"LOCAL": ALGORITHMIC CONTROL STRUCTURES FOR PROLOG

D. C. Dodson
A. L. Rector

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

ABSTRACT

In practical Prolog applications, difficult and opaque uses of control primitives are often unavoidable. To relieve these difficulties, a preliminary set of high-level control predicates has been developed. Two technical goals were achieved. The first was to cast control facilities found desirable in conventional languages into a logic-programming form. The second was to provide convenient high-level structures for all the sorts of algorithmic routines a Prolog clause might sensibly perform. The availability of such structures should make deliberately algorithmic use of Prolog respectable and may help win Prolog wider use.

I INTRODUCTION

Many Prolog users accept that Prolog programs tend towards one of two extremes in a way that corresponds to two different sorts of problem. On the one hand, for many problems there are logically elegant solutions in Prolog. These require little resort to "cut", "fail" and "repeat" beyond some well structured use of "cut" in binding and finalizing clauses. Practical applications, however, often involve many predicates which display an algorithmic flavour, with opaque and difficult uses of "cut", "fall" and "repeat". These difficult uses of the control primitives severely disrupt the otherwise lucid self-documentation of sensibly written Prolog. This is costly, both In comprehension time and induced errors. Regarding control facilities, then, we would concur with the contention of (Hardy 1982) that Prolog offers only low level primitives comparable to the assembler level of conventional programming.

We assume that a much higher level of logic programming remains a relatively distant goal. Meanwhile, algorithmic programming structures are desirable both in principle and in practice. In principle, even at the program specification level, algorithmic content Is often Important. This can arise, for example, due to the relation of algorithms to time complexity. Well-structured algorithmic programming can thus have intrinsic self-documentary virtue. With regard to practice, programmers rarely have the opportunity to rewrite, In logic programming style, routines already well understood In algorithmic terms, and frequently there are no elegant Prolog formulations available.

There is thus a need to add to Prolog the means of well-structured algorithmic expression. We have been pleased to find how extensively the desirable facilities from conventional languages coutd be incorporated without damaging Prolog. This report presents our first implementation of the "Logal" control structures, which run on the PDP-11 UNIX Prolog system, version NU7, as sketched in appendix F of (ClocksIn and Mellish 1981).

II A GENERALIZED LOOP CONSTRUCT

Our offering comprises two programming constructs, a conditional and a generalized loop. The loop is the more innovative and important of these. It follows the practice of languages such as ALG0L68 in having a "building block" syntax for loops. The following fixed-order sequence of four optional "building blocks" or "phrases" is used.

```
<for-phrase>     [generator of Instances]
<while-phrase>   [exit test before main body]
<do-phrase>      ["main body" of loop]
<until-phrase>   [exit test after main body]
```

Each of these phrases consists of an appropriate keyword followed by a predicate to be called, for example "while read_line(L)'' or "until X=Y". We have allowed ourselves two arbitrary restrictions on the selection of phrases. First, a loop cannot consist of a while-phrase or until-phrase alone. Second, no loop can include both a while-phrase and an until-phrase. Whatever selection of phrases is made for a given loop, the phrases *are* executed in the order in which they appear, repetitively, until a termination condition arises. Adaptations of the conventional loop necessary to suit the Prolog environment are detailed below. Distinctive keywords are used to signal modified semantics.

Our keywords *are* declared as operators so that the phrases can be concatenated with a minimum of parentheses. This technique makes an additional mandatory closing keyword useful to avoid ambiguity and help catch syntax errors. We use "od" in this role for loops. Relying on available syntactic facilities has meant that parentheses are still required around arguments whose top level operator has a priority equal or greater than that of the keywords, as in the following.

```
do ( p1, p2 ) until p3 od
do ( do p1 until p2 od ) until p3 od
```

We *have* not Initially provided for-phrases generating number series. Rather, after the style of "every" in Icon (Griswold et. al. 1981) a series of outcomes is generated by calling a predicate and then repeatedly backtracking and resatisfying it until It fails. To signal this we write the for-phrase In the form "for each <term>". "for each" conveys the notion "for-each Instantlatlon~of" concisely and naturally. Generating arithmetic series is reasonably easy with this arrangement.

The next adaptation Is fundamental to the semantics of loops In the Prolog environment. It concerns the action to be taken if, on any cycle of iteration, the main body of the loop (I.e. the do-phrase) should fall. Two alternatives have been provided. Loops with do-phrases Introduced by the keyword "do any" succeed regardless of any cycles of Iteration in which their body falls. Loops with do-phrases introduced by the keyword "do one", however, fail immediately should their body once fall. For example, if the predicate:

```
    for_each ru!e(R)
      do_one ( translate(R,RT), store(RT) ) od
```

succeeds, then for each rule a translation has been found and stored. "for_each G do_one P od" in fact equivalent to the well known construct "not (G, not P)". The "do_one" form is in fact the general case, as "do_any X" is equivalent to "do one (X ; true)". The "do any" option is provided for convenience and efficiency. Loops which include a "do_one" phrase and a "while" or "until" phrase Incur overheads in distinguishing between two modes of termination, whereas the corresponding "do_any" loops do not.

The syntactic limitations of the Prolog interpreters we use require a final adaptation. Special treatment is required for phrases which are sometimes but not always the first phrase of a loop. For instance, when a for-phrase precedes a do-phrase, the do-phrase is not the first phrase in a loop, thus its first keyword has to be a dyadic infix operator. When a do-phrase is the first phrase in a loop, however, its first keyword must be a monadic prefix operator. This in turn requires the use of distinct keywords. Fortunately, when a loop begins with a do phrase, It is natural to say "repeat" instead of "do", as in Pascal. Likewise, we use "repeat one" instead of "do_one" and "repeat_any" instead of "do_any" in this context. This produces structures such as the following.

```
    repeat_one P until U od
    repeat_any P od
```

Neither for-phrases nor until-phrases can appear in both monadic and dyadic contexts, leaving the case of the while-phrase. At the start of a loop, we Introduce a while-phrase with plain "while", but use "whlle_still" otherwise, as shown below.

```
    while W do_any P od
    for_each  G~whlle_still  W do_one P od
```

The complete Implementation of the generalized loop Is given In Appendix I. This includes a clause for each format of loop that can be constructed using the above rules. Although there are 15 such clauses at the top level, this does not amount to a great deal of code. However, much of this code is excruciatingly difficult to write and read, underlining the ergonomic difficulties presented by the control primitives. The following relatively simple case illustrates something of this.

```
    repeat_one P until U od :- I,
        repeat,
        detval(P,V),    /I set V to success of P 17
        (    not V    ;    determine(U)    ),
        I, V, I.    /I whether the loop succeeds 17
```

where

```
    determlne(X) :- callto(X), I.    /! a call that
      /I can't be backtracked into from outside 17

    detval(X,true) :- callto(X), I.    /I call X *17
    detval(X,fall) :- !.     /I "determinate*" and
                 /I advise whether it succeeded 17

    callto(X) :- X.     /I unlike the system "call"
            /I predicate, a real call to fall back
            /I to In case X executes " I , fall". */
```

This works as follows, "repeat" always succeeds, so every time the loop is to be repeated, control will backtrack to this point and work forwards again, "detval" is then used to call P and according to whether or not P succeeds, set V to true or fall. Now, provided V=true, a call will be made to U. If either V=fail, or V=true and the call to U succeeds, the loop must terminate. All this Is handled by (not V ; determined!)). If this falls, control backtracks to the repeat for the next Iteration. If it succeeds, a cut is then passed which throws away the backtrack history of the call to the loop. The final call to V fails the loop if the body of the loop, P, failed.

In general, variable bindings made In a phrase which succeeds remain in scope for all subsequent phrases within a cycle. In the clause above, for Instance, "detval(P,V), (not V ; determlne(U))" ensures that variable bindings from successfully calling P remain in scope when calling U. Writing "(not P ; determlne(U))" would not achieve this. At the end of each cycle, all variable bindings are undone ready for a further cycle, unless Iteration is terminated by a while-phrase or untll-phrase. After termination by a while-phrase or untll-phrase, all bindings made in the last iteration remain in force. These rules are quite natural; they were formulated only after we had exploited them for some time. When not required, variable bindings introduced by a call are easily discarded by surrounding the call with a double "not". (Variable bindings, or Instantiations, represent existence. For any set of variables, their concurrent instantiations (those currently in scope) are things of which the successfully called predicates mentioning them are true.)

III THE CONDITIONAL CONSTRUCT

Apart from loops, two other high-level intramodular control structures are conventionally recognised, the conditional and the case structure. A case structure suggests a direct access control step. Such a step cannot be programmed within a Prolog clause, though some interpreters provide it in the form of indexed clause selection. We use the following predicate for conditionals. It is an extension of a "cond" predicate described in (Bundy and We l ham 1977). The operator declarations illustrate the technique used In both constructs.

```
    ?- op(150,fx,If_any).    ?- op(148,xf,fI).
    ?- op(146,xfy,then).     ?- op(145,xfx,else).
    ?- op(146,xfy,else_If_any).
    /* 145..150 are Increasing operator domInances
                between those of "not" and ",". */

    If_any X then Y else_If_any Z fI :-
                I, ( X, I, Y ; If_any Z fI ).
    If_any X then Y else Z fI :-
                I, ( X, I, Y ; Z ).
    If_any X then Y fI :-
                I, ( X, I, Y ; true ).
    If_any X :- bad_format(If_any).
```

Recursive application of the else if any option allows case-like structures. Note~that backtracking through antecedents is prevented whilst backtracking through selected consequents is unhindered.

IV EXAMPLE OF USAGE

This example Is chosen to Illustrate recursive application of the loop format discussed in detail above, rather than for elegance or even realism. It repeatedly performs a "consultation". After each consultation It repeatedly asks the user whether another is desired until it gets a valid reply. "read_line" obtains a line of input and returns it as a list of symbols.

```
toploop :-
    repeat_one
        (
        consultation,    /* whatever this is */
        write('Care for another consultation?')
        )
      until
        (
        /* first ensure a valid response: */
        repeat_one read_line([H|_])  /* use first
                    /* word of user's response */
          until
            (
            if_any not ( pos_ans(H) ; neg_ans(H) )
              then
                (
                write('Sorry, would you like '),
                write('another consultation?'), nl,
                write('say y (yes) or n (no): '),
                fail
                )
              fi
            )
          od ,
        neg_ans(H) /* finish if neg. response: */
        )
      od.

pos_ans(y). pos_ans(yes).
neg_ans(n). neg_ans(no).
```

V CONCLUSIONS

Extensions to the syntactic facilities of Prolog interpreters would allow certain improvements and might be of wider interest. If binary prefix operators were allowed, keywords could be defined as "fxy" operators of equal dominance, and distinctions such as that made between "while" and "while_still" could be dropped. Some arrangement to eliminate the remaining uses of parentheses around control predicate arguments may also be practical.

The generalized loop could be further refined. The option of writing for-phrases generating number series in the conventional manner would be an added convenience. More radically, it may be better to extend the the building block approach to allow an arbitrary sequence of do-phrases and loop termination phrases, optionally preceded by a for-phrase, using recursive definition along the lines employed here in the conditional construct.

We have identified the sorts of intra-modular control constructs that can usefully be programmed in Prolog and developed structured ways of writing them. The two constructs have been in regular use in substantially the form presented since last August. Appendix I defines the current implementation. We find the Logal constructs of great value in developing medical decision support systems. We hope they will make the mixed use of algorithmic and logic programming within Prolog more open and respectable, and help gain users for whom the language might otherwise seem too exotic. We would be interested to know of possible improvements.

ACKNOWLEDGMENTS

Special thanks are offered to all contributors to the excellent UNIX implementation of Prolog at Edinburgh and to J. B. Brooke whose criticism of Prolog provided the stimulus for this work.

APPENDIX I

Together with the detval, determine, callto, and if_any predicates given above, the following predicates for the generalized loop and for error detection complete the Logal package.

```
?-op(150,fx,for_each).      ?-op(150,fx,while).
?-op(150,fx,repeat_any).    ?-op(150,fx,repeat_one).
?-op(148,xf,od).
?-op(147,xfx,while_still). ?-op(146,xfx,until).
?-op(145,xfy,do_any).       ?-op(145,xfy,do_one).

for_each G while_still W do_one P od :- !,
    for_each_while_do_x(G,W,P), !, do_true, !.
for_each G while_still W do_any P od :- !, (
    G, detval(W,V), ( not V ; determine(P), fail )
    ; true ), !.
for_each G while_still W od :- !,
    ( G, not W ; true ), !.
for_each G do_one P until U od :- !,
    ( G, detval(P,V), if_any not V then
      assert(do_one_failed) else determine(U) fi
    ; true ), !, do_true, !.
for_each G do_any P until U od :- !,
    ( G, detval(P,_), determine(U) ; true ), !.
for_each G do_one P od :- !, not ( G, not P ).
for_each G do_any P od :- !,
    ( G, determine(P), fail ; true ), !.
for_each G until U od  :- !,
    ( G, determine(U) ; true ), !.
for_each G od :- !, (  G, fail  ;  true  ), !.
for_each X :- bad_format(for_each).

while W do_any P od :- !, repeat, detval(W,V),
    ( not V ; determine(P), fail ), !.
while W do_one P od :- !,
    for_each_while_do_x(repeat,W,P), !, do_true, !.
while X :- bad_format(while).

repeat_one P until U od :- !, repeat, detval(P,V),
    ( not V ; determine(U) ), !, V, !.
repeat_one X od :- !, repeat, not X, !, fail.
repeat_one X :- bad_format(repeat_one).

repeat_any P until U od :- !, repeat, detval(P,V),
    ( not V ; determine(U) ), !. /*rule exception*/
repeat_any P od :- !, repeat, not(P), !.  /* do. */
repeat_any X :- bad_format(repeat_any).

for_each_while_do_x(G,W,P) :- G, detval(W,V),
    ( not V ; not P,assert(do_one_failed) ; fail ).
for_each_while_do_x(X,Y,Z).

do_true :- not retract(do_one_failed).

bad_format(X) :- nl, write('bad '), write(X),
                     write(' format'), bug.
bug :- nl, backtrace, break.
```

REFERENCES

[1] S. Hardy "Languages for Knowledge Based Systems". Verbal presentation at B. C. S. Expert Systems 82 Conference, London, 14/9/82.

[2] Clocksin W. F. and Mellish C. S. "Programming in Prolog". Springer-Verlag, New York 1981.

[3] R. E. Griswold et. al. "Generators in Icon". A. C. M. Transactions on Programming Languages and Systems 3:2 (April 1981) 144-161.

[4] Bundy, A. and Welham, B. "Utility Procedures in Prolog". D.A.I. Occasional Paper No. 9, Dept. of A.I., University of Edinburgh, 1977.