COLLOQUIA MATHEMATICA SOCIETATIS JÁNOS BOLYAI
26. Mathematical Logic in Computer Science,
Salgótarján (HUNGARY), 1978.

THE SEMANTICS OF PARALLELISM AND CO-ROUTINING IN LOGIC PROGRAMMING

L.M. Pereira and L.F. Monteiro

ABSTRACT

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

Next we present definitions of sequential, parallel and co-routined executions of programs, in strictly logic programming terms, and go on to define in logic a parallel interpreter for logic programs, obtained by a simple program transformation from a purely sequential interpreter. We then show how similar transformations may be directly applied to programs to obtain transforms that achieve parallelism or co-routining without recourse to special interpreters. Afterwards, we apply our results to data base lookup and to problems arising from the use of negation as nonderivability, and suggest the basis of a rudimentary control language for logic programs.

We conclude by examining the features of logic which make logic programming specially suitable for parallel and co-routined modes of processing. In the final sections we furnish rigorous proofs of our results, to suplement the informal and intuitive basis used to motivate and derive them.

INTRODUCTION TO THE PROLOG LANGUAGE

Prolog is a simple but powerful programming language founded on symbolic logic, developed at the University of Marseille, as a practical tool for "logic programming" ([3], [6], [7], [8], [12]). A major attraction of the language, from a user's point of view, is ease of programming. Clear, readable, concise programs can be written quickly with minimum error. Recently, an efficient compiler and an interpreter were implemented on the DECsystem-10 ([9], [13]).

Like Lisp, Prolog is an interactive language designed primarily for symbolic data processing. Both are founded on formal mathematical systems - Lisp on the lambda calculus, Prolog on a powerful subset of calassical logic [11] Pure Lisp in fact can be viewed as a specialization of Prolog [14].

Introductory syntax

Here is a Prolog program, consisting of two clauses, for relating a tree with the list of its leaves (or frontier):

```
     leaves (t(void, N, void), [N,..Z] - Z).
     leaves (t(Stl, N, Str), L - Z): - leaves (Stl,
            L - X), leaves (Str, X - Z).
```

In general, a Prolog *program* consists of a set of *procedures*, where each procedure comprises a number of *clauses*. The procedure name is called a *predicate* ("leaves" above), and has an *arity* which is the number of its arguments (2 above). A clause begins with a *head* or procedure entry point, and continues with a *body*. If the body is not empty it is separated from the head by ":-" (2nd clause above). Every clause terminates with a ".". The head displays a possible form of the arguments to the procedure's predicate. The body consists of a number (possibly zero) of *goals* or procedure calls, which impose conditions for the head to be true. If the body is empty we speak of a *unit* clause (1st clause above).

In general, all Prolog objects are *terms*. A clause is a term, a predicate or a goal with their respective arguments, and the arguments themselves are terms. A term is either a *variable* (distinguished by an initial capital letter), an *atom* ("void" above), or a *compound term*. A compound term comprises a *functor* ("leaves" or "t" above) of some arity N $\geq$ 1, and a sequence of N terms as its arguments ("t(void, N, void)" above). An atom is treated as a functor of arity ∅. A term of the form [H,..T] stands for the list. (H, T), whose head is H and tail is T. The empty list is denote by [], and a list with exactly two elements by [A, B].

The term [N,..Z] - Z, where "-" is a binary funct written in (optional) infix notation, stands for a difference list [2]. A difference list L - Z stands for t list which concatenated with list Z produces list L. Difference lists provide a convenient way for appending the lists they denote. Above, the (difference) list L - Z is conveniently split into the (difference) lists L - X and X - Z.

The second clause above is just infix notation for the term

```
:- (leaves (t(stl, N, Str), L - Z),
    "," (leaves  Stl, L - X), leaves (Str,Z)))
```

where":-"and "," are binary functors. ":-" takes as arguments the head and the body of the clause, and "," the goals. This term stands for a clause because it figures in the set of clauses for a procedure. It is distinguished by a final".".

Apart syntax conventions, the names and arities of terms (and their number) are arbitrary, except for a pre-defined set of procedures which are built into the implementation of the language, and which achieve input, output, arithmetic, etc.

Semantics

Prolog differs from most programming languages in that there are two quite distinct ways to understand its semantics. The *procedural* or operational semantics is the more conventional, and describes as usual the sequence of states passed through when executing a program. In addition a Prolog program can be understood as a set of descriptive statements (one for each clause) about the state space of a problem. The *declarative* or denotational semantics, which Prolog inherits from logic, provides a formal base for such an understanding. Informally, one interprets terms as shorthand for natural language phrases by applying a uniform translation of each functor. e.g.:-

void = "the empty tree"
t(Stl, N, Str) = "the binary tree with root N, left subtree Stl and right subtree Str"

leaves (T, L - Z) = "the leaves of tree T are the elements of L - Z"

L - Z = "the list of elements of L after Z is tail subtracted from it"

A clause "$\underline{P}$: - $\underline{Q}$, $\underline{R}$, $\underline{S}$. "where $\underline{P}$, $\underline{Q}$, $\underline{R}$ and $\underline{S}$ are metavariables standing for terms, is interpreted as

"$\underline{P}$ if $\underline{Q}$ and $\underline{R}$ and $\underline{S}$"

A clause "$\underline{P}$." is interpreted as "$\underline{P}$ is true".

Each variable in a clause should be interpreted as some arbitrary object (i.e. variables are universally quantified). The type of the object will be appropriate to the functor(s) where it figures by using terms consistently throughout the program.

The declarative semantics then simply defines (recursively) the set of terms which are asserted to be true according to a program. A term is *true* if it is the head of some clause instance and each of the goals (if any) of that clause instance is true, where an *instance* of a clause (or term) is obtained by substituting, for each of zero or more of its variables, some term for all occurences of the variable.

Thus the only true instance of the goal: -

```
leaves (t(t(void, a, void), b, Str), [A, c] - []).
```

is

```
leaves (t(t(void, a, void), b, t(void, c, void)),
       [a, c] - []).
```

It is the declarative aspect of Prolog which is responsible for promoting clear, rapid, accurate programming. It allows a program to be broken down into small, independently meaningful units (clauses), and it allows some understanding of a program without looking into the details of how it is executed.

Procedural semantics

It is the procedural semantics that describes the way a goal is executed. The objective of execution is to produce true instances of the goal. It then becomes important to know that the ordering of clauses in a program, and of goals within a clause, which are irrelevant as far as the declarative semantics is concerned, constitute crucial *control information* for the procedural semantics.

To *execute* a goal, the system searches for the first clause whose head *matches* or *unifies* with the goal. The unification process [10] finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then *activated* by executing in turn, from left to right, each of the goals of its body (if any). If at any time the system fails to find a match for a goal it backtracks, i.e. it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal. Execution terminates if no goals remain to be executed (the system has then found a true instance of the original goal). Backtracking may then be provoked to find other true instances of the goal. Execution fails when no true instances of the original goal are found, and terminates if it cannot find any more true instances. Termination however cannot

be guaranteed, even if there are no more true instances (eg. if there are infinite branches).

Note that the execution just defined is a left to right depth-first process. Note also that because unification always provides the most general common instance between a goal and a matching clause, all the most general true instances of a goal can potentially be found (i.e. aside termination issues).

Let us now briefly look at how the goal: - leaves (t(t(void, a, void), b, Str), [A, c] - []) is actually executed. The goal only matches the second clause for "leaves". The body of the matching clause instance is: - leaves (t(void, a, void), [A, c] - X), leaves (Str, X - []). The result of executing the first of these two goals against the only clause it matches is to instantiate A to a and X to c. The second goal also matches the first clause, thereby instantiating Str to t(void, c, void) since X is already instantiated to c.

If the second goal were matched with the second clause, an attempt would be made to generate an infinite tree, which need not concern us now, but showing just how crucial can a convenient ordering of clauses be.

The connection between the execution mechanism defined above and logical derivability, and also the proof of completeness of the search space can be found in [5].

Basically, each execution step is justified by Robinson's Resolution Principle [10]. This principle subsumes in a single inference rule the classical rules of "modus ponens" and "generalization" in formulations of first order predicate calculus. For example, from: -

```
    p(X): - q(a, X), r(X).
```

and

```
    q (Y, f(Y, Z)): - s (a, Z).
```

it allows to conclude

```
        p(f(a, Z)) : s (a, Z), r (f(a, Z)).
```

by "execution" of q(a, X).

Note that computer-wise it is advantageous to have a single inference rule uniformly applicable.

Besides the ordering of clauses and the sequencing of goals within clauses Prolog provides just one other essential mechanism for specifying control information. This is the "cut" symbol, written "!". It is inserted in a program just like a goal, but it is not be regarded as part of the logic of the program and should be ignored as far as the declarative semantics is concerned.

The effect of the "cut" is as follows: when first encountered, as a goal, "cut" succeeds immediately. If backtracking should later return to the "cut", the effect is to fail the goal which caused the clause containing the "cut" to be activated. In other words, the "cut" operation *commits* the system to all choices made since execution of the goal activating the clause begun. I.e. other alternatives for that goal are not considered, as well as for all goals occurying in the matching clause before the "cut". By means of a "cut" one can ensure that some goals, once partly executed by a clause up to a "cut", either must continue that partial execution or fail. The "cut" renders deterministic the whole partial execution made by the activated clause up to it.

Example of the effect a "cut" in the flux of control, when goal F fails: -

```
        P: -    A,  B,   C.

        B: -    D,  E,   !,    F.
```

where A, B, C, D, E, F, and P are metavariables standing for predicate instances.

Backtracking returns to goal A immediately before B, the goal that activated the clause with the "cut". If there was no A in P control would return to the goal calling P.

Outstanding features of Prolog

To end this introduction to Prolog, let us briefly review the combination of features which make Prolog a powerful but simple to use programming language.

(1) A declarative semantics inherited from Logic in addition to the usual procedural semantics.

(2) Identity of form of program and data - clauses can be employed for expressing data, and can be manipulated as terms by interpreters written in Prolog.

(3) The input and output arguments of a procedure do not have to be distinguished in advance, but may vary from one call to another. Procedures can be multi-purpose. The procedure for "leaves" may be given completely or incompletely specified trees or lists of nodes in any combination.

(4) Procedures may have multiple outputs as well as multiple inputs.

(5) Procedures may generate, through backtracking, a sequence of alternative results. This amounts to a high level form of iteration.

(6) Terms provide general record structures with any number of record types may be used, and there are no type restrictions on the fields of a record.

(7) Pattern matching replaces the use of selector and constructor functions for operating on structured data.

(8) Incomplete data structures may be returned (i.e. containing free variables) which may later be filled in by other procedures.

(9) All communication between co-routined or concurrent procedures is ensured by the variables through unification. No explicit interfacing is needed.

(10) Prolog dispenses with *go to*, *do*, *for* and *while* loops, *assignment*, and *references* (pointers).

(11) The procedural semantics of a syntactically correct program is totally defined. It is impossible for an error condition to arise or for an undefined operation to be performed. This totally defined semantics ensures that programming errors do not result in bizarre program behaviour or incomprehensible error messages.

(12) No part of the program is concerned with the details of the underlying machine or implementation.

SEMANTICS OF PARALLELISM AND CO-ROUTINING IN LOGIC PROGRAMS

We begin this section by supplying definitions of sequential, parallel and coroutined execution of logic

programs. Next, we describe a sequential interpreter in clausal form, and from it derive an interpreter which achieves parallel execution of two goals, whilst preserving both declarative and procedural semantics. This parallel interpreter, obtained by a simple transformation of the clausal program for the sequential interpreter, is then generalized to execute n goals in parallel. This same transformation is then shown to apply directly to any program for obtaining parallel execution of a number of goals using just the sequential interpreter. Finally, another program transformation is provided which illustrates co-routining, and examination is made of the features of logic programming which make logic a unique language for parallel and co-routined processing.

Definitions of sequential, co-routined and parallel executions

We take unification to be, from the logic programmer's point of view, a single event. When a goal unifies with the head of a clause, all unification of arguments is considered simultaneous. It is not relevant for the programmer to know how the executor of the program performs unification, or to specify how it should be performed. Thus we take the match of a goal with the head of a clause as the single elementary unanalysed event in logic programs.

We say two goals are executed *sequentially* when execution of one goal begins only after execution of the other is completed.

We say two goals are executed *in parallel*, or concurrently, when execution of one goal is interleaved with the execution of the other, such that at most only a single elementary event takes place in one execution before another elementary event takes place in the other. I.e. an execution waits for the other no more than the completion of a match with the head of a clause.

We say two goals are executed in *co-routining* fashion in all other cases.

The above definitions have the virtue of providing rigorous, implementation and hardware independent notions of parallelism and co-routining which are meaningful from the logic programming point of view. Indeed, sequencing of goal calls is what traditional search strategies are all about.

Recall the program for "leaves", displayed in the introduction to Prolog: -

```
    leaves (t(void, N, void), [N,.. Z] - Z).

    leaves (t(Stl, N, Str), L - Z): - leaves
           (Stl, L - X), leaves (Str, X - Z).
```

The procedure: -

```
    same-leaves (T1, T2): - leaves (T1, L - []),
           leaves  T2, L - []).
```

will execute the two calls to leaves sequentially to find whether T1 and T2 have the same list L of leaves. Sequential execution in this case can be largely inefficient. For suppose the two trees differ in the first leaf. This will only be discovered after the first tree is already totally processed. Parallel execution of the two goals however would provide the best search strategy for arbitrary trees assuming each tree is going to be searched in a depth-first fashion as stated by the "leaves" procedure.

In general, parallelism of n goals has the further advantage that it always fails as soon as the goal which fails soonest does. For example, one or both of the trees might be ill-formed.

A sequential interpreter for logic programs

Imagine the sequential interpreter we are about to define reads in the clauses for "leaves" and re-writes them as: -

```
clause (( leaves (t(void, N, void), [N,..Z] - Z),
          C : - C )).

clause (( leaves (t(Stl,N,Str),L - Z), C: - leaves
          (Stl,L - X), leaves (Str, X - Z), C )).
```

I.e. each clause is re-written as the single argument of a unit clause for predicate "clause", after a variable C is conjuncted to both its consequent and its antecedent. Introduction of this variable on both sides of the implication does not modify the semantics of the program as long as C is guaranteed to be bound, during execution, to some predicate instance or conjunction of predicate instances. Nor does it go outside fist order logic, on the same condition. The extra parentheses around the single argument of "clause" are just syntax due to the infix notation used for ":-". The original clauses for "leaves" in an equivalent form, are thus supplied as data to the interpreter, by means of predicate "clause".

The clauses for the interpreter which accepts and executes this data are: -

```
    i(succeed).
    i(P) : - clause ((P: - C)), i(C).
```

where a call to the original program such as

```
leaves (T, L - [])
```

is replaced by an equivalent call to the interpreter of the form

```
i((leaves (T, L - []), succeed))
```

The effect of "i" is to accept a conjunction of goals terminated with "succeed" and to execute it as follows. It considers the first goal in the conjunct, looks for the first clause that matches the conjunction of that goal with the remaining sequence of goals, and accepts from the clause a new conjunction of goals to be executed. The new conjunction of goals is just the old one, where the first goal of the conjunct has been replaced by the goals in the body of the original clause. Eventually, execution terminates when the goal "succeed" is reached.

Thus, execution of a goal by "i" exactly mimicks the procedural semantics defined for Prolog. The difference lies in that "i" explicitly carries along the conjunction of outstanding goals awaiting execution. This requires the introduction of a variable C, on both sides of the implication in a clause, for receiving and passing along the conjunction of goals still awaiting execution (sometimes referred to as the "continuation"). "succeed" is then needed to express the empty continuation.

Proof of the equivalence between the declarative and procedural semantics of the original clauses and the program made up of the clauses for "i" and the clauses for "clause" is given in a later section.

Next, we show how to obtain a parallel interpreter "ip" from the above sequential one.

Define the new predicate "ip" as: -

```
ip(P1, P2): - i(P1), i(P2).
```

Now symbolically evaluate the calls to "i", by replacing them with the body of the clauses for "i" which they match - a process also known as "unfolding" [1].

Four clauses obtain: -

```
ip(succeed, succeed).

ip(succeed, P2): - clause ((P2: - C2)), i(C2).

ip(P1, succeed): - clause ((P1: - C1)), i(C1).

ip(P1, P2): - clause ((P1: - C1)), clause ((P2: -
                        C2)), i(C1), i(C2).
```

Finally, replace calls to "i" by calls to "ip" - a process known as "folding" [1] - using the facts

i(P) = ip(P, succeed) = ip(succeed, P) where "succeed" is the recursion base argument of "i", and the definition of "ip" to obtain: -

```
ip(succeed, succeed).

ip(succeed, P2): - clause ((P2: - C2)),
                   ip(succeed, C2).

ip(P1, succeed): - clause ((P1: - C1)),
                   ip(C1, succeed).

ip(P1, P2): - clause ((P1: - C1)), clause ((P2: -
                   C2)), ip(C1, C2).
```

The symmetry of "ip" on its arguments allows a simplification. The result is: -

```
ip(succeed, succeed).
ip(succeed, P2): - ip(P2, succeed).
ip(Pl, P2): - clause ((Pl: - Cl)), ip(P2, Cl).
```

Symbolical evaluation of these clauses plus symmetry reproduce the previous ones.

Again, formal proof of the procedural equivalence between the call i(P) and the corresponding calls ip(P, succeed) and ip(succeed, P) is left to another section as well as the declarative equivalence between the conjunct i(Pl),i(p2) and ip(Pl,P2). The proof is more general though. Let $\underline{P}$ and $\underline{Q}$ be metavariables standing for two predicates. Define a new predicate $\underline{P}$ & $\underline{Q}$ with the single clause.

$$\underline{P} \;\&\; \underline{Q}: - \underline{P},\ \underline{Q}.$$

where the arguments of $\underline{P \;\&\; Q}$ are obtained by adjoining the arguments of $\underline{P}$ and of $\underline{Q}$. We prove that the declarative semantics of $\underline{P}$ & $\underline{Q}$ is equivalent to that of $\underline{P}$ and $\underline{Q}$, where the clauses for $\underline{P \;\&\; Q}$ result from a simple transformation of the clauses for $\underline{P}$ and $\underline{Q}$. Furthermore, the declarative and procedural semantics of $\underline{P \;\&\; Q}$ preserves the original semantics of $\underline{P}$ or of $\underline{Q}$. The transformation in question is just a simple one-step unfolding produced by symbolic evaluation of the defining clause, followed by an appropriate folding step.

A program transformation for direct parallelism

Take the clauses for "i". Evaluate "clause" with respect to the "leaves" procedure. Call "l" the resulting procedure: -

```
l(succeed).
l((leaves(t(void,N, void), [N,..Z] - Z),C)):-l(C).
l((leaves(t(Stl,N,Str),L - Z), C)): - l((leaves
        (Stl,L - X),leaves (Str,X - Z), C)).
```

It directly mimicks the behaviour of "i" over "leaves". The only argument to this procedure specifies a sequence of disjoint subtrees of the original tree. The effect of the procedure is to flatten the original tree into terminal trees, by successively flattening its subtrees. Every time a terminal tree is found its only leaf is inserted into the list of leaves (cf. [1]). Similarly, evaluation in "ip" of "clause" with respect to "leaves", produces a procedure "lp" which directly mimicks the behaviour of "ip" over "leaves": -

```
lp(succeed, succeed).
lp(succeed, P): - lp(P, succeed).
lp((leaves(t(void, N, void), [N,..Z] - Z), C),
        P): - lp(P, C).
lp((leaves(t(Stl,N,Str),L - Z),C), P): - lp(P,
        (leaves(Stl,L - X), leaves(Str,X-Z),C)).
```

Of course, the transformation t relating "l" to "lp" is the same that relates "i" to "ip". In other words, let i[l] informally denote both the declarative and denotational semantics of program l when interpreted by interpreter i. We have: -

$$i[l] \subset i\,[t(l)] \equiv t(i)\,[l]$$

that is i [l] $\subset$ i [t(l)] and i [l] $\subset$ t(i)[l]. The transformation t preserves both semantics.

Interpretation of non-transformed clauses. Bootstrapping

Both interpreters "i" and "ip" may access non-transformed clauses. To do so the following clause must be provided: -

```
clause ((P, C: - C)): - P.
```

which directly executes P whilst leaving the continuation C untouched. The only requirement, as before, is that P, during execution, be instantiated to some term instance, so not to remain as a free variable.

An instance of this clause, viz: -

```
clause (( clause((P: - C)), Cl: - Cl)): -
          clause ((P: - C)).
```

allows any of the interpreters to interpret the other or itself (bootstrapping).

A parallel interpreter of n goals

The interpreter "ip" can be readily generalized from two to n goals: -

```
ipn([] - []).

ipn([succeed,..PS] - Z): - ipn(PS - Z).

ipn([P,..PS] - [C,..Z]): - clause ((P: - C)),
     ipn(PS - Z).
```

This interpreter takes as argument a difference list of goal conjuncts. It processes the first goal of the first conjunct in the list for just one elementary execution step, inserts its continuation on the back of the list of conjuncts, and continues processing on the next conjunct in the list. If one conjunct succeeds, it

continues processing the remaining ones, until eventually the list of conjuncts becomes empty.

A co-routining transformation

Reconsider the original "leaves" program. Define the new predicate

```
co-leaves (T1, L1, T2, L2):- leaves (T1, L1),
          leaves (T2, L2).
```

Now unfold it by symbolically evaluating the two calls to "leaves": -

```
co-leaves (t(void, N1, void), [N1,..Z1] - Z1, t(void,
          N2, void), [N2,..Z2] - Z2).

co-leaves (t(void, N1, void), [N1,..Z1] - Z1, t(Stl2,
N2, Str2), L2 - Z2): -

     leaves (Stl2, L2 - X2), leaves (Str2, X2 - Z2).

co-leaves (t(Stl1, N1, Str1), L1 - Z1, t(void,[N2,
void), [N2,..Z2] - Z2): -

     leaves (Stl, L1 - X1), leaves (Str1, X1 - Z1).

co-leaves (t(Stl1, N1, Str1), L1 - Z1, t(Str12, N2,
Str2), L2 - Z2): -

     leaves (Stl1, L1 - X1), leaves (Str1, X1 - Z1),

     leaves (Stl2, L2 - X2), leaves (Str2, X2 - Z2).
```

Folding is now accomplished by substituting calls to "co-leaves" for pairs of calls to "leaves". If there were single calls to "leaves" these would be replaced by a call to "co-leaves" where the missing arguments would be those of the unit clause of "leaves". In general, they can be any arguments known to make true the predicate in question.

A number of sets of pairings are possible. Only one such set though preserves the procedural semantics of "leaves", in the sense that a call leaves (T, L - []) is procedurally equivalent to the call co-leaves (T, L - [], T, L - []) : -

```
co-leaves (t(void, N1, void), [N1,..Z1] - Z1, t(void,
N2, void),[N2,..Z2] - Z2).

co-leaves (t(void, N1, void), [N1,..Z1] - Z1, t(Stl2,
N2, Str2), L2 - Z2): -
     co-leaves (Stl2, L2 - X2, Str2, X2 - Z2).

co-leaves (t(Stl1, N1, Str1), L1 - Z1, t(void, N2, void),
[N2,..Z2] - Z2): -
     co-leaves (Stl1, L1 - X1, Str1, X1 - Z1).

co-leaves (t(Stl1, N1, Str1), L1 - Z1, t(Stl2, N2,
Str2), L2 - Z2): -
     co-leaves (Stl1, L1 - X1, Stl2, L2 - X2),
               co-leaves (Str1, X1 - Z1, Str2, X2 - Z2).
```

The first and fourth clauses cover the cases where both trees find a leaf and where both trees are further decomposed into subtrees, respectively. The second and third clauses are responsible for proper co-routining, i.e. they cover the cases where one tree finds a leaf but processing on that ree is suspended until the other tree produces some more leaves.

Other choices of pairings of calls to "leaves" produce different search behaviours, making interesting the corresponding transformations. We refrain here from the details.

Parallel and co-routining processing features of logic programming

The potential of logic programming for parallel and co-routining processing is described in [6], [7]. Parallel or co-routined execution of goals is advocated to be indicated by the user in some sort of control language or, alternatively, an especially smart interpreter could have the initiative of recognizing its usefulness in some parts of a program. In our opinion a control language, including specification of parallelism or co-routining, is needed anyway for allowing the user to freely specify the control he thinks best.

The importance and utility of such a control language has been argued in [4]. It should provide the programmer with the ability to specify appropriate sequencing of goal calls without affecting the meaning of programs, influencing only their efficiency and/or their manner of finding solutions for goals.

Four features of logic stand out, in parallel co-routined processing of logic programs.

First, sequencing of goals is arbitrary from the declarative semantics point of view, and thus meaning is not altered by the particular type of execution chosen.

Second, logic programs may be used as data for other logic programs, thus allowing clauses (which are terms) to be submitted as data do particular interpreters, written in logic. The clauses for the interpreters provide thus a semantics of parallelism.

Third, in logic instantanenous communication between predicates is ensured by the logic variable. Thus parallel or co-routined executions do not require any explicit interfacing for data flow. Unification does it all. Moreover, partly specified data structures in the form of terms containing variables may be constructed by one process and further completed by another through instantiation of those variables. Execution control can be effected either by a special interpreter, or by the clauses themselves, as we have shown. Fourth it is meaningful to define sequential, parallel and co-routined executions in pure logic programming terms, without recourse to implementation or machine dependent concepts and capabilities.

Next we show how parallel processing might be indicated in Prolog programs, by means of a simple syntax. This syntax simply specifies the arguments to a particular interpreter. In the following section we shall exhibit an interpreter which performs parallel execution of all the goals in a clause. Likewise, we could show an interpreter for executing several clauses at once for the same predicate, which combined with the previous one would give the ability, to perform breadth-first executions. We believe that composition of calls to special interpreters forms the basis of a simple control language.

In the two examples shown next we also display a mechanism for directing an execution to wait for another, concurrently executed with it, until some condition is met.

Consider the following definition of grandparent: -

```
grandparent (X, Z): - parent (X, Y), parent (Y, Z).
```

which acesses data base of unit clauses for "parent" indexed on both arguments. According to whether only X or Z are already instantiated in a call to "grandparent", it becomes convenient to execute one of its two goals first. This is so because the indexing of the clauses gives direct access to the relevant unit clauses if one

of the arguments in the call is already instantiated. Thus, in the case where X or Z alone are instantiated, one would like one of the goals to await execution until execution of the other goal instantiates Y. This can be done using parallel processing. Rewrite "grandparent" as follows: -

```
grandparent (X, Z): - or ((atom (X), atom (Z))||wait
      or (atom (X), atom (Y)), parent (X, Y)|wait  or
      (atom (Z), atom (Y)), parent (Y, Z)||.

grandparent (X, Z): - and (var(X), var(Z)), parent (X,
     Y), parent (Y, Z).

or (P, Q): - P,!..

or (P, Q): - Q.

wait (P): - P,!.

wait (P): - wait (P).

and (P, Q): - P, Q.
```

where atom (A) is an implementation defined predicate which tests whether A is instantiated to an atom, and var (V) is also an implementation defined predicate that tests whether V is a variable which is not yet bound.

The effect of "wait" is to postpone its execution until the condition expressed in the argument is true.

The vertical bars indicate which groups of goal sequences are to be executed in parallel, where each sequence by itself is sequentially executed.

The first of these clauses would then be rewritten by the Prolog interpreter as: -

```
grandparent (X, Z): - or (atom (X), atom (Z)),
     ipn([(wait or (atom (X), atom (Y)),
     parent (X, Y), succeed), (wait or (atom (Z),
     atom (Y)), parent (Y, Z), succeed),..W] - W).
```

Another example of the use of "wait" combined with parallel execution refers to the use of negation as non-provability. This type of negation is accomplished in Prolog by the clauses: -

```
     not (P): - P, !, fail.

     not (P).
```

where "fail" always fails. A problem with this definition is that it does not in general respect the semantics of "not" in the case where P contains some unbound variables. For example, different solutions are found for

```
     r(X): - not (p(X)), q(X).
```

and

```
     r(X): - q(X), not (p(X)).
```

given

```
     p(a).

     q(b).
```

Now, for reasons of efficiency one might want the "not" to be executed first, unless X is not instantiated. This and similar problems can be solved by the use of "wait" in conjunction with parallelism. The above clause is then written: -

```
     r(X): -|| wait (notvar (X)), not (p(X))| q(X)||.
```

where "notvar" is an implementation defined predicate which tests that X is not an unbound variable. This way, execution of not (P(X)) is postponed until X is bound, while execution of Q(X) goes on, eventually binding X.

Other interpreters

The next interpreter provides parallel execution of all the goals in a clause. I.e. it performs a breadth-first execution of the *and* nodes of a derivation tree.

```
     ipb (succeed - succeed).

     ipb((P, L) - (C, Z)): - clause ((P: - C)),
                             ipb (L - Z).
```

where parallel execution of two goals, P and Q say, is achieved by the call

```
     ipb((P, Q, Z) - Z)
```

What "ipb" does it to insert each goal in a clause at the end of the sequence of goals to be executed in parallel, where each goal gives rise to a new separate execution.

A final interpreter is shown for completeness. It is just the sequential interpreter we started out from, re-written to account for the use of the "cut" in Prolog programs. It is meant only for Prolog knowledgeable readers.

Take a clause of the form

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

Imagine the interpreter re-writes it as

```
     P: - ! ((Q, R, succeed)), ! ((S, succeed)), T.
```

The clauses for the interpreter which appropriately deals with such clauses are

```
     i (succeed).

     i (P): - clause ((P: - C)), (ifcut (C, Cl, C2),
                     i(Cl), !, i(C2); ifnocut (C),i(C)).
```

```
i((!(C1), C2)): - i(C1), !, i(C2).
ifcut ((!(C1), C2), C1, C2).
ifnocut ((!(C1), C2)): - !, fail.
ifnocut (C).
```

THE SEMANTICS OF THE INTERPRETERS

The semantics of Prolog Programs

In this section we formalize the behaviour of Prolog programs interpreted by the interpreters presented above, thus complementing the intuitive descriptions made at the appropriate places. Our formalization is what might be called the procedural (operational) semantics of the interpreters, because it describes the computation sequences obtained when the interpreters are supplied with data.

We begin by formalizing Prolog programs themselves and their semantics. This is easy, if we consider only Prolog programs consisting exclusively of strictly logical features, which we shall do henceforth. Such programs will be called "declarations", following [12].

It is advantageous to depart here a little from the notations employed in the earlier part of the paper. This will be done as needed. For the time being, let us write unit clauses in the form "A: - " instead of the form "A.". We define now a *declaration* to be a (finite) set of clauses of the form

$$A: - A_1, \ldots, A_n$$

where A, $A_1, \ldots, A_n$ are (positive) literals and $n \geq 0$. We shall consider the literals inside each clause

ordered from left to right and the clauses inside each declarations also ordered, from top to bottom as they are written.

The inputs to declarations are *goal statements*, which are clauses of the form

$$: - A_1, \ldots, A_n$$

where $A_1, \ldots, A_n$ are positive literals, called *goals* of the goal statement, and $n \geq 0$. The special case in which $n = 0$ is the *null clause*, usually denoted □. As for any other clause, we consider the goals inside each goal statement as ordered.

Let <u>D</u> be a declaration. We say goal statement G *derives directly* a goal statement G′ iff

$$G \text{ is } : - A_1, \ldots, A_n \quad \text{with} \quad n > 0$$

and

$$G' \text{ is } : - B_1\theta, \ldots, B_m\theta, A_2\theta, \ldots, A_n\theta,$$

where there is a clause $B:-B_1, \ldots, B_m$ in <u>D</u> and θ is the most general unifier of A_1 and B. (Notice that if $n = 1$ and $m = 0$ then G′ is the null clause.) In this definition, the clause $B: - B_1, \ldots, B_m$ is assumed to have no variable in common with G; if necessary, the variables occurring in the clause may be renamed.

The goal statement G is said to *derive* a goal statement G′ iff there is a sequence of goal statements $G_o, G_1, \ldots, G_n$ $(n \geq 0)$, called a *derivation* of G′ from G, such that $G_o = G$, $G_n = G'$ and G_{i-1} derives directly G_i for $i = 1, \ldots, n$. A refutation of G is a derivation of the null clause from G. It is well known [5] that <u>D</u> ∪ {G} is unsatisfiable iff there exists a refutation of G.

We now define a *derivation tree* for every declaration D and goal statement. G, as a tree containing all possible derivations from G. The root of the tree is labelled with G. If some node is labelled with a goal statement G_1, then there is an arc from this node to a node labelled with a goal statement G_2 iff G_1 derives directly, G_2. (When determining the direct descendents of a node labelled with G_1, a new node is created for every G_2 such that G_1 derives directly G_2, even if there is already a node labelled with G_2. Bearing this in mind, we shall from now on confuse the name of a node with its label, in order to simplify the exposition.) It is clear that every path starting at the root is a derivation from G and conversely. Thus $D \cup \{G\}$ is unsatisfiable iff there is a finite branch of the derivation tree of G whose leaf is the null clause.

We order the direct descendents of each node of a derivation tree as follows: if G_2 and G_3 are direct descendents of a common node G_1, we say G_2 is *generated before* G_3 iff the clause used to derive G_2 from G_1 occurs first in D than the clause used to derive G_3 from G_1. With this definition we are apt to order the derivations from a goal statement G. Given two distint derivations

(D) $G = G_o, G_1, \ldots, G_n$

(D') $G = G'_o, G'_1, \ldots, G'_m$

we say D is *generated before* D' iff

- either $n < m$ and $G_i = G'_i$ for $i = 0, \ldots, n$;
- or, for some index $i \leq \min\{n, m\}$, $G_i \neq G'_i$, and if k is the least such index then G_k is generated before G'_k.

Further, we say D *precedes immediately* D' iff for no derivation D" is D generated before D" and D" generated before D'.

Let us say a derivation D is *final* iff it is a refutation or no other derivation is generated after it.

By the *computation* of the declaration D with input goal statement G we mean the sequence of derivations beginning with the derivation consisting of a G alone and such that, for every derivation D in the sequence:

- either D is final, in which case it is the last derivation in the sequence;
- or precedes immediately the derivation that follows it in the sequence.

A computation will be said to be terminating if it is finite, otherwise *non-terminating*. A successfull computation is a terminating computation whose last derivation is a refutation.

We thus see that the computation of D with input G is the top-down depth-first search of the first final derivation in the derivation tree of G. Notice that it may well happen that $D \cup \{G\}$ is unsatisfiable, yet the computation of D with input G is non-terminating. This is so iff there exists an infinite branch of the derivation tree of G which is generated before any finite branch whose leaf is the null clause.

This concludes our description of the procedural semantics of Prolog programs.

The sequential interpreter

Recall that the sequential interpreter consists of the two following clauses:

$$i(\sigma): -$$

$$i(P): - c((P:-C)),\ i(C)$$

where we are using for short 'σ' and 'c' instead of 'succeed' and 'clause' respectively.

Given a declaration $\underline{D}$ we define the declaration $i(\underline{D})$ to consist of the two above clauses plus a unit clause

$$c((A,\ C: - A_1, \ldots,\ A_n,\ C)): -$$

for each clause $A: - A_1, \ldots, A_n$ in $\underline{D}$. We assume that the order of these clauses in $i(\underline{D})$ is the same as the order of the respective original clauses in $\underline{D}$. We assume further that the predicate symbols 'i' and 'c' as well as the constant 'σ' are not among the predicate symbols and constants occurring in $\underline{D}$.

If G is an input goal statement to $\underline{D}$, say

$$: - A_1, \ldots,\ A_n$$

we denote the term $A_1, \ldots, A_n, \sigma$ by G^σ (if n = o, so that G is $\square$, we have $G^\sigma = \sigma$). This way,

$$: - i\ (G^\sigma)$$

is the input goal statement $- i((A_1, \ldots, A_n, \sigma))$ to $i(\underline{D})$. (Note that in the term $A_1, \ldots, A_n, \sigma$ the comma is just a binary functor used in infix notation, with association on the right implicitly assumed. The extra-parenthesis above specify that $A_1, \ldots, A_n, \sigma$ is a single term and not a sequence of arguments.)

We want to relate the computation of $\underline{D}$ with input G with the computation of $i(\underline{D})$ with input $: - i(G^\sigma)$. For the rest of this section we shall keeps this notation fixed.

Let D be a derivation $G = G_o, G_1, \ldots, G_k$ from G in $\underline{D}$. Denote by D_{i1} the following sequence of goal statements:

$$:- i\ (G_o^\sigma)$$

$$:- c((G_o^\sigma\ :- C)),\ i(C)$$

$$:- i(G_1^\sigma)$$

$$\vdots$$

$$:- i(G_k^\sigma\ ).$$

Denote further by D_{i2} the previous sequence D_{i1} followed by the null clause, if G_k is the null clause, or by $:- c((G_k^\sigma\ :- C));\ i(C)$ otherwise

LEMMA 1.

(a) D_{i1} and D_{i2} are derivation from $: - i(G^\sigma)$ in $i(D)$, and every such derivation has one of these two forms.

(b) D_{i1} precedes immediately D_{i2}.

(c) If D' is another derivation from G in $\underline{D}$ such that D precedes immediately D', then D_{i2} precedes immediately D'_{i1}.

Proof

(a) We shall only prove that D_{i1} is a derivation $:- i(G^\sigma)$ in $i(\underline{D})$. It is clearly enough to prove that if G_j

derives directly G_{j+1} in $\underline{D}$ then :- $i(G_j^\sigma)$ derives directly :- $c((G_j^\sigma$:- C)), i(C) and this last goal statement derives directly :- $i(G^\sigma{}_{j+1})$ in $i(\underline{D})$. Let G_j and G_{j+1} be respectively

$$:- A_1, \dots, A_n \quad (n < 0)$$

and

$$:- B_1 \ , \dots, B_m \ , A_2 \ , \dots, A_n$$

where there is a clause B:- $B_1, \dots, B_m$ in $\underline{D}$ such that is the most general unifier of A_1 and B. Then :- $i(G_j^\sigma)$ is

$$:- i((A_1, \dots, A_n, \sigma))$$

which derives directly

$$:- c((A_1, \dots, A_n, \ \sigma \ :- C)), \ i(C)$$

that is

$$:- c((G_j^\sigma \ :- C)), \ i(C)$$

by means of the clause i(P) :- c((P:- C)), i(C) of i(D). Now the clause.

$$c((B, C' :- B_1, \dots, B_m, C')):-$$

is in $i(\underline{D})$ and $c((A_1, \dots, A_n, \ \sigma$:- C)) is unifiable with it, with most general unifier the substitution σ' given by

$$\theta' = \theta \ \{C/(B_1 \ \theta, \dots, B_m \ \theta, \ C')\} \ \{C'/(A_2 \ \theta, \dots, A_n \ \theta, \ \sigma)\}.$$

Therefore :- $c((G_j^\sigma$:-C)), i(C) derives directly

$$:- i((B_1 \ \theta, \dots, B_m \ \theta, \ A_2 \ \theta, \dots, A_n \ \theta, \ \sigma))$$

which is :- $i(G_{j+1}^\sigma)$.

(b) Clear.

(c) It is clear that if D is generated before D' then D_{i2} is generated before D'_{i1}. The statement of (c) follows immediately from this. □

THEOREM 2.

If D^0, $D^1, \dots, D^n, \dots$ is the computation of $\underline{D}$ with G then

$$D^0{}_{i1}, \ D^0{}_{i2}, \ D^1{}_{i1}, \ D^1{}_{i2}, \dots, \ D^n{}_{i1}, \ D^n{}_{i2}, \dots$$

is the computation of $i(\underline{D})$ with input :- $i(G^\sigma)$.

Proof

Use the lemma and induction on k to prove that the (2k)-th (resp. (2k+1)-th) derivation in the computation of $i(\underline{D})$ with input :- $i(G^\sigma)$ is D^k_{i1} (resp. D^k_{i2}). □

We may obtain another description of the "behaviour" of $i(\underline{D})$ with input :- $i(G^\sigma)$ if we restrict our attention to goal statements of the form :- $i(G'^\sigma)$. By an *i - derivation* from :- $i(G^\sigma)$ in $i(\underline{D})$ we mean the sequence of goal statements of the form :- $i(G'^\sigma)$ obtained from some derivation from :- $i(G^\sigma)$ in i(D). By the lemma, every i - derivation is of the form :- $i(G_o^\sigma)$, :- $i(G_1^\sigma), \dots,$:- $i(G_k^\sigma)$ for some derivation $G = G_o$, $G_1, \dots, G_k$ from G. If we denote this last derivation from G in $\underline{D}$ by D, the resulting i - derivation will be denoted i(D). By the *i - computation* of i($\underline{D}$) with input :- i(G^σ) we mean the sequence of i - derivations, without repetition, obtained from the computation of i($\underline{D}$) with input :- i(G^σ). From the theorem it follows that if D^0, D^1,..., D^n,... is the computation of $\underline{D}$ with input G then i(D^0), i(D^1),..., i(D^n),... is the i - computation of i($\underline{D}$) with input :- i(G^σ).

The parallel interpreter

The parallel interpreter consists of the following clauses

i_p (σ, σ): -

i_p (σ, P): - i_p (P, σ).

i_p (P, Q): - c ((P:- C)), i_p (Q, C).

As for the sequential interpreter, we associate with every declaration $\underline{D}$ a declaration i_p($\underline{D}$), which is defined in exactly the same way as i($\underline{D}$). It consists of the three above clauses plus a clause

c((A, C:- A_1,..., A_n, C)):-

for every clause A: - A_1,..., A_n in $\underline{D}$.

Let G and H denote respectively the terms A_1,...,A_n and B_1,..., B_m (n, m $\geq$ O), where the A's and B's are (positive) literals. Denote further A_1,...,A_n, σ and B_1,..., B_m, σ by G^σ and H^σ respectively (this notation departs a little from the one previously used), so that :- G, H and :- i_p(G^σ, H^σ) are inputs to $\underline{D}$ and i_p($\underline{D}$) respectively.

It would be interesting to compare the computations of $\underline{D}$ with input :- G, H and of i_p($\underline{D}$) with input :- i_p (G^σ,H^σ).(In the first case the terms G and H are computed sequentially, and in the latter they are computed in parallel.) It may happen, however, that the computation of $\underline{D}$ is successfull and the computation of i_p($\underline{D}$) is non-terminating, and conversely. Consider, for example, the following declaration $\underline{D}$:

```
q(f,(X),  Y):- q(X, Y)
q(X, g(Y)):- q(X, O)
q(O, O):-
m(O, O):-
m(f (X), O):- m(X, O)
m(f (X), g (Y)):- m(f (X), g(Y))
```

The computation of $\underline{D}$ with input :-q(f(O), Y), m(f (O), Y) is non-terminating and the computation of i_p($\underline{D}$) with input :- i_p((q(f(O), Y),σ), (m(f(O), Y), σ)) is successfull. This fact makes it difficult to compare the above mentioned computations.

We shall follow another approach to describe the computation of i_p($\underline{D}$). Reconsider the definition of "direct derivation" presented before. According to this definition, to derive directly a given goal statement from another goal statement we had to match the *first* literal in the last goal statement with the head of some clause, and the first goal statement was obtained by resolution without factoring or merging. The reason for using such a restrictive notion of "direct derivation" is because this is the way Prolog has been implemented. We shall now abandon this restriction and consider the possibility of selecting other literals than the first in goal statements for matching with the heads of clauses. Consider the following method of obtaining a derivation in $\underline{D}$ with top goal statement :- G, H. Every goal statement occurring in the derivation may be written in the form :- G_k, H_k, where G_k or H_k or both may be empty, so that the derivation itself is of the form :- G_o, H_o; :- G_1, H_1;...; :- G_n, H_n. Here G_o is G and H_o is H. Consider now the goal statement :- G_k, H_k. If G_k and H_k are both empty then :- G_k, H_k is the null clause and consequently is the last goal statement in the derivation. Otherwise let us distinguish two cases:

(1) k is even: if G_k is not empty select the first literal in G_k, else the first literal in H_k.

(2) k is odd: if H_k is not empty select the first literal in H_k, else the first literal in G_k.

In either case, the selected literal is matched with the head of some clause in $\underline{D}$ (if such a clause exists) and the next goal statement is obtained by resolving the given goal statement and clause without factoring or merging. The resulting goal statement can be written in the form :- G_{k+1}, H_{k+1} where if the literal chosen in :- G_k, H_k belongs to G_k, then :- G_k derives :- G_{k+1} and H_{k+1} is an instance of H_k; else :- H_k derives :- H_{k+1} and G_{k+1} is an instance of G_k.

Such a derivation will be called a *p - derivation* from :- G, H in $\underline{D}$, where the 'p' indicates that G and H are executed in parallel (see above for the justification of the last statement). We speak of a *p - refutation* when the last goal statement in a p - derivation is the null clause. We may say of two p - derivations when one of them is generated before the other, and therefrom

we define the concepts of "p - derivation tree" and "p-computation".

We now describe the computation of $i_p(\underline{D})$ with input :- $i_p(G^\sigma, H^\sigma)$ by describing its i_p - computation, to be defined below, and comparing it with the p - computation of $\underline{D}$ with input :- G, H. The concept of i_p - computation is analoguous to the concept of i - computation defined in the section on the sequential interpreter. By an i_p - *derivation* of :- $i_p(G^\sigma, H^\sigma)$ in $i_p(\underline{D})$ we mean the sequence of goal statements of the form :- $i_p(-,-)$ obtained from some derivation from :- $i_p(G^\sigma, H^\sigma)$ in $i_p(\underline{D})$. The i_p - *computation* of $i_p(\underline{D})$ with input :- i_p (G^σ, H^σ) is the sequence of i_p - derivations, without repetitions, obtained from the computation of $i_p(\underline{D})$. By a lemma analoguous to lemma 1, we may associate with every p - derivation D from :- G, H a unique i_p - derivation $i_p(D)$ from :- i_p (G^σ, H^σ), and every i_p - derivation may be obtained in this way. Next, by analogy with Theorem 2, we may say that if $D^o, D^1,..., D^n,...$ is the p - computation of D with input :- G, H then i_p (D^o), $i_p(D^1),..., i_p(D^n),...$ is the i_p-computation of $i_p(\underline{D})$ with input :- $i_p(G^\sigma, H^\sigma)$.

This finishes our description of the behaviour of $i_p(\underline{D})$ with input :- $i_p(G^\sigma, H^\sigma)$. From this description we may conclude that the parallel interpreter is an implementation in Prolog of the "parallel strategy" described above for the execution of $\underline{D}$ when the goals in the input goal statements for $\underline{D}$ are separated into two sets G and H. Notice that if G (or H) is empty then the parallel execution of :- G, H coincides with the usual (sequential) execution of :- H (or:-G). We conclude this section by showing that this parallel strategy is complete.

THEOREM 3.

Of $\underline{D}$ ∪ {:- (G, H)} is unsatisfiable then there is a p - refutation of :- G, H in $\underline{D}$.

Proof

Our proof will be directed towards reducing the problem of finding a p-refutation of :- G, H to the problem of finding a refutation of :- G, H with respect to a convenient selection function defined for goal statements based on $\underline{D}$. We therefore start by defining such a selection function ρ and then use a result by [5] (Theorem 3, p. 26) which assures us of the existence of a refutation of :- G, H with respect to ρ. We conclude the proof by showing that this refutation is also a p-refutation.

We say a goal statement is "based" on $\underline{D}$ iff all the function and predicate symbols occurring in it also occur in $\underline{D}$. By a "selection function" for $\underline{D}$ we mean a function ρ assigning to each goal statement based on $\underline{D}$ (except the null clause) a literal occurring in it. A goal statement G "ρ-derives directly" a goal statement H iff H is obtained from G and a clause in $\underline{D}$ whose head matches ρ(G) by resolution without factoring or merging. Let us define a selection function ρ for all non-null goal statements based on $\underline{D}$. This will be done by constructing a sequence $\underline{G}_o$, $\underline{G}_1, \dots, \underline{G}_k, \dots$ of ordered sets of goal statements based on $\underline{D}$, and ρ will be defined on these sets by induction on k. Further, every goal statement in any set $\underline{G}_k$ will have the form :- G , H . $\underline{G}_o$ is to consist of the single goal statement :- G, H, which we may assume to be different from the null clause, and ρ selects the first literal in G if G is not

empty, otherwise the first literal in H. Assume that $\underline{G}_k$ and the selection function σ on $\underline{G}_k$ have already been defined. A goal statement :- G", H" belongs to $\underline{G}_{k+1}$ iff some :- G', H' in $\underline{G}_k$ σ-derives directly :- G", H" and :- G", H" does not belong to $\underline{G}_i$ for any i ≤ k. The goal statements in $\underline{G}_{k+1}$ are ordered in the obvious fashion: for :- G''_1, H''_1 and :- G''_2 in $\underline{G}_{k+1}$, pick the first :- G'_1, H'_1 and :- G $_2$, in $\underline{G}_k$ which ρ-derive directly :- G''_1, H''_1 and :- G''_2, H''_2 respectively; if :- G $_1$, H $_1$ is equal to :- G'_2, H'_2 then :- G''_1, H''_1 occurs in $\underline{G}_{k+1}$ before :- G''_2, H''_2 iff the clause used to generate :- :- G''_1, H''_1 occurs in $\underline{D}$ before the clause used to generate :- G''_2, H''_2; otherwise, :- G''_1, H''_1 occurs before :- G''_2, H''_2 iff :- G'_1, H'_1 occurs before :- G'_2, H'_2. To define ρ on $\underline{G}_{k+1}$ let :- G", H" belong to $\underline{G}_{k+1}$ and :- G', H' be the first goal statement in $\underline{G}_k$ which ρ-derives directly :- G", H". If G" (resp. H") is empty then ρ selects the first literal in H" (resp. G"), unless H" (resp. G") is also empty, in which case :- G", H" is the null clause; otherwise ρ selects the first literal in G" if it selected the first literal in H , else it selects the first literal in H". This completes the definition of ρ for goal statements belonging to the sets $\underline{G}_k$. For any other goal statement based on $\underline{D}$, ρ is defined arbitrarily. Now according to the already mentioned result by [5] it follows that there is a ρ-refutation of :- G, H in $\underline{D}$. This implies that the null clause belongs to some G_k. We may then construct a ρ-refutation of :- G, H consisting of a sequence of k+1 goal statement such that the i-th goal statement belongs to $\underline{G}_{i-1}$. It is clear that this refutation is also a p-refutation, thus finishing the proof of the theorem. □

The n-parallel interpreter

This interpreter is entirely analogous to the previous one, with the only difference that any number of goal statements may be executed in parallel, instead of two; thus the use of lists instead of n-tuples. For this reason we shall attempt no description of the computation of this interpreter, which would be in all relevant ways similar to the description of the computation of the parallel interpreter.

Co-routining

Let there be given a declaration D̲ and suppose p and q are respectively an n-place and an m-place predicate symbols occurring in D̲. When presented with an input :- $p(t_1,\ldots, t_n)$, $q(t'_1,\ldots, t'_m)$ D̲ will compute the literals $p(t_1,\ldots, t_n)$ and $q(t'_1,\ldots, t'_m)$ sequentially. We shall define a transformation on D̲ such that when the transform of D̲ is presented with a convenient input it can be said to compute the two above literals co-routiningly. This transformation consists in eliminating from D̲ the symbols p and q, in creating a new (n+m) - place predicate symbol p & q, and in substituting the clauses in D̲ where p & q occur by new clauses. The transform of D̲ will be denoted D̲ (p & q).

Let us call a p-*literal* a literal of the form $p(x_1,\ldots, x_n)$ and a p-*clause* a clause whose head is a p-literal. We define similarly q-literal and q-clause. We shall begin by defining D(p & q) only in the case where there exist a unit p-clause and a unit q-clause in D̲. Later we shall indicate how to define D̲(p & q) when this requirement is not met.

The declaration D̲(p & q) is the union of the three following sets of clauses:

(1) The set of all clauses in D̲ where neither p nor q occur.

(2) The set of all clauses C obtained by the following procedure applied to clauses C' in D̲ containing p or q but which are neither p-clauses nor q-clauses:

(a) if the number of p-literals in the body of C' is different from the number of q-literals, adjoin to the body of C' as many p-literals or q-literals as necessary so as to make their numbers equal; these p-literals (resp. q-literals) may be any instances of heads of unit p-clauses (resp. q-clauses) in D̲;

(b) set up a bijective correspondance between the p-literals and the q-literals in the body of the clause obtained in (a) and substitute each pair ($p(x_1,\ldots, x_n)$, $q(y_1,\ldots, y_m)$) in correspondance by the p & q-literal p & $q(x_1,\ldots, x_n, y_1,\ldots, y_m)$.

(3) The set of all clauses C_{12} obtained by applying the following procedure to a p-clause C'_1 and a q-clause C'_2 in D̲:

(a) rename the variables occurring in C'_2 (for instance) so that they are different from those occurring in C'_2 (for instance) so that they are different from those occurring in C'_1;

(b) if the head of C'_1 is $p(x_1,\ldots, x_n)$ and the head of C'_2 is $q(y_1,\ldots, y_m)$, construct a clause C''_{12} whose head is p & $q(x_1,\ldots, x_n, Y_1,\ldots, y_m)$ and whose body is the set of literals, which belong to the body of C'_1 or of C'_2;

(c) finally, transform C''_{12} into C_{12} by applying to C''_{12} the procedure applied to C' in (2).

The input to $\underline{D}$(p & q) associated with the input :- $p(t_1,\ldots, t_n)$, $q(t'_1,\ldots, t'_m)$ to $\underline{D}$ is :- p & q $(t_1,\ldots, t_n, t'_1,\ldots, t'_m)$. Let us abbreviate these inputs to :- p, q and :- p & q respectively.

THEOREM 4.

$\underline{D}$ ∪ {:- (p, q)} is unsatisfiable iff $\underline{D}$(p & q) ∪ {:- p & q} is unsatisfiable (i.e. the declarative semantics are equivalent).

Proof

Suppose there is a model of $\underline{D}$ ∪ {:- (p,q)}. This model is a subset M of the Herbrand base of $\underline{D}$ ∪ {:-(p, q)}.

Let M(p & q) be the subset of the Herbrand base of $\underline{D}$(p & q) ∪ {:- p & q} consisting of the literals in M which are neither p-literals nor q-literals, plus a p & q-literal p & $q(x_1,\ldots, x_n, y_m)$ for every p-literal $p(x_1,\ldots, x_n)$ and every q-literal $q(y_1,\ldots, y_m)$ in M. It is clear that M(p & q) is a model of $\underline{D}$(p & q) ∪ {:-p & q}. Conversely, suppose M(p & q) is a model of $\underline{D}$(p & q) ∪ {:- p & q}. A model M of $\underline{D}$ ∪ {:- (p, q)} can be constructed by letting M consist of all the non-p p & q-literals in M(p & q) plus literals $p(x_1,\ldots, x_n)$ and $q(y_1,\ldots, y_m)$ for every p & $q(x_1,\ldots, x_n, y_1,\ldots,y_m)$ in M(p & q). This finishes the proof. □

In the definition of $\underline{D}$(p & q) we did not bother to order the clauses inside $\underline{D}$(p & q) and the literals inside each clause for two main reasons. In the first

place, this was not necessary to state and prove Theorem 4. More importantly, we did not want to impose unnecessary restrictions from the outset on the kinds of co-routining of p and q that could be obtained. But now we want to compare the computation of $\underline{D}$ with input :- $p(x_1,\ldots, x_n)$ with the computation of $\underline{D}$(p&q) with input p & $q(x_1,\ldots, x_n, b_1,\ldots, b_m)$ where $q(b_1,\ldots, b_m)$ is an instance of the head of some unit q-clause. (By symmetry, we also want to compare the computation of $\underline{D}$ with input :- $q(y_1,\ldots, y_m)$ with the computation of $\underline{D}$(p & q) with input :- p & $q(a_1,\ldots, a_n, y_1,\ldots, y_m)$, where $p(a_1,\ldots, a_n)$ is an instance of the head of some unit p-clause, but this comparison is made in a manner entirely analogous to the previous one so we will not mention it any further.) This will dictate the orderings $\underline{D}$(p & q) must be supplied with.

We may assume that the clauses in $\underline{D}$ are ordered thus: all p-clauses come first, then come q-clauses, and finally the remaining clauses. Indeed, as far as any computation of $\underline{D}$ is concerned, we may permute at will the clauses of $\underline{D}$, provided that the clauses in each of the three sets above maintain their relative sequential positions. We may arrange the clauses in $\underline{D}$(p & q) so that the p & q-clauses come first, followed by the remaining clauses. The relative positions of the remaining clauses are the same as the relative positions of the respective clauses in $\underline{D}$. As to the p & q-clauses, let C_{12} (resp. C_{34}) be a p & q-clause obtained from a p-clause, C'_1 (resp. C'_3) and a q-clause C'_2 (resp. C'_4); then C_{12} occurs before C_{34} iff either C'_1 occurs before C'_3 or $C'_1 = C'_3$ and C'_2 occurs before C'_4.

In order to arrange the literals inside the clauses of $\underline{D}$(p & q) we have to alter slightly the definition of

D(p & q). Theorem 4 still remains valid, however. We fix an instance $p(a_1, \ldots, a_n)$ of the head of a unit p-clause, and assume that this unit p-clause occurs in D before any other p-clause with whose head $p(a_1, \ldots, a_n)$ matches. We fix similarly $q(b_1, \ldots, b_m)$ and make the corresponding assumption. Now recall that D(p & q) is the union of three sets of clauses. Accordingly, we order the literals in the clauses of D(p & q) in three steps:

(1') The clauses of this set also belong to D, so the order in which the literals are written is the same as in D.

(2') Here the clause C is now obtained from C' by rewriting each p-literal $p(x_1, \ldots, x_n)$ (resp. each q-literal $q(y_1, \ldots, y_m)$) as $p \& q(x_1, \ldots, x_n, b_1, \ldots, b_m)$ (resp. $p \& q\ (a_1, \ldots, a_n, y_1, \ldots, y_m)$), while leaving the ordering of the remaining literals unchanged.

(3') Let C_{12} be the p & q-clause obtained from the p-clause C'_1 and the q-clause C'_2.

a') If neither C_1 nor C_2 are unit clauses, or if they are both unit clauses, then C_{12} is defined as before and the literals are ordered arbitrarily.

b') Otherwise order the literals in the body of C''_{12} as they are ordered in the one of C'_1 and C'_2 which is not a unit clause, and transform C''_{12} into C_{12} by applying to C''_{12} the procedure applied to C' in (2').

For any goal statement G' based on D let G be the goal statement based on D(p & q) obtained by applying to G' the procedure applied to C' in (2'). We may now state the following theorem, whose (easy) proof we omit.

THEOREM 5.

(a) If $D' = (G'_o, G'_1, \ldots, G'_k)$ is a derivation from :- $p(x_1, \ldots, x_n)$ in D, then $D = (G_o, G_1, \ldots, G_k)$ is a derivation from :- $p \& q(x_1, \ldots, x_n, b_1, \ldots, b_m)$ in D (p & q).

(b) If $D'_1, D'_2, \ldots, D'_r, \ldots$ is the computation of D with input :- $p(x_1, \ldots, x_n)$, then $D_1, D_2, \ldots, D_r, \ldots$ is the computation of D(p & q) with input :- $p \& q(x_1, \ldots, x_n, b_1, \ldots, b_m)$.

Thus the declaration D(p & q) can be used to simulate the original declaration D.

We end this section by indicating how D(p & q) can be defined whenever there do not exist in D unit p-clauses or unit q-clauses or both. Suppose there do not exist unit p-clauses but a unit q-clause exists. We assume there is a ground p-literal $p(a_1, \ldots, a_n)$ such that D ∪ {:- $p(a_1, \ldots, a_n)$} is unsatisfiable. Then we define D' to be D ∪ {$p(a_1, \ldots, a_n)$: -}, where this new clause is placed before any other clause, and let D(p & q) to be D'(p & q). Notice that Theorem 5 is no longer valid as stated. The q-version of it, however, remains valid.

CONCLUSIONS

Logic programming is highly suitable for parallel and co-routined modes of processing. First, it supports a natural self-contained definition of parallelism in terms of a single elementary event - the match of a goal with a clause. Second, it allows freedom in the ordering of goal executions, whilst preserving the declarative semantics of programs. Third, the logical variable, through unification, automatically provides for process interfacing without execution or code overheads. Fourth, because logic clauses are terms they can be given as data to a parallel interpreter also written in logic. This provides a clear semantics of parallelism.

REFERENCES

[1] Burstall, R.M., Darlington, J.: A transformation system for developing recursive program *Journal of ACM,* Vol. 24,(1977),No 1, pp. 44-67.

[2] Clark, K., Tärnlund S. A.: A first order theory of data and programs, *Proceedings of the IFIP Congress'77,* North-Holland, Amsterdam, 1977.

[3] Colmerauer, A.: Les grammaires de metamorphose, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, Marseille, 1975.

[4] Hayes, P.J.: Computation and deduction, *Proceedings of MFCS Conference,* Czechoslovakian Academy of Sciences. 1973.

[5] Hill, R.: LUSH resolution and its completeness, DCL memo 78, Dept of AI, Edinburgh, 1974.

[6] Kowalski, R.: Logic for problem solving, DCL memo 75, Dept. of AI, Edinburgh, 1974.

[7] Kowalski, R.: Predicate Logic as a programming language, *Proceedings of the IFIP Congress' 74,* North-Holland, Amsterdam, 1974, pp. 569-574.

[8] Pereira, L.M.: Prolog - uma linguagem de programacao em lógica Divisao de Informática, Laboratório Nacional de Engenharia Civil Lisbon. 1977.

[9] Pereira, L.M., Pereira, F., Waren D.H.O.: User's guide to DECsystem-10 Prolog. Divisao de Informática, Laboratório Nacional de Engenharia Civil Lisbon. 1978.

[10] Robinson, J.A.: A machine - oriented logic based on the resolution principle, *Journal of ACM,* vol. 12 (1965), pp. 23-24.

[11] Tärnlund, S.-A.: Horn clause computability, *BIT,* vol. 17 (1977), pp. 215-226.

[12] van Emden, M.H.: Programming with resolution logic, Report Cs-75-30, Dept. of Computer Science, University of Waterloo, Canada, 1975.

[13] Warren, D.H.D.: Implementing Prolog - compiling predicate logic programs, Report no 39, Dept. of AI, Edinburgh, 1977.

[14] Warren, D.H.D., Pereira, L.M., Pereira, F.C.N.: Prolog - the language and its implementation compared with Lisp, ACM Symposium on AI and Programming Languages, Rochester, *SIGART Newsletter, SIGPLAN Notices,* August 1977.

Pereira, L.M., Monteiro, L.F.

Departamento de Informática
Universidade Nova de Lisboa

QUINTA DO CABEÇO, 1899 LISBOA
PORTUGAL