SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 20(10), 985–1006 (OCTOBER 1990)

The Implementation of ProTest: a Prolog Debugger for a Refined Box Model

A. SCHLEIERMACHER AND J. F. H. WINKLER

Siemens AG, Corporate Research and Technology, Otto-Hahn-Ring 6, D-8000 Munich 83, Federal Republic of Germany

SUMMARY

We describe some aspects of the implementation of a Prolog debugger for a refined box model in which attempted unifications can also be observed. Our implementation of the ProTest debugger is based on a meta-interpreter for Prolog. We start with an existing meta-interpreter for Byrd's box model (four-port debugger) and we transform it into one for the refined box model (ten-port debugger). To explain the transformation we show several versions of the meta-interpreter. In these versions we use the technique of changing the database to implement the cut, but another possibility is also explained briefly. A simple notation for typing is used to make Prolog programs more readable. In an appendix we give a listing of a simple prototype of the extended meta-interpreter.

KEY WORDS Prolog debugger Refined box model Meta-interpreter Four-port debugger Ten-port debugger Cut implementation Typing in Prolog

INTRODUCTION

The purpose of this paper is to describe some aspects of the implementation of the ProTest debugger for Prolog using a refined box model instead of the usual one as defined by Byrd.[1,2] In the usual box model there is one box for each subgoal called in the body of a clause and every box has four ports, CALL, EXIT, REDO and FAIL, through which it may be entered or left. We may call such a box an AND box since the subgoals within a clause body are connected by sequential* *and*.

Now the different clauses of a predicate definition are connected by sequential* *or*, and the proposed refinement of the box model consists of placing OR boxes inside an AND box: one OR box is created for each clause so that the OR boxes correspond one-to-one to the clauses of the predicate. The OR boxes have six ports, TRYMATCH, FAILMATCH, ENTERBODY, EXITBODY, REDOBODY and FAILBODY, corresponding to fairly obvious states in the execution. ENTERBODY is an inner door separating a compartment for unification from a second compartment of the OR box reserved for execution of the body. This second compartment in turn will contain further AND

* Owing to the well-known depth-first left–right execution strategy of Prolog the connectives and and or in Prolog are not exactly the same as the corresponding logical connectives.

0038–0644/90/100985–22$11.00

Received 12 October 1988
Revised 26 September 1989
and 25 February 1990

boxes for the subgoals of the clause, and so on. In the next section we given some more details. For a detailed discussion of the semantics of the refined box model see Reference 3. Extended box models have also been used in References 4–8.

Our implementation is based on the concept of a meta-interpreter, i.e. an interpreter of Prolog written in Prolog. We did not implement the meta-interpreter from scratch but we started with an existing meta-interpreter which was working with Byrd's box model and transformed it to implement the refined box model. As this transformation turned out to be very simple, it seems worth while to describe it here.

It was one of the experiences gained during the implementation that understanding a Prolog program written by someone else could be helped if type information were available (compare also Reference 9, p. 41). Perhaps this is especially true in maintenance or similar work. In such situations it is not enough merely to glance over a program and get an intuitive understanding of what relations the various predicates are supposed to represent. In most cases when one has to make changes to a predicate definition precise knowledge is required as to the intended structure of the arguments that will be accepted by the predicate.

We shall therefore present informally a typing concept and show predicate definitions together with the type information required.

To explain the transformation of the four-port debugger according to Byrd's box model into our ten-port debugger we shall present five versions of a meta-interpreter. The most basic interpreter is shown in Figure 2. It does not implement the cut and does not have the appropriate structure suited for a debugger. The next stage in Figure 3 has the appropriate structure for the four-port debugger. It implements the cut using the technique of changing the database dynamically but it has only the minimal data structures required for its own control. Figure 4 shows the same program structure as Figure 3 but more appropriate data structures for the purpose of debugging have been added. The changes and extensions required to obtain our ten-port debugger are shown in Figure 5. Finally we present a listing of a simple prototype of the extended meta-interpreter in Figure 6.

THE REFINED BOX MODEL

Figure 1 shows the refined box model. In the refined box model there is an AND box for each goal. It has four ports, CALL, EXIT, REDO and FAIL. Inside the AND box there are OR boxes for each clause of the current predicate. Each OR box has six ports, TRYMATCH, FAILMATCH, ENTERBODY, EXITBODY, REDOBODY and FAILBODY.

The OR boxes have a compartment for unification of the clause head and a compartment for execution of the clause body. When unification succeeds, the compartment for execution of the body is entered via the ENTERBODY port. It contains in turn the AND boxes for the goals called in the clause body. When unification fails the first compartment of the OR box is left via the FAILMATCH port. In the next step the OR box of the next clause (if there is one) is entered via the TRYMATCH port. If there are no further OR boxes the surrounding AND box is left via the FAIL port. That is to say, indexing is excluded in this model. In fact, we think that it is justly excluded, because it could mask errors in clause heads (see Reference 3, p. 3, for an example). This ten-port box model can also be used to explain the semantics of Prolog.

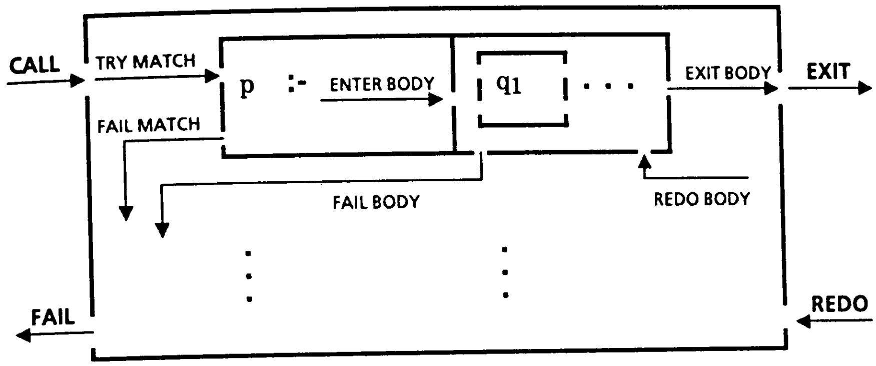

Figure 1. The refined box model

A box model with ten ports similar to the one used here was first proposed in Reference 5. In Reference 4 an execution model with eight ports is discussed and used to describe the semantics of Prolog. Plummer's CODA system uses a model which is also very similar to the one that we have adopted but he omits so to speak the second compartment of the OR boxes reserved for clause bodies. Consequently the EXITBODY, REDOBODY and FAILBODY ports are not shown in his system.

The main advantage of the refined box model is that it allows a more complete observation of the program. Bugs in clause heads are often hard to discover with a debugger using the simpler model since the debugger never stops at any point 'near the bug' (Reference 3, p. 3). This property of making the process of tried and failed unifications observable is also shared by Plummer's model with seven ports. We have preferred the more complete model with a compartment of the OR box provided for the clause body since it allows a complete representation of the AND–OR tree in terms of boxes and of the inclusion relation between them. The disadvantage that may lie in the greater number of ports and a certain redundancy in some situations is overcome by the standard technique of allowing the user to switch ports on or off as he wishes.

Furthermore, ProTest offers other well-known techniques for debugging, such as spy points, controlled jumps between ports of one box (and to the CALL port of the parent box), and leap mode together with zooming. (More details are given in Reference 3). Usually a programmer will apply several of these techniques simultaneously. In such situations the ten-port model offers better possibilities because of its completeness. At the EXITBODY port a user might, for example, realize that the clause body should have failed. With a controlled jump he can now force the exit of the clause at the FAILBODY port. At the FAILBODY port a user might realize that the clause body should have succeeded. In this case he can interrupt the debugging, modify the program and jump back to the ENTERBODY port. In a 'non-redundant' model the FAILBODY port would be missing and the equivalent action could not be performed in such a clear way.

To sum up this discussion, we think that one should not underestimate the advantage lying in the fact that the ten-port model corresponds more closely to the source language. In Prolog we have clauses consisting conceptually of a clause head and a clause body, so we should have OR boxes having a compartment corresponding to the clause head and one corresponding to the clause body. Thus, inside the compartment of the body we then have the AND boxes of the goals of the body, and the nesting of the boxes reflects directly the structure of the AND–OR tree.

META-INTERPRETERS FOR PROLOG

Basically, a meta-interpreter is a very simple recursive program. It tests the goal on which it is applied to see whether it is composed from simple goals by means of the and operator. If so, it decomposes the goal and applies itself recursively to the two resulting components. If the goal is simple the meta-interpreter requires a test whether the goal can be matched by a predefined predicate. In this case the goal is just called. Otherwise the predefined predicate clause/2 is called to find the body of the current clause unifying with the goal. Thus a meta-interpreter might consist of three lines[10] (see Figure 2).

```
interpret((G1, G2)) :-  !, interpret(G1), interpret(G2).
interpret(GOAL)     :-  system_goal(GOAL), !, GOAL.
interpret(GOAL)     :-  clause(GOAL, BODY), interpret(BODY).
```

Figure 2. A simple meta-interpreter

This meta-interpreter does not implement the cut-operator properly. Also it is not easy to see where to insert predicates to provide for stopping at test points. Therefore it seems useful to distinguish between interpreting a simple goal and a chain of goals linked by sequential and. If we do this we arrive at the structure in Figure 3.

Here we have a structure in which it is obvious where to deal with the breaks that arise when Byrd's box model is adopted. The relevant predicates look roughly as follows:

```
debug_break_before(GOAL)  :-
     ( break(GOAL, 'CALL')
     ; break(GOAL, 'FAIL'),
         GOAL \== !,
         retract($cut_executed),
         fail).

debug_break_after(GOAL)  :-
     ( break(GOAL, 'EXIT')
     ; break(GOAL, 'REDO'),
       fail).
```

Obviously, the information available in these predicates is not yet sufficient at test points. With the predicates in their present shape we could only show the current goal and the port. A data structure representing the actual state of the execution tree is required as an argument in these predicates if we wish to be able to display more detailed information.

THE CUT IMPLEMENTATION

The technique of changing the database by adding a fact of the form $cut_executed is used to implement the cut. Whenever a cut is met on backtracking the subgoal is !

```
interpret_simplegoal(GOAL) :-
    clause(GOAL, BODY),
    (  true
    ;  clause($cut_executed, true),  !,  fail),
    interpret_body(BODY).

interpret_body((G1, MOREGOALS)) :-   !,
    interpret_subgoal(G1),
    interpret_body(MOREGOALS).
interpret_body(GOAL)  :-
    interpret_subgoal(GOAL).

%  the first clause of interpret_subgoal deals with cases such as :
%  g1, (g2, g3), g4,  ...   .

interpret_subgoal((G1, MOREGOALS))  :-   !,
    interpret_body((G1, MOREGOALS)).

interpret_subgoal(GOAL)  :-
    test_goal(GOAL, NEW_GOAL),

    %  breaks at  CALL and FAIL ports  may be dealt with here:
    debug_break_before(GOAL),
    NEW_GOAL,
    %  breaks at  EXIT and  REDO ports  may be dealt with here:
    debug_break_after(GOAL),
    (  true
    ;  not_backtrackable(NEW_GOAL),
       !,  fail).

test_goal(GOAL, exec_cut)  :-
    GOAL == !,   !.
test_goal(GOAL, exec_goal(GOAL))  :-
    system_goal(GOAL),  !.
test_goal(GOAL, interpret_simplegoal(GOAL)).

exec_cut.
exec_cut   :-  asserta($cut_executed),  fail.

exec_goal(GOAL)  :-    GOAL.
```

Figure 3. Basic structure of the meta-interpreter for the four-port debugger

and therefore NEW_GOAL is exec_cut. The effect of backtracking on exec_cut is that a fact $cut_executed will be added to the database. Backtracking to the left of the cut operator will now be prevented by means of the call of not_backtrackable(NEW_GOAL) in the second alternative of the last subgoal in interpret_subgoal/1. The added fact is removed when the parent box is left finally through the FAIL port. The goal clause($cut_executed, true) is equivalent to simply calling $cut_executed when a fact of this form is present in the database, but when no fact is present most Prolog systems would react with an exception to the call of $cut_executed. Thus clause/2 is used to prevent this exception:

```
not_backtrackable(_)  :-
    clause($cut_executed, true),  !.

not_backtrackable(exec_goal(GOAL))  :-
    not_backtrackable_system_goal(GOAL).
```

The second clause in not_backtrackable/1 serves to suppress useless attempts to get more solutions on calls to predefined predicates which cannot be resatisfied.

This solution may be considered not quite satisfactory in that a debugger should not alter the database. However, if a module concept is available in the particular Prolog system at hand and if we can make sure that the added clauses are visible only to the debugger, such a solution works in the correct manner. If a module concept with the required properties is not available, one has to make suitable naming conventions to ensure that the asserted or retracted facts do not collide with any predicate definitions of the user's program.

The meta-interpreter from which we have started uses a different technique which consists of providing the possibility to cut all choice points up to a given one. This is implemented by means of two internal predicates which cannot be written in Prolog since they must have access to the choicepoint entries on the local Prolog stack:

```
read_choicept(CHOICEPT)        % unifies CHOICEPT with a pointer to the
                               % last choice point entry on the stack.

cut_choicept(CHOICEPT)         % effects a cut up to the choice point
                               % pointed to by CHOICEPT.
```

It should be evident how this can be used to implement the cut. When a cut is met on backtracking, all choice points must be cut up to the choice point at the beginning of the interpretation of the parent goal of the cut. Then a fail will result in going back to the FAIL port of the box associated with the parent goal. For more details see the remark at the end of the section on the meta-interpreter for Byrd's box model. Similar predicates are described by Uchida.[11] In ProTest we have used these internal predicates to implement the cut. But since they are not generally available to the reader, in this paper we replace them by the database technique which allows us to keep the structure of the meta-interpreter almost unchanged. Further techniques for the implementation of the cut are reported in References 12 and 13.

THE DATA STRUCTURES OF THE META-INTERPRETER AND TYPING

To describe Prolog data structures we adopt a simple notation which we hope is almost self-explanatory. We write type definitions by means of type and subtype clauses which syntactically resemble Prolog clauses. The type identifiers thus defined will be used later in the program definition. We shall place them into the clause heads after each first occurrence of a variable separated from the variable by ':'. To distinguish type identifiers from ordinary literals we enclose the type identifiers in angular brackets, e.g. ⟨and_box⟩. There are two special type identifiers, namely ⟨_⟩ for the universal type and ⟨variable⟩ for an uninstantiated variable.

The syntax of a type clause is

```
type_clause---> type(type_identifier) :- typed_term
```

(---> and | are meta-symbols, terminal symbols are printed in bold face).

The syntax of typed_term resembles that of ordinary Prolog terms except that wherever a variable may occur in an ordinary Prolog term it is to be replaced by typed_variable, which has the syntax

```
typed_variable --->       variable : type_indicator
                          | type_indicator

type_indicator --->       type_identifier
                          | typed_term
                          | type_union

type_union --->           type_indicators
                          | (type_indicators)

type_indicators --->      type_indicator ;; type_indicator
                          | type_indicator ;; type_indicators
```

The syntax of subtype clauses is somewhat simpler:

```
subtype_clause --->
     subtype(subtype_identifier of type_identifier)  :– instantiations.
subtype_identifier --->          type_identifier
instantiations      --->         variable   : type_indicator
                                 | variable : type_indicator, instantiations
```

In the meta-interpreter we need the following type and subtype definitions written in the notation just introduced:

```
type(⟨and_box⟩)   :– and_box (GOAL              : ⟨legal_goal⟩,
                              CONTEXT            : ⟨context⟩,
                              PREVIOUS_BOX       : (⟨and_box⟩ ;; none),
                              LAST_CHILD_BOX     : ⟨_⟩).

type(⟨context⟩)   :– context  (CURRENT_CLAUSE    : (⟨clause⟩ ;; ⟨query⟩),
                              PARENT_BOX         : (⟨and_box⟩ ;; none)).
type(⟨legal_goal⟩)  :– ⟨atom⟩ ;; ⟨structure⟩.
type(⟨clause⟩)      :– HEAD : ⟨legal_goal⟩  :– BODY : ⟨body⟩.
type(⟨query⟩)       :– ?– ⟨body⟩.
type(⟨body⟩)        :– ⟨subgoal⟩.
type(⟨body⟩)        :– ⟨subgoal⟩, ⟨body⟩.
type(⟨subgoal⟩)     :- ⟨atom⟩ ;; ⟨structure⟩ ;; ⟨variable⟩.
type(⟨subgoal⟩)     :- (⟨body⟩).
subtype(⟨and_box_with_child_boxes⟩ of ⟨and_box⟩)  :–
          LAST_CHILD_BOX : ⟨and_box⟩.
```

We use the special symbol ';;' to indicate alternatives in type indicators. Thus, for example, the value of PREVIOUS_BOX can be either of type ⟨and_box⟩ or it can be the literal none which indicates that no previous box is present. Another way of expressing alternatives is of course to write several clauses for the same type definition.

Let us now turn to the description of how the types are to be used in procedure definitions. In principle two type definitions are needed for each variable occurring in an argument of a Prolog clause: the first one to describe the assumptions made on the call and the second to describe the result. We reflect this in our notation by writing, for instance,

X : ⟨initial_type⟩ => ⟨final_type⟩.

There are two special cases which deserve extra consideration. First, if X is a pure input variable ⟨final_type⟩ is not really needed and writing X:⟨initial_type⟩=⟩ ⟨initial_type⟩ would be unnecessarily long. Besides, that approach is not precise enough. We shall write X:⟨initial_type⟩ instead in this special case. This indicates that in the call the variable should have a value of type ⟨initial_type⟩ and that no further instantiations take place, i.e. the variable is read-only. The second situation which needs special consideration arises when ⟨initial_type⟩ and ⟨final_type⟩ are the same but further instantiations may take place. We shall write X:⟨initial_type⟩=⟩ to express this situation, which is a shorthand denotation for X:⟨initial_type⟩=⟩⟨initial_type⟩.

It should be obvious from the notation itself and from the above remarks that our type declarations are to be considered as compatible with any further bindings which are possible through unification and that the arrow between the two type names reflects such further bindings. It will also be seen that our notation is flexible enough to express the usual modes 'pure input' and 'pure output', i.e. our notation also covers some of the mode annotations used in other approaches.[14–18] The pure input mode has already been mentioned and is denoted by

X : ⟨initial_type⟩.

For a pure output variable there is no restriction on its form in the call. Hence the output mode is denoted by X:⟨_⟩=⟩⟨final_type⟩. But there is an alternative and more stringent definition if we require the initial type to be ⟨variable⟩. Thus X:⟨variable⟩=⟩ ⟨final_type⟩ is a pure output variable in this stricter sense.

A different approach would consist of combining a type indicator with a mode tag. This has been chosen, for instance, in the language Trilogy.[19] However, it seems to us that at least for the present largely explanatory purposes our approach of showing an initial type and the precise degree of unification to take place is more suitable.

In the most general case when a variable is specified as X:⟨initial_type⟩=⟩⟨final_type⟩ it may often be desirable to declare ⟨final_type⟩ as a subtype of ⟨initial_type⟩. Subtype declarations help to clarify hierarchical relations between types, and they also help to save space.

There is a certain number of basic types such as ⟨integer⟩, ⟨atom⟩, ⟨structure⟩, ⟨variable⟩ etc., which will not be explained further. In these cases there is always a well-defined Prolog notion corresponding to the type and even a predefined predicate for the corresponding type check.

In the case of non-basic types, predicates for type checks at run-time can be derived automatically from the corresponding type clauses. Initial and final checks may then be inserted optionally at the beginning and at the end of each clause. The predicate for checking the type ⟨context⟩ would, for instance, be

```
type(⟨context⟩, CXT)  :-
    functor(CXT, F, A), F == context, A == 2,
    arg(1, CXT, CURRENT_CLAUSE),
    (  type(⟨clause⟩, CURRENT_CLAUSE)
    ;  type(⟨query⟩, CURRENT_CLAUSE) ),
    arg(2, CXT, PARENT_BOX),
    (  type(⟨and_box⟩, PARENT_BOX)
    ;  PARENT_BOX == none).
```

The final check for a pure input variable X:⟨initial_type⟩ is somewhat more complicated, since we would have to save the initial value of X somehow, for instance in the database. An elegant solution would consist of putting marks into tag fields for all the variables inside the term X which must not be further instantiated and take this into account in the unification algorithm.

It should be noted that our notation cannot be regarded as some kind of discipline to be enforced on programmers. One may always give the type of a variable X as X:⟨_⟩=⟩, and we make the convention that this may be omitted.

It is often argued that in the presence of good tools for type inference[20–25] explicit typing is not necessary in Prolog. But although type inference methods may become more and more perfected there are still good reasons for providing explicit type information in Prolog programs.

From a human engineering point of view, explicit type declarations have the advantage that they can be tailored by the programmer, for instance with respect to the degree of detail provided or the choice of (suggestive) names. Names created automatically are usually not very suggestive.

Besides that, automatically-derived types may reflect program bugs. They can therefore not be used for type checks or in correctness proofs, nor can they be relied upon in debugging situations. If on the other hand, type information is provided explicitly by the programmer it reflects the programmer's proper intentions and this may be checked against the actual program text.[26,27]

It is therefore our view that the existence of type inference algorithms should not be used as an argument against explicit typing. Rather, tools for type inference should be considered as a means to check explicit type information against the remaining information contained in the program. Zobel observes that explicit type declarations can improve the effectiveness of algorithms for type derivation.[28]

The type notation described above is similar to others already described in the literature.[15,29–31] The unique feature of our approach seems to be the distinction of initial and final type.

THE META-INTERPRETER FOR BYRD'S BOX MODEL

We are now in a position to show the essential ideas of the program which was the starting-point of our implementation (see Figure 4). The predicate interpret_simplegoal-

```
interpret_simplegoal(BOX :  <and_box>  =>
                            <and_box_with_child_boxes>)   :-
      get_arg(1, BOX, GOAL),
      clause(GOAL, BODY),
      (  true
      ;  clause($cut_executed, true),   !,    fail),
      CXT  =   context(GOAL :- BODY, BOX),
      interpret_body(BODY, CXT, none, LAST_CHILD),
      set_arg(4, BOX, LAST_CHILD).

interpret_body((G1 : <legal_goal> =>, MOREGOALS : <body>  =>),
                CXT : <context>  =>,
                PREVIOUS_BOX : (<and_box> => ;;  none),
                LAST_BOX : <_> => <and_box>)     :-
      !,
      interpret_subgoal(G1, CXT, PREVIOUS_BOX, NEW_BOX),
      interpret_body(MOREGOALS, CXT, NEW_BOX, LAST_BOX).

interpret_body(GOAL : <legal_goal> =>, CXT : <context>  =>,
                PREVIOUS_BOX : (<and_box> => ;;  none ),
                LAST_BOX : <_> => <and_box>)     :-
      interpret_subgoal(GOAL, CXT, PREVIOUS_BOX, LAST_BOX).

interpret_subgoal((G1 : <legal_goal> =>, MOREGOALS : <body> =>),
                CXT : <context>  =>,
                PREVIOUS_BOX : (<and_box> =>  ;;  none ),
                LAST_BOX : <_> => <and_box>)     :-
      !,
      interpret_body((G1, MOREGOALS), CXT, PREVIOUS_BOX, LAST_BOX).

interpret_subgoal(GOAL : <legal_goal> =>, CXT : <context> =>,
                PREVIOUS_BOX : (<and_box> => ;;  none),
                NEW_BOX : <_> => <and_box>)     :-
      NEW_BOX = and_box(GOAL, CXT, PREVIOUS_BOX, _),
      test_goal(NEW_BOX, NEW_GOAL),
      debug_break_before(NEW_BOX),
      NEW_GOAL,
      debug_break_after(NEW_BOX),
      (  true
      ;  not_backtrackable(NEW_GOAL),
         !,  fail).
not_backtrackable(_)      :-
      clause($cut_executed, true),  !.
not_backtrackable(exec_goal(BOX : <and_box>))    :-
      get_arg(1, BOX, GOAL),
      not_backtrackable_system_goal(GOAL).
```

Figure 4. Continued

```
test_goal(BOX : <and_box>, exec_cut)    :-
      get_arg(1, BOX, GOAL),
      GOAL == !,  !.
test_goal(BOX : <and_box>, exec_goal(BOX))   :-
      get_arg(1, BOX, GOAL),
      system_goal(GOAL),  !.
test_goal(BOX : <and_box>, interpret_simplegoal(BOX)).

exec_cut.
exec_cut :-  asserta($cut_executed), fail.

exec_goal(BOX : <and_box> =>)   :-
      get_arg(1, BOX, GOAL),
      GOAL.
```

Figure 4. The four-port debugger

(BOX) is defined on any argument of type ⟨and_box⟩. It can only succeed if there is a predicate definition in the database matching GOAL, the goal contained as the first argument in BOX. It does succeed if GOAL succeeds, and in that case it produces the corresponding answer substitution in GOAL. On backtracking interpret_simplegoal/1 succeeds as long as GOAL would continue producing further solutions. Instantiations may occur also in the other arguments of BOX. In particular, the last argument is bound to the AND box of the last subgoal of the current clause that produced the solution to GOAL.

Note that interpret_simplegoal(BOX) is never called when GOAL is ! or when GOAL is matched by another predefined predicate. It is the purpose of the predicate test_goal/2 to ensure this.

The function of the predicate interpret_body(BODY, CXT, PREVIOUS_BOX, LAST_BOX) is to interpret the composed goal given by BODY. It succeeds if, and on backtrack as long as, BODY itself would succeed. The assumptions made on the form of BODY are described by the data type ⟨body⟩. We have deliberately simplified the situation by not admitting alternatives (A; B) within clause bodies. To implement ';' one would need another clause in test_goal and a corresponding predicate interpret_alternative to be called in this case.

Another simplification consists of the assumption that no illegal goals may be called either directly or indirectly. Although uninstantiated variables are allowed as subgoals in the data structure ⟨body⟩ they must be properly instantiated when it is their turn to be interpreted. Thus a variable that remains uninstantiated would lead to an infinite loop owing to the fact that the variable is split up by unification into (G1, MOREGOALS) in the first clause of interpret_subgoal, etc.

The predicate interpret_subgoal(GOAL, CXT, PREVIOUS_BOX, NEW_BOX) effects the interpretation of the current subgoal and as a side-effect it causes the breaks at the four ports of Byrd's box model. The first clause deals with bracketed subgoals of the form (G1, G2) which must be decomposed into simple goals by calling interpret_body (G1, G2), ...). The second clause deals with a simple goal GOAL. A new box NEW_BOX is formed, and a new goal NEW_GOAL which may either be exec_cut when GOAL == ! or exec_goal(NEW_BOX) when GOAL is matched by a predefined predicate or else finally interpret_simplegoal(NEW_BOX). The breaks are treated immediately before and after the call to NEW_GOAL. The goal interpret_subgoal(GOAL, . . .) succeeds whenever GOAL would succeed, because NEW_GOAL has that property.

The predicate break/2 will now have a box passed as its first parameter. Therefore all information available in the current execution tree can be shown either by default or on the user's request. We shall not go into details here about the predicate break/2. But it should be obvious that it involves the dialogue interface with the user. (The user interface of ProTest is based on windows and menus.[3]):

```
debug_break_before(BOX : ⟨and_box⟩)      :-
    ( break(BOX, 'CALL')
    ; break(BOX, 'FAIL'),
      get_arg(1, BOX, GOAL),
      GOAL \== !,
      retract($cut_executed),
      fail).

debug_break_after(BOX : ⟨and_box⟩)      :-
    ( break(BOX, 'EXIT')
    ; break(BOX, 'REDO'),
      fail).
```

Only one of the components in the data structure ⟨and_box⟩ is needed for controlling the meta-interpreter itself, namely the GOAL component defining the current goal. The remaining components provide for all the information that one might wish to display at breakpoints. The component named CURRENT_CLAUSE in the substructure CONTEXT may be used to show the current clause. The PREVIOUS_BOX link may be used to identify the current goal within the current clause. The upward link may be used to establish the calling history. But as we have linked the data structure ⟨and_box⟩ upwards and downwards, the complete actual state of the execution tree may be shown at each breakpoint.

Remark

We wish to indicate briefly how to use the internal predicates cut_choicept/1 and read_choicept/1 to implement the cut. At the beginning of the clause body of interpret_simplegoal/1 we insert the subgoal read_choicept(CUTCPT). In the data type ⟨context⟩ we need an additional argument CUTCPT to store the value determined by read_choicept(CUTCPT). All the statements changing and restoring the database and inspecting whether a fact $cut_executed is present become obsolete. Thus the first clause of not_backtrackable/1 is no longer needed. The body of the second clause of exec_cut/0 consists only of fail and interpret_simplegoal/1 the alternative subgoal (true ; clause($cut_executed, true), !, fail) is no longer needed. In the second part of the alternative subgoal of debug_break_before if GOAL == ! the value of CUTCPT is extracted from BOX and used as argument in a call to cut_choicept(CUTCPT). The subsequent fail then causes the parent goal to fail.

The internal predicates cut_choicept/1 and read_choicept/1 may also be used for other purposes apart from the cut implementation which we have not shown here. One might for instance wish to go back to the CALL port of the current box or of the parent box or the previous box. Or one might wish to force a jump to the EXIT or FAIL port. In ProTest such jumps are implemented by a cut up to an appropriately-chosen choice point and subsequent fail.

THE META-INTERPRETER FOR THE REFINED BOX MODEL

We now describe the changes that are required to obtain a meta-interpreter which serves our refined box model. (For a complete listing see the Appendix).

The changes are remarkably simple. In fact most of the meta-interpreter can be left unchanged and the changes are concentrated in interpret_simplegoal/2. Before a user-defined predicate is called we collect all the clauses of the predicate that are currently in the database and put them into a list. This list will be part of the data structure ⟨and_box⟩. The meta-interpreter then uses this list exclusively to interpret the goal. That is to say, changes to the database affecting the called predicate will only become effective when the particular call is no longer active.

The final form of the data structure ⟨and_box⟩ is therefore:

```
type(⟨and_box⟩) :-
    and_box( SIMPLE_GOAL         : ⟨legal_goal⟩,
             CONTEXT             : ⟨context⟩,
             PREVIOUS_BOX        : (⟨and_box⟩ ;; none),
             LAST_CHILD_BOX      : ⟨_⟩,
             PROCEDURE           : ⟨_⟩ ).
type(⟨procedure⟩) :-
    procedure(CLAUSE_INDEX : ⟨_⟩, CLAUSE_LIST : ⟨clause_list⟩).

type(⟨clause_list⟩) :-
    [ ] ;; [CLAUSE : ⟨clause⟩ | CLAUSE_LIST : ⟨clause_list⟩].

subtype(⟨and_box_with_procedure_definition⟩ of ⟨and_box⟩) :-
    PROCEDURE : ⟨procedure⟩.

subtype(⟨procedure_with_selected_clause⟩ of ⟨procedure⟩) :-
    CLAUSE_INDEX : ⟨integer⟩.

subtype(⟨and_box_with_selected_clause⟩ of
        ⟨and_box_with_procedure_definition⟩) :-
    PROCEDURE : ⟨procedure_with_selected_clause⟩.

subtype(⟨and_box_with_child_boxes⟩ of ⟨ and_box_with_selected_clause⟩) :-
    LAST_CHILD_BOX : ⟨and_box⟩.

type(⟨context⟩) :-
    context(CURRENT_CLAUSE : (⟨clause⟩ ;; ⟨ query⟩),
        % the current clause could now also be accessed via the
        % PROCEDURE entry in the parent box. But we would have to
        % process the list of clauses to find its N-th entry.
        % The alternative ⟨query⟩ is needed on the top level.
        % (compare debug_goal/1 in the listing in the Appendix).
          PARENT_BOX : (⟨and_box_with_selected_clause⟩ ;; none) ).
```

Of course the first goal in the second clause of intepret_subgoal/3 has to be adapted to the new form of the data structure ⟨and_box⟩:

```
NEW_BOX = and_box(GOAL, CXT, PREVIOUS_BOX, _,_),
```

Collecting all the clauses is achieved by means of the predicate:

```
collect_clauses(BOX : ⟨and_box⟩ => ⟨and_box_with_predicate_definition⟩)  :-
    get_arg(1, BOX, GOAL),
    functor(GOAL, F, Arity),
    functor(T, F, Arity),
    findall(T :- Body, clause(T, Body), CLAUSE_LIST),
    set_arg(5, BOX, procedure(_, CLAUSE_LIST) ).
```

The call to collect_clauses/1 has to be placed into the last clause of test_goal/2. The other clauses of test_goal/2 remain unchanged.

```
test_goal (BOX : ⟨and_box⟩ =⟩ ⟨and_box_with_procedure_definition⟩,
           interpret_simplegoal(BOX))   :-
    collect_clauses(BOX).
```

All the other changes required are in interpret_simplegoal/1 whose final form is shown in Figure 5.

In next_in_list/3 the backtrack behaviour of append/3 is used to produce successively all elements of CLAUSE_LIST. Therefore when backtracking reaches the predicate next_matching_clause/2 we get the next clause from CLAUSE_LIST. The meta-interpreter stops at the TRYMATCH port to show the clause. Then the head is tested whether it matches the actual goal. If it matches the ENTERBODY port will be entered. If it does not match the break at the FAILMATCH port will be treated and backtracking in the predicate next_matching_clause will be repeated. This may go on until CLAUSE_LIST is exhausted.

It may be noted that OR boxes are not present explicitly in the data structure defined for the meta-interpreter. They exist only implicitly in the substructure PROCEDURE which could be viewed as the collection of all possible OR boxes with one actually chosen by CLAUSE_INDEX. The decision to collect all clauses (i.e. the procedure definition) before stopping at the first breakpoint does not have as its primary objective control of the meta-interpreter. The meta-interpreter could be controlled equally well by the backtrack behaviour of the predefined predicate clause/2. The decision was rather made with the view that the procedure definition should be shown to the user at each breakpoint. ProTest uses two windows to display the state of the program execution. In one window it shows the current clause, i.e. that clause whose body is being executed. Within the body the current subgoal is highlighted. In another window ProTest shows the clause heads of the procedure definition belonging to the current subgoal. In this window the clause head to be tried next is highlighted.[3] Collecting all the clauses in advance also accords with the static (logical) view of the predicates modifying the database which has been favoured by the various standardization bodies involved in the standardization of Prolog.[32]

```
interpret_simplegoal(BOX : <and_box_with_procedure_definition> =>
                           <and_box_with_child_boxes>)    :-
    get_arg(1, BOX, GOAL),
    next_matching_clause(BOX, BODY),
    (   true
    ;   clause($cut_executed, true), !, fail),   % Stop backtrack after cut.
    (   break(BOX, 'ENTERBODY')
    ;   break(BOX, 'FAILBODY'),
        fail),
    CXT = context(GOAL :- BODY, BOX),
    interpret_body(BODY, CXT, none, LAST_CHILD),
    set_arg(4, BOX, LAST_CHILD),
    (   break(BOX, 'EXITBODY')
    ;   break(BOX, 'REDOBODY'),
        fail).

next_matching_clause(BOX : <and_box_with_procedure_definition> =>
                           <and_box_with_selected_clause>,
                     BODY : <_> => <body>)     :-
    get_arg(5, BOX, procedure(N, CLAUSE_LIST)),
    next_in_list(CLAUSE_LIST, N, HEAD :- BODY),
    test_clause(BOX, HEAD :- BODY).

next_in_list(CLAUSE_LIST : <clause_list>,
             N : <_> => <integer>,
             HEAD : <_> => <legal_goal> :- BODY: <_> => <body>)   :-
    append(BEGINNING,  HEAD :- BODY | TAIL], CLAUSE_LIST),
    length(BEGINNING, L),
    N is L+1.

test_clause(BOX : <and_box_with_selected_clause> =>,
            HEAD : <legal_goal> => :- BODY : <body> => )   :-
    get_arg(1, BOX, GOAL),
    ( break(BOX, 'TRYMATCH'),
      HEAD = GOAL, !
    ; break(BOX, 'FAILMATCH'),
      fail).
```

Figure 5. The changes required to obtain a ten-port debugger

This describes on a certain level of abstraction the meta-interpreter required for the refined box model and the changes required to transform a meta-interpreter for Byrd's box model into one for the refined box model. It has been our aim to show that this transformation is very simple.

Of course, the meta-interpreter is only one out of several components of the debugger. Other important components are a command interpreter and I/O modules working on the basis of a window system.

CONCLUSIONS

At the present time most of the debuggers that are available for Prolog still use Byrd's box model which does not allow the user to follow the details of attempts at unification. We have shown here a simple way to implement a more complete box model in which attempted and failed unifications are also shown. In this model the nested structure of boxes corresponds closely to the AND–OR tree.

In our presentations of Prolog programs or pieces of programs a simple notation for type definitions is employed. The type information significantly improves the readability and understandability of Prolog programs.[30] Usually, if one wants to know, for example, which values a variable X in a head p(X,Y) may ever assume, one has to read 'backwards' through the body of the clause and follow all calls which may affect X, both in a transitive manner. With typing, this information is given in the head and can be used when reading the body.

In this paper we have used the typing notation mainly for explanatory purposes. Besides its obvious advantages for such purposes we think that explicit typing is useful in the further areas of maintenance, testing and debugging. Finally, explicit typing is required in correctness proofs, since formal or informal correctness proofs must be based on specifications of some sort in which typing information given explicitly by the programmer has to play an important part.[29] In our special situation of transforming an existing program the typing information had to be extracted from the sources in the complicated way hinted at above. We hope to have shown through the examples presented in this paper that Prolog programs with explicit typing do in fact become more readable and are easier to understand.

The debugger of a new Siemens Prolog system is based on the refined box model described in Reference 3 and in this paper.

ACKNOWLEDGEMENTS

We thank Klaus Dässler, Alexander Herold, Cosima Schmauch, and Heinz Seidl for a number of useful discussions and suggestions, and Axel von Reeken for the implementation of the user interface of ProTest. We also thank the referees for their valuable criticisms and suggestions which have led to a clarification in the cut implementation and to a better presentation of our typing concepts. Finally, thanks are due to John Campbell for numerous stylistic improvements.

APPENDIX: PROTOTYPE LISTING

The predicates read_choicept/1 and cut_choicept/1 are not readily available for a reader who wishes to experiment with the refined box model. We therefore give in Figure 6 a listing of a prototype version, which uses the technique of changing the database. To obtain a simple experimental debugger it is only necessary to copy this program, remove the type definitions from it, provide for the definitions of some predicates that we assume to be predefined (such as system_goal/1, length/2, etc.) and load everything into the database.

```
% The following predicate serves as a convenient entry point.

debug_goal(GOAL : <body> =>)    :-
     CXT = context(?- exec(GOAL), none),
     (  interpret_body(GOAL, CXT, none, _)
      ; cut_in_query).

% Restores the data-base when there has been a cut in GOAL.

cut_in_query  :-
     retract($cut_executed ),
     fail.

interpret_simplegoal(BOX : <and_box_with_procedure_definition> =>
                           <and_box_with_child_boxes>)    :-

     get_arg(1, BOX, GOAL),
     next_matching_clause(BOX, BODY),
     ( true
     ; clause($cut_executed, true), !, fail),   % Stop backtrack after cut.
     ( break(BOX, 'ENTERBODY')
     ; break(BOX, 'FAILBODY'),
       fail),
     CXT = context(GOAL :- BODY, BOX),
     interpret_body(BODY, CXT, none, LAST_CHILD),
     set_arg(4, BOX, LAST_CHILD),
     ( break(BOX, 'EXITBODY')
     ; break(BOX, 'REDOBODY'),
       fail).

next_matching_clause(BOX : <and_box_with_procedure_definition> =>
                           <and_box_with_selected_clause>,
                     BODY : <_> => <body>)     :-
     get_arg(5, BOX, procedure(N, CLAUSE_LIST)),
     next_in_list(CLAUSE_LIST, N, HEAD :- BODY),
     test_clause(BOX, HEAD :- BODY).

next_in_list(CLAUSE_LIST : <clause_list>,
             N : <_> => <integer>,
             HEAD : <_> => <legal_goal> :- BODY : <_> => <body>)   :-
     append(BEGINNING, [HEAD :- BODY | TAIL], CLAUSE_LIST),
     length(BEGINNING, L),
     N is L+1.

test_clause(BOX  : <and_box_with_selected_clause> =>,
            HEAD : <legal_goal> => :- BODY : <body> => )   :-
     get_arg(1, BOX, GOAL),
   ( break(BOX, 'TRYMATCH'),
     HEAD = GOAL, !
   ; break(BOX, 'FAILMATCH'),
     fail).
```

Continued

```
interpret_body((G1 : <legal_goal> =>, MOREGOALS : <body> =>),
               CXT : <context> =>,
               PREVIOUS_BOX : (<and_box> => ;; none),
               LAST_BOX : <_> => <and_box>)          :-
     !,
     interpret_subgoal(G1, CXT, PREVIOUS_BOX, NEW_BOX),
     interpret_body(MOREGOALS, CXT, NEW_BOX, LAST_BOX).

interpret_body(GOAL : <legal_goal> =>, CXT : <context> =>,
               PREVIOUS_BOX : (<and_box> => ;; none),
               LAST_BOX : <_>=> <and_box>)                :-
     interpret_subgoal(GOAL, CXT, PREVIOUS_BOX, LAST_BOX).
interpret_subgoal((G1 : <legal_goal> =>, MOREGOALS : <body> =>),
                  CXT : <context> =>,
                  PREVIOUS_BOX : (<and_box> => ;; none),
                  LAST_BOX : <_> => <and_box>)          :-
     !,
     interpret_body((G1, MOREGOALS), CXT, PREVIOUS_BOX, LAST_BOX).

interpret_subgoal(GOAL : <legal_goal> =>, CXT : <context> =>,
                  PREVIOUS_BOX : (<and_box> => ;; none),
                  NEW_BOX : <_> => <and_box>)          :-
     NEW_BOX = and_box(GOAL, CXT, PREVIOUS_BOX, _, _),
     test_goal(NEW_BOX, NEW_GOAL),
     debug_break_before(NEW_BOX),
     NEW_GOAL,
     debug_break_after(NEW_BOX),
     ( true
     ; not_backtrackable(NEW_GOAL),
       !, fail).

test_goal(BOX : <and_box>, exec_cut)    :-
     get_arg(1, BOX, GOAL),
     GOAL == !, !.
test_goal(BOX : <and_box>, exec_goal(BOX))    :-
     get_arg(1, BOX, GOAL),
     system_goal(GOAL), !.
test_goal(BOX : <and_box> => <and_box_with_procedure_definition>,
          interpret_simplegoal(BOX))     :-
     collect_clauses(BOX).

exec_cut.
exec_cut :-
     % Insert a fact $cut_executed to prevent backtrack to the left of cut.
     asserta($cut_executed), fail.

exec_goal(BOX : <and_box> =>)  :-
     get_arg(1, BOX, GOAL),
     GOAL.

collect_clauses(BOX : <and_box> =>
                      <and_box_with_procedure_definition>)  :-
```

Continued

```
    get_arg(1, BOX, GOAL),
    functor(GOAL, F, Arity),
    functor(T, F, Arity),
    findall(T :- Body, clause(T, Body), CLAUSE_LIST),
    set_arg(5, BOX, procedure(_, CLAUSE_LIST )).

debug_break_before(BOX : <and_box>)   :-
     ( break(BOX, 'CALL')
     ; break(BOX, 'FAIL'),
       get_arg(1, BOX, GOAL),                    % remove any clause
       GOAL \== !,                               % added previously on
       retract($cut_executed),                   % backtracking across
       fail).                                    % a cut operator.

debug_break_after(BOX : <and_box>)    :-
     ( break(BOX, 'EXIT')
     ; break(BOX, 'REDO'),
       fail).

not_backtrackable(_)  :-
     % Test whether backtrack is tried across a cut operator.
     clause($cut_executed, true), !.
not_backtrackable(exec_goal(BOX : <and_box>)) :-
     get_arg(1, BOX, GOAL),
     not_backtrackable_system_goal(GOAL).

type(<port> ) :-  ('CALL';;'EXIT';;'REDO';;'FAIL';;
                   'TRYMATCH';;'FAILMATCH';;'ENTERBODY';;
                   'EXITBODY';;'REDOBODY';;'FAILBODY').

% The reader may replace this rough and ready definition of break/2 by
% a more complete definition which suits his own taste and requirements.
break(BOX : <and_box>,  PORT : <port> )  :-
     telling(CURRENT_STREAM), tell(user), nl,
     get_parent_goal(BOX, PARENT_GOAL),
     write('parent : '), write(PARENT_GOAL), write(' :- ... '), nl,
     mark_goal(PORT),              % insert AND port
     get_arg(1, BOX, GOAL), write(GOAL),  nl,
     write_clauses(BOX, PORT),
     write('to continue press return key >>'),
     get0(_),
     tell(CURRENT_STREAM).

get_parent_goal(BOX : <and_box>,
                PARENT_GOAL : <_> => <legal_goal>) :-
     get_arg(2, BOX, CXT),
     ( get_arg(1, CXT, PARENT_GOAL :- _), !
     ; get_arg(1, CXT, ?- PARENT_GOAL)).

mark_goal(PORT : <port> ) :-
     member(PORT, ['CALL', 'EXIT', 'REDO', 'FAIL'] ), !,
     write(PORT), write('        ').              % eight blanks
```

Continued

```
mark_goal(PORT : <port> ) :-
     write('            ').                                % twelve blanks

write_clauses(BOX : <and_box>, _) :-
     get_arg(5, BOX, PROCEDURE),
     var(PROCEDURE), !,
     write('predefined predicate !'), nl.
write_clauses(BOX : <and_box_with_procedure_definition>, _) :-
     get_arg(5, BOX, procedure(_, [ ])), !,
     write('undefined predicate !'), nl.
write_clauses(BOX : <and_box_with_procedure_definition>,
              PORT : <port>)       :-
     write('procedure definition :  '), nl,
     get_arg(5, BOX, procedure(N, CLAUSE_LIST)),
     write_clauses(1, procedure(N, CLAUSE_LIST), PORT).

write_clauses(INDEX : <integer>,
              procedure(N : <_>,
              [CLAUSE : <clause>| REST : <clause_list>]),
              PORT : <port>)       :-
     ( INDEX == N,
       mark_current_clause(PORT)          % insert OR port
     ; write('            ')), !,         % twelve blanks
       write(CLAUSE), nl,
       I1 is INDEX + 1,
       write_clauses(I1, procedure(N, REST), PORT).
write_clauses(INDEX : <integer>, procedure(N : <_>, [ ]), _).

mark_current_clause(PORT) :-
     port_with_blanks(PORT, PWB),
     write(PWB).

port_with_blanks('CALL'         , '            ' ).
port_with_blanks('EXIT'         , '          * ' ).
port_with_blanks('REDO'         , '          * ' ).
port_with_blanks('FAIL'         , '            ' ).
port_with_blanks('TRYMATCH'     , 'TRYMATCH    ' ).
port_with_blanks('FAILMATCH'    , 'FAILMATCH   ' ).
port_with_blanks('ENTERBODY'    , 'ENTERBODY   ' ).
port_with_blanks('EXITBODY'     , 'EXITBODY    ' ).
port_with_blanks('REDOBODY'     , 'REDOBODY    ' ).
port_with_blanks('FAILBODY'     , 'FAILBODY    ' ).

% To improve readability we use the two predicates get_arg/3 and set_arg/3
% instead of the usual arg/3 wherever appropriate.
get_arg(N : <integer>, S : <structure>, A : <_> => ) :- arg(N, S, A).
set_arg(N : <integer>, S : <structure> =>, A : <_> ) :- arg(N, S, A).
```

Figure 6. Listing of a prototype of the ten-port debugger with the database technique to implement the c

REFERENCES

1. L. Byrd, 'Understanding the control flow of Prolog programs', *Workshop on Logic Programming*, Debrecen, 1980.
2. L. Byrd, 'Prolog debugging facilities', *Technical Report D.A.I. Occasional Paper No. 19*, University of Edinburgh, 1980.
3. J. F. H. Winkler, A. von Reeken and A. Schleiermacher, 'A Prolog debugger based on a refined box model', *Internal Report*, Siemens AG, Munich, 1988.
4. N. Francez, Sh. Goldenberg, R. Y. Pinter, M. Tiomkin and Sh. Tsur, 'An environment for logic programming', *Symposium on Language Issues in Programming Environments*, Seattle, U.S.A., 1985, pp. 179–190.
5. T. Yokoi, Sh. Uchida and ICOT Third Laboratory, 'Sequential Inference Machine: SIM its Programming and Operating System', *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984, ICOT (ed.), pp. 70–81.
6. D. Plummer, 'Coda: an extended debugger for PROLOG', *Logic Programming, Proc 5th International Conference and Symposium*, Seattle, USA, 1988, Vol. 1, pp. 496–511, MIT Press, Cambridge MA.
7. N. Numao and H. Maruyama, 'PROEDIT—a screen oriented Prolog programming environment', *Logic Programming '85, Lecture Notes in Computer Science Vol. 221*, Springer-Verlag, pp. 100–107.
8. S. Morishita and M. Numao, 'Prolog computation model BPM and its debugger PROEDIT2', *Logic Programming '86, Lecture Notes in Computer Science, Vol. 264*, Springer-Verlag, pp. 147–158.
9. J. Darlington, A. J. Field and H. Pull, 'The unification of functional and logic languages', in Doug DeGroot and Gary Lindstrom (eds), *Logic Programming, Functions, Relations, and Equations*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
10. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, 2nd edn, Springer Verlag, Berlin, 1984.
11. S. Uchida, *ESP Guide*, ICOT Research Center TM-0388, September 1987.
12. R. Venken, 'A prolog meta-interpreter for partial evaluation and its application to source to source transformation and query optimization' in T. O'Shea (ed.), *ECAI-84 Advances in Artificial Intelligence*, pp. 91–100.
13. R. A. O'Keefe, 'On the treatment of cuts in Prolog source-level tools', *Symposium on Logic Programming*, Boston, U.S.A., 1985, pp. 68–72.
14. Z. Somogyi, 'A system for precise modes for logic programs', *Proc 4th International Conference on Logic Programming*, 1987, pp. 769–787, MIT Press, Cambridge, MA.
15. R. Dietrich, 'Modes and types for Prolog', *Arbeitspapiere der GMD No. 285*, February 1988.
16. S. K. Debray and D. S. Warren, 'Automatic mode inference for logic programs', *J. Logic Programming* **5**, 207–229 (1988).
17. D. H. D. Warren, 'Implementing Prolog—Compiling predicate logic programs', *Research Report 39*, Dept. of Artificial Intelligence, University of Edinburgh, 1977.
18. C. S. Mellish, 'Some global optimizations for a Prolog Ccompiler', *J. Logic Programming*, **2**, 43–66 (1985).
19. P. J. Voda, 'Types of trilogy', *Logic Programming, Proc. 5th International Conference and Symposium, Seattle, U.S.A.*, 1988, Vol. 1, pp. 580–589, MIT Press, Cambridge MA.
20. K. Horiuchi and T. Kanamori, 'Polymorphic type inference in Prolog by abstract interpretation', *Logic Programming '87, Proc. 6th Conference,* Tokyo, 22–24 June, 1987, *Lecture Notes in Computer Science Vol. 315*, Springer-Verlag, Berlin, pp. 195–214.
21. H. Azzoune, 'Type inference in Prolog', *9th International Conference on Automated Deduction*, Argonne, Illinois, U.S.A. 23–26 May 1988, *Lecture Notes in Computer Science Vol. 310*, Springer-Verlag, Berlin.
22. M. Bruynooghe and G. Janssens, 'An instance of abstract interpretation integrating type and mode inferencing', *Logic Programming, Proc. 5th International Conference and Symposium*, Seattle, U.S.A., 1988, Vol. 1, pp. 669–683, MIT Press, Cambridge MA.
23. F. Kluzniak, 'Type synthesis for ground Prolog', *Proc. 4th International Conference on Logic Programming*, 1987, pp. 788–816, MIT Press, Cambridge, MA.
24. J. Xu and D. S. Warren, 'A type inference system for Prolog', *Proc. 5th International Conference and Symposium*, Seattle, U.S.A., 1988, Vol. 1, pp. 604–619, MIT Press, Cambridge, MA.
25. P. Mishra, 'Towards a theory of types in Prolog', *Proc. IEEE Int. Symposium on Logic Programming*, 1984, pp. 289–298.
26. M. Hanus, 'Polymorphic higher-order programming in Prolog', *Proc. 6th International Conference on Logic Programming*, 1989, pp. 382–397, MIT Press, Cambridge, MA.

27. L. Naish, P. W. Dart and J. Zobel, 'The NU-Prolog debugging environment', *Proc. 6th International Conference on Logic Programming*, MIT Press, Cambridge, 1989, pp. 521–536.
28. J. Zobel, 'Derivation of polymorphic types for Prolog Programs', *Proc. 4th International Conference on Logic Programming*, 1987, pp. 817–838, MIT Press, Cambridge, MA.
29. L. Naish, 'Specification = program + types' in K. V. Nori (ed.), *Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, Vol. 287*, Springer-Verlag, Berlin, 1987, pp. 326–339.
30. A. Mycroft and R. A. O'Keefe, 'A polymorphic type system for Prolog', *Artificial Intelligence*, **23**, 295–307 (1984).
31. H. Komatsu, N. Tamura, Y. Asakawa and T. Kurokawa, 'An optimizing Prolog compiler', *Logic Programming '86, Lecture Notes on Computer Science, Vol. 264*, Springer-Verlag, 1987, pp. 104–115.
32. Prolog, *Draft for Working Draft 1.0*, ISO Document N 28, February 1989, see Section 9.6 Database.