_{m_0} - \mathfrak{B}_{m_0+1}$. $\mathfrak{B}_{m_0} \cap \mathfrak{U}$ is the searched set $\mathfrak{B}_\varphi$ of theories. The property a/ is a consequence of the way of construction of $\mathfrak{B}_{i+1}$ from $\mathfrak{B}_i$. Moreover any theory $\mathfrak{G} \in \mathfrak{B}_i$ includes all the formulas from $\mathfrak{T}_\varphi$ - the theory corresponding to φ. Thus if $\mathfrak{T}_\varphi$ has no model, then $\mathfrak{G}$ cannot have a model as well, i.e. all the sequents from $\mathfrak{B}_\varphi$ are valid if φ is valid.

Lemma 2 :

Each complete valid sequent of process logic has a proof using the rules P, $\perp$ and $\lceil 1$ only.

Proof :

Let us define duration of a formula of process logic by induction on construction of formulas :

- duration of a propositional variable p is $d(p) = 0$
- let q_1, q_2 be formulas of a known duration, then

$$d(a \lceil q_1) = d(a \rfloor q_1) = d(a \perp q_1) = d(a \sqcup q_1) = d(q_1) + 1$$

$$d(q_1 \wedge q_2) = d(q_1 \vee q_2) = \max(d(q_1), d(q_2))$$

$$d(\neg q_1) = d(q_1).$$

Let us define a relation between sequents of process logic :

$\varphi_1 \prec \varphi_2$ iff
- maximal duration of formulas in φ_1 is smaller then that of φ_2

or

- maximal durations of formulas in φ_1 and φ_2 are equal, but there is less formulas in φ_1 of this duration then in φ_2.

Obviously $\prec$ is a good ordering.

Suppose the lemma does not hold, i.e. there is a complete valid sequent, which has no proof. We shall drive this to contradiction.

Let $\mathcal{G}_0$ be $\prec$ minimal element of the class of all complete valid sequents without a proof. Let $\mathfrak{T}_0$ be the theory corresponding to $\mathcal{G}_0$ and $\mathcal{M}_0$ be the set of all formulas from $\mathfrak{T}_0$ of the maximal duration.

The set $\mathcal{M}_0$ must contain only formulas of the form $\alpha \perp p, \alpha \lceil p, \alpha \rfloor p$ and $\alpha \sqcup p$ or their negations.

Two cases must be distinguished :

a/ Let there be a negative occurrence of a modality in $\mathcal{M}_0$, i.e. $\neg \alpha \perp \sigma \in \mathcal{M}_0$. Then $\mathfrak{T}_0 \cup \{\neg \alpha \perp \sigma\}$ cannot be valid, because it corresponds to a sequent $\mathcal{G}_1 \prec \mathcal{G}_0$, whose validity implies the existence of the proof of $\mathcal{G}_1$, hence of $\mathcal{G}_0$, too. If $\perp 1$ is not appliable to $\mathcal{G}_0$, then there can be constructed a structure satisfying $\mathfrak{T}_0$ by adding to the structure for $\mathfrak{T}_0 \cup \{\neg \alpha \perp \sigma\}$ a new trajectory beginning in its base state and visiting consequently the structures for sequents as named in the example excluding the possibility to apply $\perp 1$. Contradiction with assumption $\mathcal{G}_0$ is valid. But if $\perp 1$ is appliable to $\mathcal{G}_0$, this is the first step of the proof of $\mathcal{G}_0$, because the application of $\perp 1$ leads to valid sequents, which are $\prec$ smaller then $\mathcal{G}_0$, thus they have proof. Contradiction.

b/ Suppose there is no negative occurrence of modality in $\mathcal{M}_0$.

1. Let there be at least one formula of the form $\neg \alpha \rfloor q$, $\neg \alpha \lceil q$ or $\neg \alpha \perp q$ in $\mathfrak{T}_0$. Surely none of the rules $\perp 1, \perp 2, \perp 3$ is appliable to any negative occurrence of a modality in $\mathfrak{T}_0$. Let us considere a modal formula from $\mathcal{M}_0$, e.g. $\alpha \perp q \in \mathcal{M}_0$. The theory $\mathfrak{T}_0 \cup \{\alpha \perp q\}$ cannot correspond to a valid sequent, because this sequent is $\prec$ smaller then $\mathcal{G}_0$ and it would have to have a proof. The structure $\langle \mathcal{W}, \mathcal{P}, \mathcal{C} \rangle$ for $\mathfrak{T}_0 \cup \{\alpha \perp q\}$ can be reconstructed into a structure for $\mathfrak{T}_0$ if no $\perp 1, \perp 2, \perp 3$ is applicable. This is done as follows :
Let ν be a sequence from $\mathcal{C}$ beginning in the base state. If $\neg(\forall \nu_i (\nu_i \neq \Lambda \supset \nu_i \vDash \neg q))$, this sequence is carried over

to the new structure and new sequence of $\mathcal{C}$ is considered. Otherwise we shall find the negative modality, e.g. $\neg a \lceil r \in \mathcal{T}_0$, which is illustrated by ν. If there is none we procced to a new sequence of $\mathcal{C}$.
As stated above $\perp 3$ is not appliable to $\mathcal{G}_0$ thus there is a sequence of structures contradicting the use of $\perp 3$. This sequence - ending in Λ - is concatenated to the base state of the original structure to form a new sequence which should replace ν in $\mathcal{C}$. This is done to all negative modalities illustrated by ν. This procedure is repeated for all sequences from $\mathcal{C}$. The structure thus obtained with the same base state as the original structure realizes all the formulas from $\mathcal{G}_0$. Contradiction.

2. If there is no negative modal formula in $\mathcal{G}_0$. Let us use the rule $\perp 0$ and let us consider the sequent $\mathcal{G}_0'$ with the theory $\mathcal{T}_0' = \mathcal{T}_0 \cup \{\neg a \perp \underline{\text{false}}\}$. Let there be a modal formula, e.g. $a \perp q \in \mathcal{M}_0$. If $\mathcal{T}_0' \setminus \{a \perp q\}$ has no model, then $\mathcal{T}_0 \setminus \{a \perp q\}$ has no model, too. Theory $\mathcal{T}_0 \setminus \{a \perp q\}$ corresponds to a sequent $\prec$ smaller then $\mathcal{G}_0$, thus it has a proof. Contradiction.
If $\mathcal{T}_0' \setminus \{a \perp q\}$ has a model, and if $\perp 1$ is not appliable to the sequent $\mathcal{G}_0'$ corresponding to $\mathcal{T}_0'$, the structure for $\mathcal{T}_0'$ can be constructed as in the case 1. Thus $\perp 1$ must be appliable to $\mathcal{G}_0'$ and the proof of $\mathcal{G}_0$ can be constructed. Contradiction.

Theorem 2 :

Each valid sequent of process logic has a proof using the rules P1 - P4, $\lceil 1$ and $\perp 0 - 3$.

The completeness of the dedicion procedure for process logic using the above rules is stated in the following theorem, which is an easy consequence of the Lemma 1, Lemma 2.

References

[1] V.R. Pratt : Process Logic. Preliminary Report. Conference Record of the 6th Annual ACM Symp. on Principles of Progr. Lang., San Antonio, TX, Jan. 29. - 31., 1979

[2] V.R. Pratt : Six Lectures on Dynamic Logic, Research Report MIT/LCS/TM - 117

Holms Gårds Tryckeri