Logic control with logic†

L. M. Pereira, Universidade Nova de Lisboa

1 A LOGICAL KIND OF CONTROL

In this paper, I shall be concerned with the control of top-down executions, with backtracking of logic programs consisting of Horn clauses. A case in point is Prolog, where a standard depth-first search strategy is enforced by the system, but where the need for other strategies can be felt.

However, I will only deal with the forward component of search. Elsewhere (Pereira and Bruynooghe, 1981; Bruynooghe and Pereira, 1984; Pereira and Porto, 1982), I have dealt with the backtracking component, irrespective of what the forward component may be.

The objective will be to provide a well-defined, easy-to-use methodology for enacting the control of logic programs, so as to render them more efficient, as well as to allow more freedom in their writing. The importance of this objective stems from the fact that the only practical programming language widely available, Prolog, can be very inefficient for executing some programs. Among these are programs whose declarative reading is highly perspicuous, but for which depth-first execution is prohibitive, if not impossible.

The emphasis of the approach will be on a kit of methods for building the desired control, and not on any particular control constructions, although important control constructs will be exhibited, namely those put forward in (Porto, 1982; 1982a) (which we find superior to other proposals for co-routining).

The kernel of the methodology consists in using interpreters written in logic both for specifying and exercising the desired control component over logic programs. The interpreters themselves are driven by a depth-first control strategy. The roots of this idea can be found in Pereira and Monteiro (1982), especially on pp. 631–633. Recent applications to debugging and concurrent programming in logic are developed in (Shapiro, 1983 and 1983a).

An interpreter exercises control by choosing which goals to match next with clause heads, clauses being used in the order given. Choice of the goals to be

---

† This is the paper with the same title published in the Proceedings of the First International Logic Programming Conference, Marseille 1982, enlarged with section 2.4 consisting of the short communication 'A Prolog demand driven computation interpreter', which appeared in *Logic Programming Newsletter*, No. 4, Winter 1982/83, and slightly altered with additional references.

matched will be the result of how the resolvent is structured, and of how that structure is interpreted. The resolvent may be structured with a combination of connectives, all having the declarative meaning of 'and', but eliciting different sequencing behaviour from the interpreter. To each structure there corresponds a behaviour that gives operational meaning to that particular combination of connectives. The connectives themselves are simply functors which take literals and other connectives as arguments. For every non-empty resolvent, the interpreter takes the corresponding term, threads through it as specified by the connectives, considers those literals next in order for activation, and replaces each of them by the body of the next available clause with which they match. Clause bodies, having been structured with the said connectives by the programmer, add to the structure of the ensuing resolvent.

In summary, the resolvent is a term which is rewritten by the interpreter clauses to form the next resolvent. They do so through selective replacement of some literals by the structured bodies of matched clauses.

Besides general clauses for each connective, which incidentally have the form of recursive rewrite rules, the interpreter may have additional program-dependent clauses for enforcing specific control over selected patterns of connectives and literals. A programmer may write both program and controlling interpreter, or rely on a pre-existing interpreter, perhaps supplementing it with particular control clauses. Furthermore, program clauses, rather than merely rewriting the head into the body, may also feature additional predicate calls, like the interpreter.

The advantages of this approach are:

- The interpreter clauses have a declarative reading and, being written in Prolog, allow the user to understand and verify exactly how they behave. Both the general and the specific control clauses are logical truths, or lemmas, that do not change the declarative semantics of programs, only their procedural semantics.
- The interpreter, driven by a depth-first strategy which has been efficiently implemented, can be compiled and so can the program clauses (as active data that perform pattern matching, but also include additional active predicate calls).
- The interpreter can be called from any point in a Prolog program, for selectively executing some goals only with the control it provides. Other goals are not burdened with any overhead required by the more elaborate control (piecewise control).
- The interpreter, being written in Prolog, can easily call system predicates, predicates defined by standard Prolog clauses, and itself (bootstrapping).
- Unification is inherited.
- The programmer can write his own control interpreter in Prolog itself, or rely on pre-existing interpreters written in Prolog, perhaps adding specific control clauses.

- Interpreter clauses can be expediently added or deleted for a particular program, and tuned to the specific control desired (modularity of control).
- Control interpreters take full advantage of Prolog and boost its expressive power as a programming language. Indeed, I shall systematise this methodology by defining in section 3 a language trivially implemented in Prolog called Epilog: extended logic programming.
- Low-level implementations of such interpreters are difficult and simply repeat the data structures more easily created by their more immediate implementation in Prolog (Filgueiras, 1984).

2 HOW TO CONTROL YOUR LOGIC

In this section I shall show how to write interpreters in logic to control logic programs, and exemplify and justify the approach. First, an interpreter for the standard depth-first strategy is shown. Second, an interpreter for performing delays, co-routining and sequencing is introduced. Third, one for controlling priority execution of goals. All the interpreters are assumed driven by Prolog, and are written in Edinburgh Prolog syntax. Additionally, I shall explain how to obtain the derivation tree from these interpreters.

2.1 An Interpreter to Control Depth-first Search

The interpreter uses the metapredicate '-|', written in postfix as in G -|, meaning G is proved or true where G is some goal expression. It also uses the infix binary functor '<-', as in H <- B, meaning H is implied by or rewrites to B, where H is a goal and B a goal expression. Goal expressions are either a goal or a conjunction of goals formed with the infix functor '&'. The program's Horn clauses have the form H <- B, where B is 'true' for unit clauses, and 'true' is a goal true under any interpretation. They can also have the form H <- B :- C, where goal expression C is a condition for H being implied by B. This is declaratively equivalent to H <- C & B, but more on this later.

Clauses are asserted using '-|'. Furthermore, the following non-standard operator declarations are in force: op(200, xfy, &), op(230, xfx, <-), and op(240, xf, -|).

Depth-first interpreter clauses:

```
I1   true  -| .
I2   Goals -| :- Goals <- New_goals -|, New_goals -|.
I3   true & Gs <-  NGs  -| :- Gs <- NGs -| .
I4     G & Gs <- NG & Gs -| :- G  <- NG  -| .
```

Format of user clauses:

```
unit      H <- true -| .   or   H <- true -| :- C .
nonunit   H <- B    -| .   or   H <- B    -| :- C .
```

Some comments are in order, first regarding the declarative semantics of the interpreter. Clause I2 is the inference rule 'modus tollens', the converse of 'modus ponens'. It states: Goals can be proved if some New_goals imply them

and New_goals are proved. ':-' stands for the metalanguage 'if', and ',' for the metalanguage 'and'. Clauses I3 and I4 state simple derived inference rules regarding the logical conjunction, expressed in the language by '&'. The reason why '&' is used in the language instead of ',' is to avoid use of parentheses.

The syntax of clauses relies on '<-' to express 'if' in the language. The reason ':-' is not used, besides avoidance of the use of parentheses, is that I will now drop the use of '-|' to assert program clauses, as is normal in Prolog; if ':-' was used instead of '<-', clauses of the form H <- B would become H :- B and would only be obtained with system predicate clause(H, B). Moreover, clauses of the form H<-B :-C would become (H :- B) :- C, and their head would not be obtainable by clause(H, B). This is due to the special treatment Prolog attaches to ':-' when it occurs as principal functor. Instead, I prefer to use '<-' which allows uniform access to H <- B in both cases.

The use of two language levels neatly expresses the possibility of temporary detachment from the computation going on at one level (the replacing of a head by a body) so as to initiate and complete some computation on the right of the ':-', at another level, before returning to the suspended one. Also, the two computations may share variables and the change of level may recurse.

The new rendering after '-|' is dropped from program clauses is:

```
J1  true -| .
J2  Gs  -| :- Gs <- NGs , NGs -| .

J3  true & Gs <-   NGs   :- Gs <- NGs .
J4    G & Gs <- NG & Gs :- G  <- NG  .

J5      G   <-   true   :- G \== true , G .

unit       H <- true .  or  H <- true :- C .
nonunit    H <- B    .  or  H <- B    :- C .
```

Clause J5 has been introduced to allow execution of system predicates, such as 'write', etc., calls to predicates defined by ordinary Prolog clauses, and also bootstrapping: goals of the form G-| are allowed, to enable calling the interpreter from within itself. The condition G\==true is required to avoid the loop true <- true.

Regarding the operational semantics of the interpreter, clause J1 and J2 define the termination condition and the recursive nature of the interpretative cycle: the resolvent is successively rewritten in an attempt to reach the empty resolvent, denoted by 'true', where Prolog's depth-first strategy is used to drive the interpreter. Clauses J3 and J4 state that the old resolvent is rewritten into the new one by threading through conjunctions of goals ignoring any 'true's until the leftmost goal is replaced by the body of a clause with which if matches.

N.B. Henceforth, 'true' will be dropped from unit clauses. Assume it is provided where needed as clauses are read in.

Fibonacci Example

A simple example is the Fibonacci function:

```
F1  fib(0, 1) <- .
F2  fib(1, 1) <- .
F3  fib(N, F) <- fib(N1, F1) &
                 fib(N2, F2) &
                 F is F1+F2  :-  N>1 , N1 is N-1 , N2 is N-2 .
```

A final ingredient of this programming methodology is now introduced: the ability to impose specific control clauses for specific combinations of goals. An example of its use is provided by the Fibonacci program, and the desire to avoid its redundant computations. In fact, the recursive call fib(N1, F1) has a subgoal equal to its uncle fib(N2, F2). The redundancy can be avoided by having the interpreter detect the pattern of the body of the recursive clause for 'fib' and identify the nephew with the uncle. To do so a special control clause is inserted before the general clauses for conjunctions, J3 and J4:

```
F4  fib(N1, F1) & fib(N2, F2) & Addition <- Body_for_N1 & Addition
                          :- N1>1 , N2 is N1-1 ,
                             fib(N1, F1) <- Body_for_N1 ,
                             Body_for_N1 = (fib(N2,F2) & Rest_of_body).
```

Translation: to solve the two consecutive brother calls for N1 and N2 plus the corresponding Addition, replace them simply by the Body_for_N1 and the Addition, just in case N1>1 and their brotherhood is checked (N2 is N1-1), where Body_for_N1 can be obtained by finding a clause for fib(N1, F1) whose body is bound to be of the form fib(N2, F2) conjoined to the Rest_of_body, thereby identifying nephew and uncle.

It is important to remark that this clause does not alter the declarative semantics of the original program. In fact, the extra clause is a tautology. To see it, note that all the conclusions (fib(N1, F1), fib(N2, F2) and Addition) are either part of the premises or implied by the premises by hypothesis. The fib(N2, F2) in the conclusion can be identified with the one figuring in the unification equality because the two arithmetic conditions are assumed.

Notice how the metalanguage part is performing a lookahead which allows control to profit from a structural property of the derivation tree.

Fux Example

A similar example is provided by the fux function (Dijkstra):

```
FX1  fux(1, 1) <- .
FX2  fux(N,F) <- fux(M, F)   :- even(N), M is N/2 .
FX3  fux(N,F) <- fux(M, FM) &
                 fux(Q, FQ) &
                 F is FM+FQ :- odd(N), M is (N-1)/2,
                                       Q is (N-1)/2 - 1 .
```

As before, a special control clause can be added to transform the double recursion into a single one:

```
FX4  fux(M,FM) & fux(Q, FQ) & Addition <- Body_for_Q & Addition
                    :- M>1, Q is M-1, even(M),
                       fux(Q, FQ) <- Body_for_Q,
                       Body_for_Q =
                                  ( fux(M,FM) & Rest_of_body ) .
```

Because Q is odd, the Body_for_Q will likewise contain two recursive calls, and so will be trapped by this same control clause, except when Q finally becomes 1.

2.2 An Interpreter to Control Co-routining

The novel logic programming methods put forth in the previous section – the rewriting of structured resolvents, the use of metalanguage features both in the interpreter and program, and the inclusion of specific control clauses for particular patterns – will be further emphasised in this section, to stress their power.

An interpreter is devised to attain the co-routining control regimes presented in Porto (1982; 1982a). These regimes are expressed by means of three basic connectives, used in nested combinations to structure the resolvent as a term whose terminals are goals. The connectives are '&' (sequentiation), '\' (co-routining) and ':' (delay). Declaratively they all read as the logical 'and', but operationally they elicit different resolvent rewriting behaviour from the interpreter.

Without loss of generality, as will be seen, we assume the top or principal functor of the resolvent to be always ':'. Let the resolvent be r1:r2. The intended operational semantics is that, to rewrite r1:r2, only r1 will be successively rewritten, and r2 thereby delayed, until r1 reduces finally to the empty resolvent 'true'. During its rewriting, r1 may produce subresolvent parts which are '\'ed with r2 to form r2′ : in fact, just those subresolvent parts occurring on the right of a ':' during the rewriting of r1.

When r1 becomes reduced to 'true', the new resolvent becomes r2′:true and the same process takes place, stopping only if and when true:true is reached. The rewriting thus takes place in cycles. The left side of the top ':' originates the current cycle, its right side the next cycle. When the first cycle is finished the next cycle becomes the current one, and an empty next cycle is created.

To rewrite r1\r2 simply do a rewrite step on r1 to obtain r1′, next a rewrite step on r2 to obtain r2′, and then rewrite r1′\r2′.

To rewrite r1&r2 rewrite r1 to 'true', even if it takes more than one cycle, and only then start rewriting r2. Thus, r2 may be delayed so as to be sequenced after the last of any delayed subresolvents of r1.

The best way to express, rigorously, the exact meaning of these constructs is to exhibit the clauses that interpret them. Their declarative and operational semantics are fixed by those of the interpreter clauses.

Co-routining interpreter clauses:

```
:- op(200,xfy,&), op(210,xfy,\), op(220,xfy,:), op(230,xfx,<-),
   op(240,xf,-|).
```



```
C1   true:true -|.
C2   true: B   -| :- B\== true    , B:true -| .
C3       G     -| :- G\==(true:_) , G <- RG , RG -| .
C4     true\true  <-      true:true     .
C5     true\B     <-        RB        :- B <- RB .
C6     A\true     <-        RA        :- A <- RA .
C7     A\B        <- CA\CB : NA\NB    :- A <- CA:NA ,
                                          B <- CB:NB .
C8     true&true  <- true:true          .
C9     true&B     <-   RB              :- B <- RB .
C10     A & B     <-   RAB             :- A <- RA , &(RA, B,
                                                     RAB) .
C11    true:B     <- true: B           .
C12     A : B     <- CA : B\NA         :- A <- CA:NA .
C13    &(true:true , B ,       B       ).
C14    &( CA :true , B , CA & B :  true ) :- CA\==true .
C15    &(true:NA , B ,  true  : NA & B) :- NA\==true .
C16    &(    A    , B , A & B :  true ) :- A=CA:NA,
                                            CA\==true, NA\==true.
C17    G <- true:true :- G\==true , G .
```

These clauses are not difficult to understand. Let me just point out that the bodies of program clauses are also assumed to have the form b1:b2. There is no loss of generality, since any body b can be transformed into b:true if necessary, as the clauses are read in. Also note that the conditions B\==true and G\==(true,_), in C2 and C3, are needed only to avoid looping on backtracking. The last clause allows bootstrapping, system calls, and calls to regular Prolog clauses. No cut (!) symbol is used.

N.B. Although this rendering of the interpreter is clearer than others, it is not very efficient as it stands. Use of the cut, more indexing, and non-normalised resolvents and clause bodies permit much more efficient versions, even surpassing in efficiency compiled Prolog in the case of programs that do benefit much from the special control. They also surpass lower-level implementations (see Filgueiras, 1984).

In the examples that follow, ':' will also be used as a prefix operator op(190,fx,:) such that a\:b stands for a\(true:b) and ::c for true:(true:c). Accordingly, the following interpreter clause is added:

```
C18   :G <- true:G .
```

Moreover, bodies of clauses of the form :r do not have 'true' added when read in. Similarly to our previous convention, H <- stands for H <- true:true and H <- :- C stands for H <- true:true :- C. Just assume clauses are changed as they are read in.

Admissible Pairs Example (Kowalski, 1979)

A list of pairs of integers is admissible if, for each pair(X,Y), Y is 2*X, and for any two consecutive pairs pair(X,Y) and pair(U,V) U is 3*Y. The problem is to generate an admissible list starting with pair(1,_).

```
A1  admissible(L) <- double(L) \ :triple(L) .
A2  double(pair(X,Y).L) <- ::double(L) :- Y is 2*X .
A3  triple(pair(X,Y).pair(U,V).L) <- ::triple(p(U,V).L) :- U is 3*Y.
```

Note how 'triple' in A1 is delayed to the next cycle, whereas 'double' starts in the current cycle. Thereafter 'double' and 'triple' take turns at being rewritten on alternate cycles. The successive resolvents are, in skeletal form:

CLAUSES APPLIED	RESOLVENTS
——	admissible -|
C3+C12+A1	double\:triple : true -|
C3+C12+C7+A2+C18	true\true : true\:double\triple -|
C3+C12+C4	true : true\:double\triple\true -|
C2	true\:double\triple\true : true -|
C3+C12+C5+C7+C18+C6+A3	true\true : true\double\:triple -|
C3+C12+C4	true : true\double\:triple\true -|
C2	true\double\:triple\true : true -|

Remark: the large number of 'true's is a consequence of our normalised notation; they can be done away with in less compact versions of the interpreter.

Sort Example

A sorted list is defined as an ordered permutation. A permutation U.V. of X.Y is generated by taking some element U of X.Y as first element, followed by a permutation V of the list W obtained from X.Y by deleting U from it. Note that 'del' is defined by standard Prolog clauses.

```
S1  sort(L,S) <- perm(L,S) \ :ord(S) .

S2  perm(nil,nil) <- .
S3  perm(X.Y,U.V) <- :perm(W,V) :- del(U,X.Y,W) .

S4  del(X,X.Y,Y) .
S5  del(X,U.Y,U.V) :- del(X,Y,V).

S6  ord(nil) <- .
S7  ord(X.nil) <- .
S8  ord(X.Y.Z) <- :ord(Y.Z) :- X=<Y .
```

With this control, as soon as an element of the permutation is generated a check is made whether it is greater than the preceding one.

CLAUSES APPLIED	RESOLVENT
——	sort -|
C3+S1	perm\:ord : true -|
C3+C12+C7+S3+C18	true\true : true\perm\ord -|
C3+C12+C4	true : true\perm\ord\true -|
C3+C12+C5+C7+S3+C6+S8	true\true : true\perm\ord -|
C3+C12+C4	true : true\perm\ord\true -|

etc.

Primes Example

The method known as 'the sieve of Eratosthenes' will be used to generate the primes greater than 1. It consists in sifting from the list of positive integers greater than 1 all the multiples of any of its elements. The program co-routines between generating the next integer and either filtering it from the list by detecting that it is a multiple of some previous element in the list or by creating a new co-routined process to filter from the list its multiples.

Clause P1 defines the overall co-routining between generating and sifting. P2 generates the integers. P3 sifts a list of integers by co-routining between filtering the list with respect to its first element and sifting the remaining list. As a side-effect, the first element (a prime) is output.

```
P1  primes <- integers(2,I) \ sift(I) .
P2  integers(N,N.I) <- integers(N1,I) :- N1 is N+1 .
P3  sift(P.I) <- filter(I,P,R) \ sift(R) :- write(P), nl .
```

However, a filtering process need only be activated when an integer is generated that is not filtered by the preceding filters. Hence P4: if N is filtered by P Other_filters are not activated; otherwise the general interpreter clauses for co-routining C7 and P5 are used and Other_filters (and eventually 'sift') will be activated. When N is not filtered by P, the P filter will continue on filtering the generated integers that follow. But before, control is passed on either to the next filter if it already exists, or to 'sift', which will then create the next filter.

```
P4  filter(N.I,P,R) \ Other_filters <- filter(I,P,R) \ Other_filters
                                        :- N mod P is 0 .
P5  filter(N.I,P,N.R) <- filter(I,P,R) :- N mod P =\= 0 .
```

For this program there is no need for the cycle or sequencing connectives. Besides, there are no unit clauses. So the controlling interpreter can be reduced to two clauses (and the program clauses used as they stand):

```
CP1   G -| :- G <- NG , NG -| .
CP2   A\B <- RA\RB :- A <- RA , B <- RB .
```

This shows how the control can be modulated to suit a particular program. Indeed, this interpreter can be made more efficient by writing it as:

```
CP1'  G -| :- G <- NG , NG .
CP2'  A\B :- A <- RA , B <- RB , RA\RB .
```

Same Leaves Example (Porto, 1982)

This program detects whether three binary trees have the same list of leaves or frontier. The lists of leaves from subtrees are appended together by using difference lists (Pereira and Monteiro, 1982). The program follows without further ado (see Porto (1982) for execution details):

```
L1  same_leaves_3(T1,T2,T3) <-
                    leaves(T1,L) \ leaves(T2,L) \ leaves(T3,L) .
```

```
L2  leaves( tree(Left, Right), L-X) <-
                        leaves(Left, L-R) & :leaves(Right, R-X) .
L3  leaves( Leaf, Leaf.X-X) <- :- atom(Leaf) .
```

Busy Waiting Example

Busy waiting of a goal G until some condition C is true is achieved with the interpreter clauses:

```
BU1  wait(C, G) <- NG :- C, !, G <- NG .
BU2  wait(C, G) <- wait(C, G) .
```

The sort example can now be written:

```
BS1  sort(L, S) <- perm(L, S) \ ord(S) .

BS2  perm(nil, nil) <- .
BS3  perm(X.Y, U.V) <- perm(W, V) :- del(U, X.Y, W) .
BS4  del(X, X.Y, Y) .
BS5  del(X, U.Y, U.V) :- del(X, Y, V).
BS6  ord( nil ) <- .
BS7  ord(X.nil) <- .
BS8  ord(X.Y.Z) <- wait( nonvar(Y), X=<Y ) \ ord(Y.Z) .
```

Synchronous Logic Example: Buffered communicating processes

Consider a producer with initial state s and a consumer with initial state t, which communicate through an initially empty buffer. The basic idea, inspired by Monteiro (1982), is that either the producer or the consumer only can be rewritten simultaneously with the buffer. The corresponding initial resolvent, where op(215, xfx, +), is:

```
producer(s) \ consumer(t) + buffer(nil) -|
```

Clause SL1 specifies that the goal expressions on the left of the '+' partake of the goal expression on its right. B can be rewritten synchronously with P to produce an intermediate IB which can be written synchronously with C. SL2, SL3 and SL4 explain how the producer and consumer processes update the buffer. Production takes the previous state and produces X and its new state, whereas consumption takes the previous state and X to produce its new state. In particular, SL4 allows the consumer to wait until the buffer becomes non-empty.

```
SL1  P\C + B <- NP\NC + NB :- P+B <- NP+IB , C+IB <- NC+NB .
SL2  producer(S) + buffer(L) <- producer(NS) + buffer(NL)
                        :- production(S, X, NS) , append(X.nil, L, NL) .
SL3  consumer(S) + buffer(X.L) <- consumer(NS) + buffer(L)
                                        :- consumption(S, X, NS) .
SL4  consumer(S) + buffer(nil) <- consumer(S) + buffer(nil) .
```


Bidirectional Search Example

The problem is to find a path P to go from some initial state Si to some final state Sf, in a graph, without repeating nodes, and by searching in both directions concurrently. The gap is defined by the symmetric predicate connected(_, _).

Predicate traj(Sc, Gs, SP, P) searches for subpath SP, with no repeated nodes, that continues from current state Sc and reaches one of the goal states in Gs, where the path from the start state is P. The list of goal states is an open list (with a variable in place of nil) that for the forward trajectory is being updated by the backward path BP, and for the backward trajectory by the forward path FP.

Clause B1 specifies that the problem is solved by co-routining both search directions until the two paths meet, and then constructing the solution path with them. B2 tests whether a goal state has been reached. B3 generates the next current state, checks that it hasn't been visited before on the current path, updates it and returns a new trajectory subgoal.

```
B1  go(Si, Sf, P) <- ( traj( Si, Sf.BP, FP, FP) \ traj( SF, Si.FP, BP, BP) )
                        & construct(Si.FP, Sf.BP, P) .
B2  traj(Sc, Gs, nil ,P) <-          :- member(Sc, Gs) .
B3  traj(Sc, Gs, S.SP, P) <- traj(NSc, Gs, SP, P) :- connected (Sc, NSc) ,
                                                     not member(NSc, P) ,
                                                     S=NSc .
B4  member(S, E.L) :- S==E .
B5  member(S, E.L) :- nonvar(E), member(S, L) .
```

To provide a lookahead breadth-first search of depth two on each path before proceeding to the next depth level, all that is needed is special control clause B6, to be inserted before the general co-routining clauses of the interpreter. It merely replaces a bi-directional search goal by a similar one, but with co-routined checks on the maximum length of the subpaths. If the attempt fails, the general interpreter clause for co-routining will then be used instead.

```
B6  traj(F1, F2, F3, F4) \ traj(B1, B2, B3, B4) <-
        traj(F1, F2, F3, F4) \ max_length(F3, 2) \
        traj(B1, B2, B3, B4) \ max_length(B3, 2) .
B7  max_length(E.L, N) :- N>0, nonvar(E), M is N-1, max_length (M, L) .
B8  max_length( V , 0) :- var(V) ; V=nil .
```

2.3 An Interpreter to Control Priority

In this section we exhibit interpreter clauses that can be added to the co-routining interpreter so that it first executes any goals which have priority before any others are considered in any resolvent preceded by '~', where op(225, fx, ~). The term G~NG, where op(225, xfx, ~), means G rewrites to NG by rewriting in turn all priority goals of G. NG may still contain priority goals, since some goal may have become a priority one only after subsequent goals were rewritten. Only when G==NG (formal equality) is there assurance that no priority goals are left.

Accordingly, the top level for ~G is:

```
PR1   ~G <- FG :- G ~ NG , ( G==NG, FG=G ; G\==NG, ~NG <- FG ) .
```

The next clauses allow penetration into the (sub)resolvent while preserving the meaning of the connectives:

```
PR2   A:B ~ NA:B    :- A ~ NA .
PR3   A\B ~ NA\NB   :- A ~ NA , B ~ NB .
PR4   A&B ~ NA&B    :- A ~ NA .
```

For single goals we need:

```
PR5   G ~ NG  :- priority(G) , G <- BG , BG ~ NG .
PR6   G ~ G   :- not priority(G) .
```

Where, for example, we have:

```
priority( append(X,Y,Z) ) :- nonvar(X) .
```

In this example, the priority is accorded to determinism (append is assumed deterministic for X nonvariable). For examples of the use of this control strategy see Pereira and Porto (1979).

2.4 An Interpreter to Control Demand Driven Computations

This interpreter realises the demand driven computation process described in Hansson *et al.*, (1982) from which all the examples are taken.

Predicate '#' evaluates any predicate call. If the predicate is functionally defined, the interpreter evaluates it recursively until a list is produced, where the head of the list, if any, contains the first result of evaluating the call, and the body a call to a functional predicate for producing the next result. The call [ ] evaluates to the empty list.

To do so, it uses predicate '@', which picks up a clause for a functionally defined predicate and evaluates its body of there is one. However, if any argument is demand driven and is not yet evaluated, no clause can be picked up and '@' will evaluate the demand driven arguments, and return to '#' the call with its arguments evaluated.

Predicate 'stream' accepts a functional predicate goal R and delivers a stream X of results, where difference lists are used to represent streams. After each value in the stream is produced it pauses, and displays the stream. If a <CR> is given it continues, if a <space> is given it shows the calls waiting to be demand driven and continues. Example call: stream(p=:X).

Predicate 'up_to' produces up to N values of a stream for a given call. Example call: up_to(3,conc([1,2],[3,4])=:X) .

```
?- op(230,xfx,=:) .
?- op(240,fx ,@) .
?- op(240,fx ,#) .
?- op(254,xfx,<-) .
```



```
/*    USER INTERFACE    */

stream(R=:X) :- s(R,X-X) .
s(R,X-Z) :- # R=:A , !, ( ( A=[] ; A=[_|T], list(T) ), Z=A    ;
                          A=[V|T], Z=[V|Y], show(T,X), s(T,X-Y) ) .
show(T,X) :- write(X), get0(C), ( C=32, write(T), skip(10), nl;
                                                  C=10 ), nl.
up_to(N,R=:[V|Y]) :- N>0, # R=:[V|S] , !, M is N-1, up_to(M,S=:Y).
up_to(_,_=:[ ]).

/*    INTERPRETER    */

# R=:S  :- ( list(R), R=S  ; @ R=:A , # A=:S ) .
# (A,B) :- # A , # B .
# G :- G .

list([ ]) .
list([_|_]) .

/* access to non-unit and unit functional predicate clauses,
   regular Prolog clauses, and system predicates                */

@ G  :- (G <- C) , # C .        /*    non-unit clauses    */
@ G  :- G .                     /* unit clauses, Prolog, and system */

/* user-specified information about demand driven arguments ;
   if backtracking is to be allowed into functional relations, then the
   condition 'not list(A)' should be included in the bodies of the clauses
   below, for each evaluable argument A                         */

@ conc(A,X) =: conc(EA,X)       :- # A=:EA .
@ select(N,A) =: select(N,EA)   :- # A=:EA .
@ sift(A) =: sift(EA)           :- # A=:EA .
@ filter(X,A) =: filter(X,EA)   :- # A=:EA .
@ merge(A,B) =: merge(EA,EB)    :- # A=:EA , # B=:EB .
@ mul(A,B) =: mul(A,EB)         :- # B=:EB .

/***    PROGRAMS    ***/

/*    conc    */

conc([ ],X)    =: X .
conc([X|Y],U) =: [X|conc(Y,U)].

/*    bounded buffer    */

bounded_buffer(WS,RS) =: AS <- bmerge(WS,RS,0,S1),
                                buffer(S1,U-U)=:AS .
```

```
bmerge([write(X)|WS], RS, I, [write(X)|AS]) :-
                                I<5, K is I+1, bmerge(WS, RS, K, AS) .
bmerge(WS, [read|RS], I, [read|AS]) :-
                                I>0, K is I-1, bmerge(WS, RS, K, AS) .
bmerge(_,[ ],_,[ ]) .

buffer([write(X)|S],V-[X|W]) =: buffer(S,V-W) .
buffer([read|S], [X|V]-W)    =: [X|buffer(S,V-W)] .
buffer([ ], _-[ ])           =: [ ] .

/*    infinite list of integers    */

intfrom2 =: inc(2) .

inc(X) =: [X|inc(K)] <- K is X+1 .

n_integers(N) =: Y <- intfrom2=:X , select(N,X)=:Y .

select(0,_)       =: [ ] .
select(N,[X|Y]) =: [X|select(K,Y)] <- N>0, K is N-1 .

/*    primes    */

primes =: sift(intfrom2) .

sift([X|Y]) =: [X|sift(filter(X,Y))] .

filter(X,[Y|Z]) =: [Y|filter(X,Z)] <- Y mod X =\= 0 .
filter(X,[Y|Z]) =: filter(X,Z)     <- Y mod X =:= 0 .

/*    quicksort    */

qs([ ])     =: [ ] .
qs([X|Y]) =: conc(qs(Y1),[X|qs(Y2)]) <- part(X,Y,Y1,Y2) .
part(X,[H|T],[H|S],R) := H=<X, part(X,T,S,R) .
part(X,[H|T],S,[H|R]) := H>=X, part(X,T,S,R) .
part(_,[ ],[ ],[ ]) .

/*    cyclic network of agents    */

p=:Y <- merge( mul(2,[1|Y]), merge( mul(3,[1|Y]),
                                    mul(5,[1|Y]) ) ) =: Y .
merge([X|Y],[U|V]) =: [X|merge(Y,[U|V])] <- X<U .
merge([X|Y],[U|V]) =: [U|merge([X|Y],V)] <- X>U .
merge([X|Y],[U|V]) =: [X|merge(Y,V)]     <- X=U .
mul(X,[Y|Z]) =: [W|mul(X,Z)]  <-  W is X*Y .
```

2.5 Obtaining the Derivation Tree

Next we show how to modify the previous interpreters to make them produce the derivation tree. The tree can be used to explain the computation if needed, as for expert systems. It can also be used for more elaborate control.

We illustrate the technique with the sequential depth-first interpreter (clauses J1–J5). To every goal expression G corresponds the term G ^ D, which pairs it with its derivation D, where op(225,xfx,^). To the application to G of a clause G <- B there corresponds in the derivation D the structure G <- DB where DB is a variable to be bound to the derivation of B.

```
D1  true ^ true  -| .
D2   G  ^ DG  -| :- G ^ DG <- NGD , NGD -|*.
D3  true & B ^    DB   <-        NBD          :- B ^ DB <- NBD .
D4   A  & B ^ DA & DB <- NA & B ^ DNA & DB
                                          :- A ^ DA <- NA ^ DNA .
D5  G ^ ( G<-true ) <- true ^ true  :- G\==true , G .
D6  G ^ ( G<-DG  ) <- NG ^ DG   :- G <- NG .
```

Clauses D1–D5 correspond to clauses J1–J5. D6 is now needed to access program clauses and build the structure that they contribute to the derivation.

3 EPILOG: EXTENDED PROGRAMMING IN LOGIC

According to my dictionary 'epilogue' derives from the Greek 'epilogos' = conclusion, from 'epi' = upon and 'logos' = speech,reason, to be conclusive upon what is spoken or reasoned, speaking about what is spoken or reasoned, in a conclusive way. Thus, it is a concept from the realm of the metalanguage.

Epilog aspires to a double meaning: as an epilogue to Prolog, and as a discourse over Prolog, as a logic that speaks conclusively about a logic, i.e. an epilogic, a metalogic, an extension of logic.

As a metalanguage, Epilog includes Prolog, although it is advantageously implemented in Prolog, because language and metalanguage are made to coincide, only notation distinguishing one from the other. The new notation is needed though, to make the metalanguage constructions more apparent and principled, much as with grammar rules as compared to standard Prolog.

Epilog is thus introduced to emphasise the basic concepts we found necessary for achieving control by taking in logic about logic. And these are the concept of explicitly rewriting goal expressions to perform controlled computations at the language level, and the concept of detaching from a computation going on at one level, have the ability to carry out a metalevel computation, and eventually return to the suspended one.

The functor '<-' is reserved for the first concept; and its stronger form '<->' is allowed. Several clauses using '<->' are permitted, it being understood that clauses are used in the order given and that clauses after a clause with H <-> B can only be used if pattern matching with H <-> B fails.

The functor '<=' is reserved to express metalanguage implication; its stronger form '<=>' is permitted, where again clause order is respected.

More rigorously, the following Epilog forms exist, where B may be 'true'

(or, equivalently, absent). Their Prolog counterparts, which are used to implement them, are shown as well.

EPILOG	PROLOG
A <- B .	A <- B .
A <-> B .	A <- B :- ! .
A <- B <= C .	A <- B :- C .
A <- B <=> C .	A <- B :- ! , C .
A <-> B <= C .	A <- B :- C , ! .
A <-> B <=> C .	A <- B :- ! , C , ! .

It should be clear how all the previous interpreters and examples can be written in Epilog notation ...

> I know not ... what impression I may have made, so far, upon your understanding; but I do not hesitate to say that legitimate deductions even from this part of the testimony ... are in themselves sufficient to engender a suspicion which should give direction to all further progress in the investigation of the mystery.
>
> Edgar Allan Poe in *The Murders in the Rue Morgue*, pp. 396 (Poe, 1971)

4 ACKNOWLEDGEMENTS

Thanks are due to Antonio Porto for assidous dialogue and to Maurice Bruynooghe for crtitical exposure.

In no way is the Instituto Nacional de Investigação Científica thanked for the support it did not give to this project.

5 REFERENCES

Bruynooghe, L. M. and Pereira, L. M., (1984), Deduction revision by intelligent backtracking, this volume.

Dijkstra, E., (word of mouth).

Filgueiras, M., (1984), On the implementation of control in logic programming languages, *Research Report*, Departamento de Informática, Universidade Nova de Lisboa.

Kowalski, R., (1979), *Logic for Problem Solving*, North-Holland.

Hansson, A., Haridi, S. and Tärnlund, S.-Å., (1982), Properties of a logic programming language in *Logic Programming*, (eds. K. Clark and S.-Å. Tärnlund), Academic Press.

Monteiro, L., (1982), *A Horn-clause-like logic for specifying concurrency,* First International Conference on Logic Programming, Marseille.

Pereira, L. M. and Bruynooghe, M., (1981), Revision of top-down logical reasoning through intelligent backtracking, *Research Report*, Departamento de Informatica, Universidade Nova de Lisboa.

Pereira, L. M. and Monteiro, L., (1982), The semantics of parallelism and co-routining in logic programming, *Conference on Mathematical Logic in Programming*, Salgotarjan, 1978, Proceedings published by North-Holland.

Pereira, L. M. and Porto, A., (1979), Intelligent backtracking and sidetracking in logic programs – the theory, *Research Report*, Departament de Informática, Universidade Nova de Lisboa.

Pereira, L. M. and Porto, A., (1982), Selective backtracking in *Logic Programming*, (eds. K. Clark, and S.-Å. Tärnlund), Academic Press.

Poe, E. A., (1971), *Tales of Mystery and Imagination*, Everyman's Library, Dent, London.

Porto, A., (1982), Coroutining for logic programs, *Research Report*, Departamento de Informática, Universidade Nova de Lisboa.

Porto, A., (1982), Epilog: extended programming in logic, *First International Conference on Logic Programming*, Marseille (and this volume).

Shapiro, E., (1983), *Algorithmic Program Debugging*, M.I.T. Press.

Shapiro, E., (1983), A subset of concurrent Prolog, *Preprints of Logic Programming Workshop '83, Albufeira,* Departamento de Informática, Universidade Nova de Lisboa.