THE CONTROL FACILITIES OF IC-PROLOG

K.L. Clark and F.G. McCabe

ABSTRACT
The most significant difference between IC-PROLOG and the other PROLOG implementations is the programmer's ability to control the computation using program annotations. In this paper we exemplify the use of these annotations and give them a precise semantics. We shall see that they allow a two-pass programming methodology. In the first pass the programmer concentrates on the logic of the program, in the second pass, on the control. To make the paper self-contained, and to lay the necessary foundations for our description of the control facilities, we have included a general introduction to Horn clause logic programming.

INTRODUCTION

The paper is divided into three sections. Section 1 gives a brief history of logic programming and defines the syntax of a special class of logic programs, the Horn clause programs. We show how a given program can have several different procedural uses. In Section 2 we define an abstract interpreter for Horn clause programs. The input to the interpreter is a logic program, and a computation rule. In Section 3 we consider ways in which the computation rule can be specified. In IC-PROLOG it is partly implicitly given, by the text ordering of the program, and partly explicitly given, by clause annotations. Using them the programmer can specify a separate execution strategy for each procedural use of the program. They also enable him to indicate that certain sub-computations should be co-routined, with the resumptions and suspensions being triggered by the flow data through a shared variable.

1. SYNTAX OF HORN CLAUSE PROGRAMS

In this section we introduce the idea of sets of sentences of first order logic as programs. We consider, in particular, the special case of Horn clause logic programs.

1.1 Inference as computation: a brief history

Robinson's machine-oriented resolution inference rule for the clausal notation of predicate logic (Robinson 1965) was the first essential step along the road to viewing inference as computation. Then Green (1969) showed that a resolution theorem prover could be used to 'simulate' a computation. However it was Hayes (1973) and Kowalski (1974) who gave resolution inference an explicit procedural interpretation, Kowalski for the Horn clause subset of predicate logic and Hayes, for the most part, for logic programs comprising a set of equality statements. The final credibility was provided by the Marseilles (Roussel 1975) and Edinburgh (Warren et al. 1977) PROLOG implementations. These are essentially computationally efficient Horn clause theorem provers. The Edinburgh implementation,

which borrowed and extended the implementation techniques of the Marseilles PROLOG, is particularly impressive. It pre-compiles much of the work of a unification (the basic unit of computation of a logic program) as a sequence of machine level instructions associated with the program clauses. For list manipulation programs it compares favourably with compiled pure LISP; Warren et alia (1977) has the details. PROLOG has also been implemented in Budapest (Futo et al. 1977), Leuven (Bruynooghe 1976), London (Clark and McCabe 1979) and Waterloo (Roberts 1977).

We shall follow Kowalski and use Horn clause implications as the basic programming notation. We assume some familiarity with the concepts of unification and resolution, however we shall give a brief explanation of these ideas.

1.2 Syntax of Horn clause programs

Definitions

(1) A *Horn clause implication* is a sentence of the form

$$R(t_1, \ldots, t_n) \leftarrow A_1 \& \ldots \& A_m \quad m \geqslant 0 \qquad (1)$$

Each A_i, like $R(t_1, \ldots, t_n)$, is an atomic formula. When $m = 0$, and the antecedent of the implication is empty, we call the implication an assertion. We will generally drop the '←' when writing assertions.

(2) Something is an *atomic formula* (or *atom*) if it is of the form $R(t_1, \ldots, t_n)$ where R is an n-adic relation name (predicate) and $t_1, \ldots t_n$ are terms.

(3) A *term* is a variable, a constant, or of the form $f(t_1, \ldots, t_n)$, where f is an n-adic function name (functor) and $t_1, \ldots t_n$ are terms.

We assume a denumerable set of names comprising finite length strings over a finite alphabet which does not include the logical symbols '←' ',' '(' ')' and '&'. Whether or not a name is a functor or a predicate can be determined by context. To distinguish variables and constants we adopt the convention that names beginning with a lower case letter are variables. All other names in the context of a variable or constant are assumed to be constants. Informally, we shall use infix notation for both terms and atoms. For example, we shall write 'u *in* u.x' instead of 'in(u, . (u,x))'. We shall assume that all infix functors associate to the right, so '2.3.NIL' is syntactic sugar for '.(2,. (3,NIL))'. We will italicise infix predicate symbols.

Declarative reading. If $x_1, \ldots, x_k$ are all the variables of implication (1) we can read it as:

For all $x_1, \ldots, x_k$, $R(t_1, \ldots, t_n)$ if A_1 and A_2 ... and A_m.

Alternatively, if $y_1, \ldots, y_i$ are variables that only appear in the antecedent $A_1 \& \ldots \& A_m$, and $z_1, \ldots, z_j$ are all the other variables, we can read it as:

For all $z_1, \ldots, z_j$, $R(t_1, \ldots, t_n)$ if
there exists $y_1, \ldots, y_i$ such that A_1 and ... and A_m.

The alternative reading derives from the fact that a universally quantified implication

$$(\forall x)\ [P \leftarrow Q]$$

is logically equivalent to

$$[P \leftarrow (\exists x)Q]$$

when x does not appear in P. We use '$(\forall x)$' when we want to explicitly indicate a universal quantification of a variable x, '$(\exists x)$' for an existential quantification.

We shall say that implication (1) is about the predicate R, because R is the predicate of the consequent atom of the implication.

Definition. A *logic program for a predicate R* comprises a set of Horn clause implications about R, together with logic programs for any other predicates that appear in the antecedents of these clauses.

As in all programming notations we shall assume that some predicates are primitive, with a fixed 'system' provided program. An example would be the product predicate. Atoms that use the predicate will usually be written as $t_3 = t_1 * t_2$, t_1, t_2 and t_3 being the terms of the atom and ' . . = . . * . . ' being the infix relation name. Conceptually, we treat the system provided implementation as equivalent to the infinite set of assertions:

$$\begin{array}{l} 0 = 0 * 0 \\ 0 = 1 * 0 \\ 0 = 0 * 1 \\ 1 = 1 * 1 \\ \vdots \\ 4 = 2 * 2 \\ \vdots \end{array}$$

Of course, in practice only a finite subset of these assertions will be accessible, that subset being represented procedurally so as to make use of the hardware operations of the machine. We shall elaborate on this procedural implementation of a set of assertions in Section 3.

1.3 Example Logic Programs

Example program - 1

append(NIL,x,x)
append(u.x,y,u.z) ← append(x,y,z) (2)

is a logic program for the predicate 'append'. As the mnemonic content of this relation name indicates, the two statements of the program can be read as statements about appending list structures. We take 'NIL' to be the name of the empty list, and '.' to be the name of a list constructor such that u.x is the list x with u 'consed' onto the front. With this interpretation the assertion of the program tells us that for all x, appending the empty list onto x leaves it unchanged. The implication tells us that for all x, y, z and u, if z is the result of appending x and y then the list u.z is the result of appending u.x and y.

Recursive data structures. A term of the form $t_1 . t_2 . . . t_n$. NIL is the name of the list $[t_1, t_2, . . t_n]$. The set of all such terms is the recursive data structure implicitly introduced by this logic program. Note that the 'cases' treated by each clause are indicated by the 'patterns' of terms in the consequent atom of the clause, the 'procedure head' as Kowalski calls it. This is a common form of logic programs for relations on recursive data structures. Since we usually give the form of both the 'input' and 'output' structures it corresponds to a slight generalisation

of the case format proposed by Hoare (1973) for programs manipulating recursive data structures.

Computational use. Suppose we want to compute the concatenation of two lists, say the two unit lists [2] and [3]. We do this by asking for the (single) instance of the unary relation:

x is a concatenation of [2] and [3].

Under our assumed meaning of the 'append' predicate this relation is named by append(2.NIL,3.NIL,x), and a request for an instance of this relation is expressed by the goal clause

← append(2.NIL,3.NIL,x)

An evaluation of this clause using the program clauses is a constructive proof, using the program clauses as premises, that there is an x such that append (2.NIL,3.NIL,x). It is constructive since it will result in x being bound to '2.3.NIL'. That this term names the concatenation [2,3] of the lists [2], [3] is guaranteed by the fact that under our 'reading' of 'NIL' as the empty list and '.' as an add front element list constructor each of our logic program clauses is a true statement about the append relation for lists. Because the computation of the output binding is an inference, which preserves truth, it must denote a true instance of the relation:

x is a concatenation of [2], [3].

This truth-preserving aspect of a logic program computation is its crucial property. It enables us to understand a logic program *declaratively* or *procedurally*; to understand it as a set of statements about the relation we want to compute, or as a recipe for finding instances of the relation.

In the case of our append program its declarative reading is as a pair of universally quantified statements about the list append relation. Its procedural reading depends on its intended use. As a program for appending a pair of lists it should be read as:

to append the empty list NIL to a list y return y,
otherwise, to append a constructed list u.x to a list y first append x to y giving z, then return u.z.

Non-deterministic use. We can use the same logic program non-deterministically to split a list into front and back sublists. To split the list [2,3] we use the goal clause

← append(x,y,2.3.NIL)

since this names the relation

{⟨x,y⟩: [2,3] is the concatenation of x,y}

As we shall see, each of the substitutions

{x/NIL,y/2.3.NIL}
{x/2.NIL,y/3.NIL}
{x/2.3.NIL,y/NIL}

is a possible answer substitution. Each correctly denotes an instance of the relation. The ability to use the same set of clauses to compute both a function and its inverse, and more generally to find an instance of any relation that can be named by a goal clause, is a particular feature of logic programs.

As a program for decomposing a list into front and back sublists, the append program has the procedural reading:

To decompose the empty list NIL, return ⟨ NIL,NIL ⟩,
otherwise, to decompose a constructed list u.z
either return ⟨ NIL,u.z ⟩
or decompose z into ⟨ x,y ⟩ and return ⟨ u.x,y ⟩.

The *either, or* indicates a non-deterministic branch.

Most general answers. The goal clause:

$\leftarrow$ append(x,y,z)

is a request for any tuple of lists in the append relation. As we might expect there are an infinite number of possible answer substitutions. However, our goal clause evaluator will return answers that denote not a single instance, but an infinite set of instances. This is because the evaluator always returns 'most general answers' whenever possible. One answer is

{x/NIL,y/y,z/y}

which denotes the infinite set of list tuples

{⟨ [ ],1,1 ⟩ : 1 any list}

Another answer is

{x/u.NIL,y/y,z/u.y}

which gives the general form of the infinite set of append instances that have a unit list as first argument. Each answer substitution for this goal clause denotes not just a single instance of the relation named by the clause but an infinite subset of the relation.

As a program for generating the general form of an instance of the append relation our logic program has the procedural reading:

Either return the instance ⟨ NIL,x,x ⟩,
or find an instance ⟨ t_1, t_2, t_3 ⟩ and return ⟨ u.t_1,t_2,u.t_3 ⟩ where u is a variable not appearing in t_1,t_2,t_3.

Example program-2

When we describe the abstract interpreter in the next section we shall see that the generation and modification of answer substitutions is the data structure construction and manipulation of a logic program evaluation. Viewed as data structures, answer bindings that contain variables have a special role. This is because the answer binding x/t, where t contains variables, is implicitly modified to an answer binding x/t′ whenever any variables in t are bound by a subsequent evaluation step. This gives the effect of a data structure modification that would normally require an explicitly programmed assignment. The following logic program is an example of a program that can be used to manipulate most general answers in just this way. It is a program for the predicate 'front' which is intended to name the relation which holds between a number n and a pair of lists z,x when x comprises the first n elements of z. It makes use of the above program for 'append' and an auxiliary program for the list length relation.

front(n,x,z) $\leftarrow$length(x,n) & append(x,y,z)

length(NIL,0)
length(u.x,s(n)) $\leftarrow$length(x,n)

Let us note in passing that the program implicitly 'declares' two recursive data structures. The first is the set of lists generated from the empty list, named by 'NIL', using the list generator named by '.', which we have already come across. The other is the recursive data structure of the natural numbers, the set of objects generated from zero, named by '0', using the number generator, add 1, named by 's'. The term 's(0)' denotes the number 1, 's(s (0))' the number 2, and so on. We can, and we shall, use the normal decimal notation numerals as syntactic sugar for these term structure names.

Returning to our example program, relative to the intended interpretation of its predicates and function names $\leftarrow$front(2,A.B.C.NIL,x) names the unary relation that is true of x when it names the list [a,b] of the first two elements of the list [a,b,c]. One computational use of the program to construct the answer substitution {x/A.B.NIL} (and as we shall see in section 3 there is another quite different use) is equivalent to an evaluation of the goal clause $\leftarrow$length(2,x) to produce an answer substitution {x/u.v.NIL}, followed by an evaluation of the goal clause $\leftarrow$append(u.v.NIL,y,A.B.C.NIL) to produce the final answer substitution {x/A.B.NIL}. The first answer binding for x gives us the general form of a list of two elements, and the second answer binding is generated when the evaluation of $\leftarrow$append(u.v.NIL,y,A.B.C.NIL) 'fills-in' the u,v slots by generating the substitution {u/A,v/B}.

This two-pass style of data structure seems to be unique to logic programming. It provides the logic programmer with an elegant and powerful programming feature. (Warren et alia (1977) elaborate on this point.) It is, incidentally, an unavoidable consequence of using a resolution theorem prover as the program executor.

Example program-3

The two sets of assertions:

Jack *fathered* George	Tom *married* Mary
Tom *fathered* Bill	Bill *married* Jane
⋮	⋮
Bob *fathered* Tom	Jack *married* Susan

(3)

are a quite different kind of logic program. They are more like a data base than a program in the conventional sense. With the English language meaning of the relation names we can read them as a description of certain family relations that prevail amongst a group of people named 'Jack', 'George', etc.

Data retrieval. We 'compute' with the data-base-style program in exactly the same way as with the 'recursive' program for append. Thus a goal clause

$\leftarrow$Tom *fathered* x & x *married* Jane

is a request for an instance of the unary relation (presumably the only instance) that is satisfied by offsprings of Tom that are married to Jane. The evaluation of this goal clause will be a search over the *'father-of'* and *'married-to'* assertion sets. One result is the binding x/Bill.

2. PROCEDURAL SEMANTICS

The abstract interpreter we shall describe in this section is a resolution theorem prover. The unit of computation is a resolution inference of which the essential component is a unification. Unification is the process of finding a substitution that makes two atoms syntactically identical. Before describing the interpreter we shall give a brief introduction to the concepts of substitution and unification. For more detailed information the reader should consult Robinson (1965) or his more recent book (Robinson 1979).

2.1 Substitutions and substitution instances

Definitions

(1) A *substitution* is a set

$\theta = \{ x_1/t_1, .., x_k/t_k \}$

of variable/term pairs in which the $x_1, .., x_k$ are distinct variables. We say that x_i is bound to the term t_i.

(2) A *substitution instance* of a clause C is any clause that can be obtained from C by simultaneously replacing each of the variables bound by a substitution θ, at each of its occurrences in C, by the term to which it is bound. We use $C\theta$ to denote this substitution instance

(3) For a substitution

$\theta = \{x_1/y_1, .., x_k/y_k\}$

which binds all the variables $x_1, .. ,x_k$ of a clause C to a tuple of terms which are just k distinct variables $y_1, .. ,y_k$, the substitution instance $C\theta$ is called a *variant* of C.

(4) The *composition* $\theta_1 * \theta_2$ of two substitutions

$\theta_1 = \{x_1/t_1, .. ,x_n/t_n\}, \theta_2 = \{y_1/t'_1, .. ,y_k/t'_k\}$

is the substitution

$\theta'_1 \ \theta'_2$

where

$\theta'_1 = \{x_1/[t_1]\theta_2, .. ,x_n/[t_n]\theta_2\}$

and θ'_2 is θ_2 with any bindings for the variables $x_1, .. ,x_n$ deleted.

Examples

If C is the clause

$P(f(x,a),g(b,y)) \leftarrow Q(x,z) \ \& \ P(y,z)$

and $\theta_1 = \{x/u,y/v,z/x\}$

$\theta_2 = \{x/h(u),y/x,z/b\}$

$\theta_3 = \{u/g(b),x/h(u)\}$

then $[C]\theta_1$ is the C variant

$P(f(u,a),g(b,v)) \leftarrow Q(u,x) \ \& \ P(v,x)$

$[C]\theta_2$ is the substitution instance

$P(f(h(u),a),g(b,x)) \leftarrow Q(h(u),b) \ \& \ P(x,b)$

and $\theta_2 * \theta_3$ is the substitution

$\{x/h(g(b)),y/h(u),z/b,u/g(b)\}$.

A clause variant just 'says the same thing' using different variables. The def-

inition of a substitution is a little different from the standard one. Normally identity bindings such as x/x are excluded as the addition of an identity binding does not change the effect of a substitution. We allow identity bindings in substitutions since we want to include them in answer substitutions produced by a logic program evaluation. For us two different sets of bindings $\theta_1 \ \theta_2$ are the *same* substitution, i.e.

$\theta_1 = \theta_2$

if they are identical *sets* of bindings after the deletion of any identity bindings. Thus the identity substitution, the substitution that leaves any clause unchanged, is denoted by any set of identity bindings, including the empty set. Composition of substitutions is defined so that

$[C\theta]\theta' = [C]\theta * \theta'$.

It also has the property that composition is associative. In consequence we shall leave out the brackets in expressions such as $\theta_1 * \theta_2 * \theta_3$.

Definition. A *unifier* of two atoms A_1,A_2 is a substitution θ such that $[A_1]\theta$ and $[A_2]\theta$ are syntactically identical. It is a *most general unifier* (m.g.u.) if every other unifying substitution θ' is such that $\theta' = \theta * \theta''$ for some substitution θ''.

Example. The substitution

{w/2.z,u/2,x/NIL,y/3.NIL}

is an m.g.u. for the pair of atoms

append(2.NIL,3.NIL,w) & append(u.x,y,u.z).

When applied to each atom it produces the substitution instance

append(2.NIL,3.NIL,2.z).

An algorithm which tests whether or not two atoms are unifiable, and which returns an m.g.u. if they are, is a unification algorithm. Robinson (1965) gives such an algorithm and proves it correct. For us unification does the work of data structure component selection (the bindings u/2, x/NIL in the above example), and unification and composition does the work of data structure construction (the binding w/2.z composed with some binding for z). Most of the pairs of atoms that must be unified during a logic program evaluation fall within the confines of a special case for which a much faster, modified form of Robinson's algorithm can be used.

The two atoms never have variables in common and it is nearly always the case that one of the atoms only has a single occurrence of each of its variables. For this special case, the expensive *occur check* of Robinson's algorithm is not needed. This is the check, which must normally be performed each time a variable x is matched against a non-variable t, that x does not occur in t. If it does then the attempted unification fails. However for the special case we have described an occur failure *cannot* arise. For this reason most PROLOG implementations never perform the occur check, and in IC-PROLOG in must be explicitly requested.

2.2 An abstract interpreter

Any resolution theorem prover can be used as a Horn clause program executor. However the inference system which is most obviously computational is LUSH resolution (Hill 1974). This is the inference system that Kowalski describes when he talks about the procedural interpretation of predicate logic (Kowalski 1974).

The LUSH inference rule defines a search space of alternative derivations. We prefer to think of these alternative derivations as alternative paths of a non-deterministic evaluation of a 'call' of the logic program given by some *goal clause*

$$\leftarrow B_1 \& \ldots \& B_n$$

Here, $B_1, \ldots, B_n$ are all atomic formulae, and, if $x_1, \ldots, x_k$ are the variables of the clause, it is a 'call' for the computation of a substitution

$$\theta = \{x_1/e_1, x_2/e_2, \ldots, x_k/e_k\}$$

such that

$$\langle e_1, e_2, \ldots, e_k \rangle$$

denotes an instance, or set of instances, of the relation 'named' by the conjunction $B_1 \& \ldots \& B_n$ for any 'reading' of the program as a set of true statements. If there are no variables in the goal clause it is a request for the confirmation that for any such reading $B_1 \& \ldots \& B_n$ is a true statement.

The trace of some evaluation path of the computation is given by a sequence of goal clauses

$$C_1, C_2, \ldots, C_n, \ldots$$

each one derived from the proceeding one by the following evaluation step. We shall follow Kowalski (1974) and refer to the program clauses as procedures. The consequent atoms of the clauses are then procedure heads and the antecedent atoms are procedure calls. We shall also refer to the atoms of the goal clause as procedure calls.

Evaluation step. Suppose the current goal comprises the conjunction of procedure calls

$$\leftarrow B_1 \& \ldots \& B_k$$

An evaluation step is the execution of one of these calls. Which one it should be is determined by a *computation rule*, a rule which for any conjunction of procedure calls encountered during the evaluation uniquely determines which call is to be executed first. For now we shall not concern ourselves with how the computation rule is specified or applied. We shall simply assume that there is an effective rule that will select a single call from the goal clause. Any such rule is a possible computation rule.

Assume that the selected call is B_i, and that this is the atom $R(t_1, \ldots, t_n)$. An attempt is made to execute this call using a procedure

$$R(t'_1, \ldots, t'_n) \leftarrow A_1 \& \ldots \& A_m$$

which is a variant of one of the program clauses for R that does not contain variables appearing in the goal clause. If there is more than one program clause for R the choosing of the procedure is a non-deterministic step in the evaluation, each alternative procedure giving us an alternative branching of the evaluation.

We try to unify $R(t_1, \ldots, t_n)$ with $R(t'_1, \ldots, t'_n)$.

If they do not unify this branching of the evaluation path terminates with *fail.*

If they do unify, with most general unifier θ, this branching of the evaluation path leads to a new conjunction of procedure calls, a new goal clause, which is the resolvent

$$\leftarrow [B_1 \& \ldots \& B_{i-1} \& A_1 \& \ldots \& A_m \& B_{i+1} \& \ldots \& B_k]\theta$$

Note that the substitution instance of the conjunction of procedure calls in the body of the program procedure have replaced B_i in the new goal clause. But also note that the unifying substitution is applied to all of the procedure calls of the new goal clause. This means that any output binding of the substitution, any binding of a variable x that was in the executed call B_i, is immediately communicated to any other call B_j in which x appears. As we shall see later, we can use the idea of data flow through the shared variables of procedure calls to specify and implement data flow computation rules.

Finally, if there are no program procedures for R, we consider that there is just one next step for the evaluation path that has produced the goal clause $\leftarrow B_1 \& \ldots \& B_k$ which gives an immediate *fail.*

Successful evaluation. An evaluation path terminates with *success* when the empty goal clause is generated. This happens when the last step is the execution of the single call of a goal clause $\leftarrow B$ using an assertion procedure $B' \leftarrow$.

Computed answer substitution. Suppose $\theta_1, \ldots \theta_n$ is the sequence of unifying substitutions of a successful evaluation, θ_1 being the substitution of the first step, θ_n that of the last step. Let

$$\theta = \theta_1 * \theta_2 * \ldots * \theta_n$$

be the composition of these unifying substitions. The subset of θ that gives bindings for the variables of the initial goal clause C, augmented with the identity substitution x/x for any variable of C not bound by θ, is the answer given by that successful evaluation. If there are no variables in C, the answer is *true*.

Procedural vs. Declarative semantics. The above definition effectively provides us with a *procedural semantics* for answer substitutions. It characterises answer substitutions in terms of a mechanism for computing them. In Van Emden and Kowalski (1976) and Clark (1979) a quite different declarative semantics is given. In the declarative semantics answer substitutions are characterised in terms of models of the program clauses and the corresponding denotations of the goal clauses. In what amounts to a completeness proof for LUSH resolution, first given in Hill (1974), it can be shown that for *any* computation rule the procedural and the declarative semantics characterise essentially the same set of answer substitutions. This independence of the computation rule is a great boon. It enables the logic programmer to select a computation rule purely on the grounds of computational efficiency. We shall return to this point later.

Evaluation trees A logic program P, a goal clause C, and a computation rule R, together define an evaluation tree of all the possible evaluation paths for C. The form of the tree is as depicted in Fig. 1. Each interior node in the tree is a clause derived from its parent by an evaluation step. Each of the children of a node are the results of attempted executions of the selected procedure call of that node.

A fail node offspring records a failed unification. A success leaf node marks the end of each successful evaluation path. Some of the evaluation paths may be infinite.

Figs. 2 and 3 are evaluation trees for two of the calls to the logic programs of the preceding section. The computation rule used is 'select the leftmost call'.

The very first step of the successful evaluation path of Fig. 2 requires the unific-

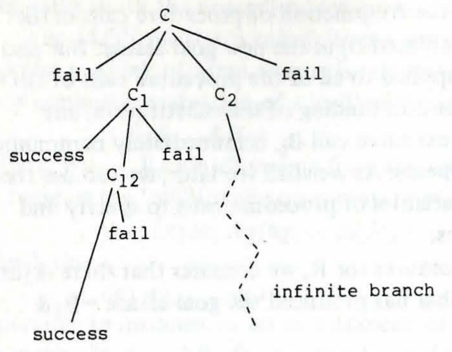

Figure 1. An evaluation tree

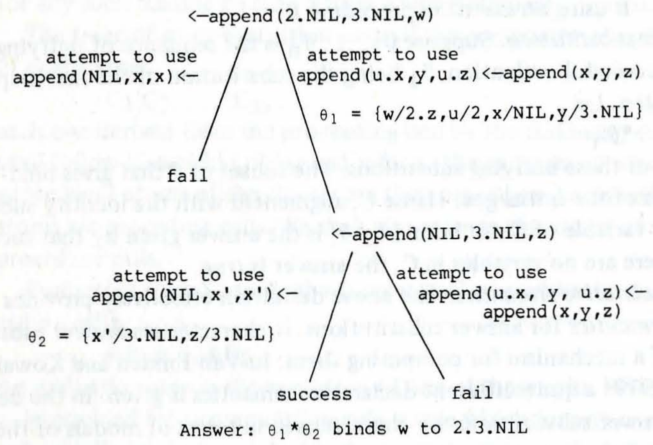

Figure 2. Evaluation of append

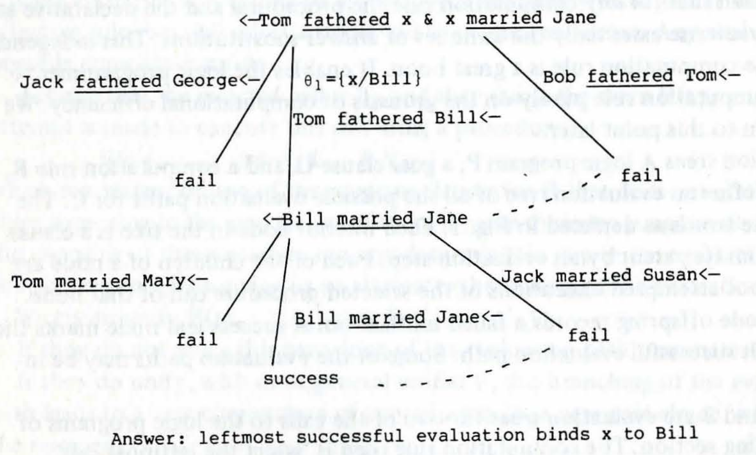

Figure 3. Evaluation of fathered

ation of append(2.NIL,3.NIL,w) and append(u.x,y,u.z). The m.g.u. binds u to 2 and x to NIL, that is it decomposes the 'input' 2.NIL into its head and tail sublists. As we mentioned above, the selection of components from data structures in accordance with a given pattern, in this case the pattern 'u.x', is one of the major roles of unification. The other major role is the production of partial approximations, templates, for the output. This is illustrated by the binding 2.z for w. This binding gives us a first approximation to the answer substitution w/2.3.NIL which is the final output of the evaluation path. It tells us that any answer for the path will bind w to 2.'something'. This first approximation is a partial result that could now be accessed by another procedure call in which w appeared. We shall return to this idea of 'data flow' activation of procedure calls in Section 3.

Both evaluation trees of Figs.2 and 3 are finite. Hence a back-tracking search, which is a walk over an evaluation tree as depicted in Fig.4, can be used to search for a successful evaluation.

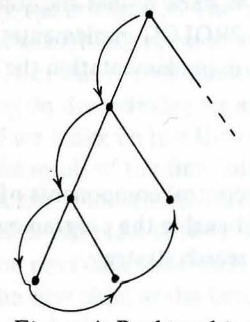

Figure 4. Backtracking

Such a walk over the evaluation tree of Fig.3 amounts to a double loop search over the *'fathered'* and *'married'* assertion sets. A similar walk over the evaluation tree of Fig.2. does not involve significant backtracking. Each failure node is an immediate offspring of the one and only successful evaluation path. Evaluation trees such as this record an essentially deterministic computation. At each step there is only one clause whose consequent will unify with the selected call of the goal clause.

A genuinely non-deterministic use of the append program is depicted in Fig.5. It is the evaluation tree for the goal clause ←append(x,y,2.3.NIL). Each of the answer substitutions gives one of the possible decompositions of the list [2,3] into front and back sublists.

A note on implementation. A partially constructed branch of the evaluation tree can be represented as a stack of activation records. Extending this branch to derive a new goal clause is then a stack push operation which adds an activation record to the stack. This contains a pointer to the program clause that was used, and a pointer to a binding environment containing the bindings for all the new variables introduced by that clause. This stack representation of the current and preceeding goal clauses, which is essentially the Boyer and Moore (1972) structure sharing representation of resolvents, corresponds to the conventional stack implementation of recursion. It also carries an added bonus. Backtracking to try an alternative clause for the previous

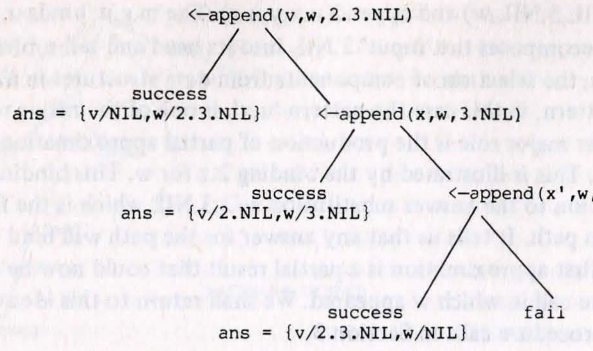

Figure 5. Non-deterministic append evaluation

evaluation step is little more than a stack pop operation. So a backtracking search for a successful evaluation is an interleaved sequence of stack pushes and stack pops. Because backtracking is so easy to implement, all the PROLOG implementations use backtracking as the search strategy. For more details on implementation the reader should consult Warren (1977).

3. CONTROLLING THE EVALUATION

The search strategy and the computation rule are the control components of a logic program. A logic programming implementation should enable the programmer to specify, at least partly, the computation rule and the search strategy.

3.1 Search Control

As we remarked above, for reasons of efficient implementation all PROLOG systems use backtracking as the search strategy. This leaves the programmer with limited search control. He can only control the order in which the program clauses are tried. He specifies this implicitly by the before-after order in which the clauses are written. Although a fixed try order is not adequate for 'problem solving' uses of the evaluator (Kowalski 1979b), which in any case would usually need some sort of breadth-first search strategy, a judicious choice of the try order is all that is required for nearly all the LISP-style, and even the data-base-style, logic programs. A programmer quickly learns how to exploit the backtracking, and to appreciate it as a powerful control primitive of logic programs.

Clause indexing. As an additional search facility we could index the clauses. The Edinburgh PROLOG automatically indexes all the program clauses on the constant or top-level functor of the first argument. This cuts out much of the shallow backtracking of the search for an applicable clause. Alternatively, one could allow the programmer to specify which clauses should be indexed, and the degree of indexing required.

IC-PROLOG gives the programmer just such indexing control. For any predicate the programmer can request indexing of all the clauses for the predicate on any or all of the argument positions. For each indexed argument the index groups together all the clauses that have the same constant, the same top-level functor, or a variable

in that argument position. The indexing is particularly useful for data-base-style programs. By indexing the set of assertions that provide the extensional definition of a data base relation we can access directly the subset, even the individual assertion, that can possibly match the call. Indexing can also be useful for the LISP-style programs. It can make the try order irrelevant for most calls, the index effectively providing us with a conditional branch to the appropriate clause.

Indexing for data-base style programs. To see how indexing can affect the evaluation of a goal we look again at the example program-3 from Section 1.3. In particular, we see how indexing affects the evaluation of

← Tom *fathered* x & x *married* Jane (1)

If we just index on the first argument of the *fathered* relation, then for the call 'Tom *fathered* x' indexing 'extracts' from the *fathered* relation all those instances where 'Tom' is mentioned in the first argument. In other words, indexing on the first argument will cause only those assertions mentioning children of Tom to be considered by the interpreter. The backtrack search mechanism will then iterate through this set of assertions, trying to evaluate the second call: 't *married* Jane' for each possible 't'.

For this second call we can have three different kinds of search behaviour depending on the indexing we apply to the *married* relation (assuming we do some indexing). If we index on just the first argument, then the index access will be controlled by the result of the first call. Indexing will extract from the *married* relation those instances where t is married to someone. The interpreter will then search through these instances to see if the someone is in fact 'Jane'. If the backtrack search fails the next candidate child of Tom is used, a new access to the index is made, using the new child as the key, and so on until a solution is found. The behaviour we get by indexing on the first argument of each relation is that of the iteration:

```
for each child x of Tom
    for each y married to x
        if y = Jane, exit with x.
```

If instead, we index on the second argument of the *married* relation then access via the index will return as candidate assertions only those where 'some-one is married to Jane'. In this case the accesses to the indexes are independent (and could be done in parallel). The backtracking evaluation of the two call goal clause effectively computes the intersection of the children of Tom and spouses of Jane. The behaviour is equivalent to:

```
for each child x of Tom
    for each spouse y of Jane
        if x = y, exit with x.
```

Finally, suppose we have indexed on *both* arguments of the *married* relation. In this case index access for the second call amounts to a look-up of the married relation to confirm (or otherwise) that the child of 'Tom' returned by the first call is indeed married to Jane. The behaviour of the goal clause evaluation is that of the iteration:

```
for each child x of Tom
    if x is married to Jane, exit with x.
```

Compare these search behaviours with the case when no indexing is done. The whole of the *fathered* relation is sequentially searched until an occurrence of 'Tom *fathered* t' is found. Each time one is found the whole of the *married* relation is sequentially searched to see if 't is *married* to Jane'. The behaviour is that of:

```
for each fathered pair ⟨x,y⟩
    if x = Tom,
        for each married pair ⟨y′,z⟩
            if z = Jane & y = y′, exit with y.
```

For large relations this is an expensive method of searching, which indexing can eliminate. As we have seen, indexing can be used to get some interesting search behaviours from data-base style programs; in particular, an unacceptable sequential search can be transformed into an acceptable direct look-up.

Indexing for LISP-style programs. Consider the length program (from example program-2 in Section 1.3). If we index on both arguments, we get a case analysis on both the list structure (NIL vs. u.v) *and* the size of the length; ie. whether the length of the list is 0, or greater than 0 (0 vs. s(x)). This means that whether we use the program to find the length of a list, or to find a list of a given length, the abstract interpreter always only tries to use the correct clause. In this case, the access via the index is a conditional branch to the appropriate clause.

The notion of indexing on the top-level function names and constants, generalises to indexing on lower-level terms as well. This would allow finer discrimination, which would be useful for certain LISP-style programs, though rather less useful for common data-base style programs.

Arithmetic Data Base. We can now be more specific about the nature of the built-in arithmetic relations discussed in Section 1.2. The reader will recall that the implementation of the addition relation was stated as being equivalent to a data base of assertions, each of which described a single arithmetic sum. We will now say a little about the representation of these assertions. We have already seen how indexing can turn an apparently inefficient sequential search through a relation, into a fast look-up into the relation. The implementation of these arithmetic predicate is in fact equivalent to a fully indexed set of assertions.

This means, that when the interpreter is evaluating an addition procedure call, instead of searching through all the assertions in the addition relation trying to find the correct ones, only those portions of the relation that are directly relevant to the call are considered. For example, in evaluating the goal:

$\leftarrow x = 2 + 3,$

only the instance '5 = 2 + 3' will be looked at by the interpreter, from the whole addition relation. Normal evaluation, using this assertion, will cause the answer binding x/5 to be made. For 'normal' computation the evaluation of arithmetic expressions will be straight-forward. Each expression will be evaluated just as in any ordinary programming language, there is no searching 'for the correct next step'.

Of course we do not *actually* have this large set of assertions stored somewhere, nor do we *actually* use the indexing mechanism to sort the relation. What in fact happens is the interpreter uses the underlying hardware to do the additions etc., the 'indexing' is translated into a case analysis of the call: depending on the actual usage of the arithmetic relation, the hardware will be requested to do additions,

subtractions, multiplications and so on.

We do not sacrifice any flexibility however, the 'arithmetic data base' still reflects faithfully the declarative semantics of an indexed set of assertions. For example, in the evaluation of the call ←5 = x+y the instances of the addition relation considered by the interpreter will be:

```
5 = 5+0 with answers for x and y: {x/5, y/0}
5 = 4+1                           {x/4, y/1}
   ⋮                                  ⋮
5 = 0+5                           {x/0, y/5}
```

The interpreter will search through this set of assertions in the normal way, finding values for x and y, and check that they satisfy the constraints of the other call(s) in the goal. Because of the indexing, no other instances from the addition relation will be considered. This is an example of a non-deterministic use of the arithmetic data-base. To our knowledge no other implemented language allows the non-deterministic use of arithmetic primitives.

We can use this flexibility in the arithmetic data base to write some very elegant programs, for some examples of this, and for a more detailed explanation of the way indexes are requested, constructed and used, the reader is referred to the IC-PROLOG reference manual (Clark and McCabe 1979).

3.2 Computation Rule Control

The computation rule is an area in which we can more usefully exploit control flexibility. In this, IC-PROLOG differs significantly from the Marseilles and Edinburgh PROLOG implementations. Both of these use a fixed, leftmost call computation rule. As with the use of a fixed search strategy, this means that programmer control is rather limited. By the left-to-right ordering of the procedure calls of each clause he does control the order which they will be selected. However this given order is then fixed for each use of the clause, and the sequence of calls are always evaluated in a strictly sequential fashion. We shall see that this fixed order, strictly sequential execution has considerable drawbacks, that it does not enable us to fully exploit the potential of logic programming. On the credit side, sequential execution, being the norm for programming languages, is a control concept that is easy to understand and use, and the leftmost call computation rule is extremely easy to implement. If we are to allow more elaborate programmer specified computation rules, which is possible in IC-PROLOG, we must make sure that they embody equally intuitive control concepts. We must also make sure that they impose the minimum run-time overhead.

Drawbacks of a fixed order strictly sequential execution. To see what sort of control facilities might prove useful let us consider the disadvantages of the leftmost call rule. There are two main disadvantages. The first is a consequence of the fact that the textual left to right order of the procedure calls of a clause defines the execution order for *every* use of the clause. This means that, despite the theoretical possibility of using a logic program to find any unspecified arguments of a relation, it is usually only computationally viable for one input-output pattern. The second disadvantage is a consequence of the depth first evaluation, the fact that each call, once selected, is completely evaluated before control moves to an adjacent call.

To illustrate these drawbacks let us consider the procedure:

R(x,z) ←P(x,y) & Q(y,z).

This tells us that x and z are in the R relation if there is some y which 'connects' x and z via the P and Q relations.

With the given ordering of the procedure calls we can expect that the use of this procedure is computationally viable for calls in which both x and z are given, or for calls in which x is given and a corresponding z is to be computed. However, for calls of the form ←R(x,t), in which a non-variable term t supplies a value for z and a corresponding x is to be found, the given order of the calls is far from suitable. It will result in the call ←P(x,y), rather than the more suitable ←Q(y,t), being used to generate the intermediary value for y. Thus, the candidate values for y will be produced by unconstrained 'guesses', while the computation of a y value that satisfies Q(y,t) might be essentially deterministic. To fully exploit the input-output flexibility of logic programs we should at least make the order of execution of the calls of a procedure dependent upon the input-output of the invoking call.

However, even for calls of the form ←R(t,z), for which the given order of the calls is satisfactory, a strictly sequential execution can be far from optimal. Let us suppose that the evaluation for the derived call ←P(t,y) is truly non-deterministic, that there are several values for y that satisfy P(t,y). Let us further suppose that each of these values is denoted by a term t′ which is approximated to by a sequence of partial results, $t_1, t_2, \ldots, t_n$ that are produced at various stages of the evaluation of P(t,y). We observed this partial result phenomena in Fig.2. We saw that even the first step in the evaluation of the call ←append(2.NIL,3.NIL,z) produces the partial result 2.x as the first approximation of the final value 2.3.NIL. In these circumstances, which are not uncommon, the best execution strategy is to co-routine between the P and Q evaluations. The Q evaluation is entered, and then re-entered, whenever a new partial result is generated, and the P evaluation is resumed whenever a new partial result is required. If at any stage the latest partial result that is communicated to Q cannot denote any value that satisfies the Q relation, we shall benefit from an early failure. The overall evaluation can then backtrack to find an alternative partial result for y, and we have saved what might have been a lengthy computation of the complete result for the abandoned ←P(t,y) evaluation.

3.3 Control Annotation

In IC–PROLOG the medium for the specification of the computation rule is program annotation. With quite simple syntactic additions to the basic Horn clause syntax the programmer can define a computation rule which changes the try order of the calls of program procedure in accordance with its input-output use, and he can specify a computation rule that will result in a co-routining interaction between certain sub-computations that is triggered by the flow of data through shared variables.

Control alternatives. In order to allow for input-output dependent use of a procedure the programmer can give for each procedure an explicit list of control alternatives. This is a list of annotated program clauses

$[C_1, C_2, \ldots, C_k]$, $k > 0$,

each of which, ignoring the annotations and the order of the antecedent atoms, is an exact copy of a single clause

$B \leftarrow A_1 \& \ldots \& A_n$.

Declaratively, the list of control alternatives is equivalent to this single implication. Procedurally, it also represents just a single alternative for the non-deterministic selection of a program clause to respond to the procedure call ←B′. If B′ unifies with B at most one of the clauses in the list may be used for the evaluation step. Which one is used is determined by the '^', '?' annotations attached to terms in the procedure head of each copy of the clause.

These annotations specify extra constraints that must be satisfied by the unification of B′ and B before that particular clause can be used. Intuitively, the annotated term 't?' tells us that t must be used as an input template. More formally it signals the requirement that each of its variables must have been matched against a non-variable, or an *input variable of the* call. If t does not contain variables, it signals the requirement that t must have been matched against an identical variable free term. We must wait until the discussion of the data flow co-routining before we can define input variable. Roughly speaking, it is a variable through which the call will eventually 'receive' data, in the form of a non-variable binding, generated by some other procedure call.

The annotation '^' signals the opposite use. A term t, annotated t^, must be used as an output template. More formally, in the unification it must have been matched against an *output variable of the* call. An output variable is any variable that is not an input variable.

Example-1. The list of control alternatives

[Grandparent (x?,z) ← Parent(x,y) & Parent(y,z),
 Grandparent(x^,z?) ← Parent(y,z) & Parent(x,y)]

for the clause defining grandparent would be used to indicate that the intermediary parent y is to be found by looking up the children of the grandparent when the grandparent is given, and by looking up the parents of the grandchild when the grandchild is given and the grandparent is to be found.

Example-2. Suppose we have a data-base of assertions for two binary predicates R and Q and that we have requested indexing on the first argument of R and the second argument of Q. We should use the control list:

[P(x?,y) ← R(x,y) & Q(x,y) ,
 P(x^,y?) ← Q(x,y) & R(x,y)]

as a program that computes the intersection of Q and R.

Selection rule. When a clause appears as one of a list of control alternatives then it can be selected only if all the match constraints signalled by the procedure head '?', '^' annotations are satisfied. The first clause in the list with its match constraints satisfied is the one which is used. If there are none, then the evaluation terminates with a control error.

In the above example-1 there is no clause copy for the case when both x and z are to be found. The attempt to use this set of control alternatives for this case will result in a control error. Thus the I/O annotation also gives the programmer a run-time check that no unexpected input-output use will arise.

Data flow co-routining. Having found a control alternative whose input-output conditions are satisfied control 'enters' the body of the selected procedure. In the absence of control annotations in the procedure body the sequence $A_1\& \ldots \&A_n$ of calls will be executed in a strictly left-to-right order. That is, for each i the call A_i is selected only when the evaluation of $A_1\& \ldots \&A_{i-1}$ is complete.

To relax this strict left-to-right evaluation, to enable the evaluation of the call A_i to be commenced prematurely, we can attach a '?' or '^' annotation to one of the variables of A_i. The effect of this is to make the annotated variable, v, a *data channel* for A_i. If the '?' annotation is used A_i is an *eager consumer* of the data that will be sent down the channel by the evaluation of $A_1\& \ldots \&A_{i-1}$. If the '^' annotation is used then A_i is a *lazy producer* of the data that it sends down the channel.

A_i as an eager consumer. If the channel variable v does not appear in at least one of the calls $A_1, \ldots ,A_{i-1}$ the annotation 'v?' on the occurrence of v in A_i has no effect. If it does appear, the effect is to make the evaluation of A_i a special process that can co-routine with the evaluation of $A_1\& \ldots \&A_{i-1}$.

During the evaluation of $A_1\& \ldots \&A_{i-1}$ we can expect that the final output binding for v is approximated to by a sequence

$$t_1, t_2, \ldots, t_n$$

of non-variable terms. Here, t_1 is the first partial result. It is generated when the composition of the unifying substitutions being generated by the evaluation of $A_1\& \ldots\&A_{i-1}$ first gives a non-variable binding for v. Thereafter, t_{j+1} is the term t_j with one or more of its variables replaced by non-variables. It is generated when the composition of the unifying substitutions induces such a change in the previous binding for v.

A *data call* of A_i occurs whenever one of these partial results t_j is generated. A *data return* from A_i occurs when the computation of the A_i is completed, or, more usually, when it generates the 'need' for the next partial result t_{j+1}. A_i generates the need for t_{j+1} when the next step in its evaluation would result in one or more of the variables of t_j being bound to non-variable terms. That is, when it is about to 'guess' the next partial result. On the return from A_i the evaluation of $A_1\& \ldots \&A_{i-1}$ is resumed at the point of the last data call. Similarly the next data call of A_i, which occurs when the evaluation of $A_1\& \ldots \&A_{i-1}$ generates the required t_{j+1}, will resume the required evaluation of A_i at its last suspension point.

The effect of the sequence of data calls and returns is to co-routine the evaluation of A_i with the evaluation of $A_1\& \ldots \&A_{i-1}$. Each slice of the evaluation of $A_1\& \ldots \&A_{i-1}$ is the generation of a new partial output binding for v, and each slice of the evaluation of A_i is the consumption of this partial output. We call A_i an *eager consumer* because it 'runs' as soon as the next partial output is generated.

The co-routining interaction terminates if either the evaluation of $A_1\& \ldots \&A_{i-1}$

or that of A_i is successfully completed. If it is the former default sequential execution takes over but with the evaluation of A_i picked up at its last suspension point. If it is A_i, then, following what is the last data return from A_i, the evaluation of $A_1\& \ldots \&A_{i-1}$ runs to completion, irrespective of whether or not it generates any more partial results for v. On completion, the default sequential execution continues with A_{i+1}.

A_i as a lazy producer. If the occurrence of the variable v in A_i is annotated v much the same co-routining interaction takes place. The significant difference is that this time it is the evaluation of A_i that generates the sequence of partial output bindings for v. Correspondingly, the call and return triggers are reversed.

This time a data call of A_i occurs when the evaluation of $A_1\& \ldots \&A_{i-1}$ needs the next partial result, i.e. just before the evaluation step that would result in a variable of the current partial result being bound to a non-variable. A data return from A_i occurs as soon as the next partial result is generated, i.e. just after the evaluation step that so binds a variable of the current partial result. Since we return from A_i as soon as it produces this next level of approximation, we call it a lazy producer.

Nested co-routining. In the above description of the data flow co-routining we have deliberately imposed no constraint on the 'internal' evaluations of the procedure call segment $\leftarrow A_1\& \ldots \&A_{i-1}$ and the procedure call A_j, $j<i$, is also data flow annotated with a '?' or a ' '. (The current IC-implementation allows only one eager consumer/lazy producer for each variable. For lazy producers this is a reasonable restriction. For eager consumers it slightly cramps one's style.) This will cause a co-routining interaction between the evaluation of $\leftarrow A_1\& \ldots \&A_{j-1}$ and $\leftarrow A_j$, within the evaluation of $\leftarrow A_1\& \ldots \&A_j\& \ldots \&A_{i-1}$ .. Other, 'internal' co-routining can take place as a result of data flow annotations in the procedure bodies of the clauses used in the evaluation of $\leftarrow A_1\& \ldots \&A_{i-1}$ and in the evaluation of $\leftarrow A_i$. Each of these low-level co-routining interactions takes place in accordance with the above procedural semantics.

Lazy evaluation mode. By systematically making all the producers of intermediary results lazy producers the programmer can signal a lazy evaluation mode for his logic program, causing it to compute answer bindings in a lazy LISP fashion (Henderson & Morris 1976, Friedman & Wise 1976). Thus, where he would normally use the annotated clause

R(x?,w^) ←R(x,y) & Q(y,z) & T(z,w)

for a strictly sequential computation of the output w which corresponds to some input x, for a lazy evaluation he can use

R(x?,w^) ←T(z,w) & Q(y,z^) & R(x,y^).

Similarly, if the goal clause

←P(t,x) & R(x,w)

would be used for the sequential construction of x and w bindings, it is rewritten as

←R(x,w) & P(t,x^)

to signal the lazy construction.

Eager evaluation mode. An alternative, eager evaluation mode, can be requested by making all the consumers of the intermediary results eager consumers. For this, one would write the above program clause as

$R(x?,w\hat{}) \leftarrow R(x,y) \& Q(y?,z) \& T(z?,w)$

and the above goal clause as

$\leftarrow P(t,x) \& R(x?,w).$

In execution, the eager mode program contains much the same data transmissions and requests as the lazy mode program. It just starts by doing some work on generating the first intermediary result, whereas the lazy mode starts by doing some work generating the final result.

Mixed Mode. The fact that lazy or eager interaction is explicitly requested by an annotation means that they can be mixed. For example, one can signal a lazy construction with an eager test, by the annotated clause

$P(x?,z\hat{}) \leftarrow Q(y,z) \& R(x,y\hat{}) \& T(z?)$

This ability to mix the execution modes offers a control dimension that we have not yet fully explored. To our knowledge only Schwarz (1977), with his annotated recursion equations, offers a programming notation with the same mixed mode facility. However, that is for an essentially deterministic programming language. As we shall see in our later examples, the conjunction of data flow co-routining and back-tracking search is the really novel control mix offered by IC-PROLOG.

Input variables. We can now define input variable. To make the definition precise we need the concepts of data-transmission and data request. A *data transmission* is a call of an eager consumer or an early return from a lazy producer. A *data request* is a call of a lazy producer or an early return from an eager consumer.

Definition

Let v be a data channel for a call A_i.

Case (1) v is as yet unbound and A_i is a lazy producer of v. v is an input variable of any call selected before the first data request to A_i.

Case (2) v is currently bound to a term t which is one of the partial approximations to the final output binding and A_i is an eager consumer or lazy producer of v. Any variable in t is an input variable for any call selected after the data transmission that communicated t but before the data request for the next partial approximation.

Informally, input variables are those variables that can only be bound to a non-variable by an evaluation step involving some other call.

Delaying the data transfer as described above, a data transfer occurs immediately after the evaluation step that generates the next partial result. However, it might be that this evaluation step has made use of one of several clauses that would have matched the call, a situation which arises when the 'pattern' of terms in the procedure heads of the alternative clauses does not, on its own, determine a single clause which must be used. In this circumstance the appropriate clause is usually selected by one or more test calls that begin each procedure body. Any output that the invoking unification may have produced is really conditional upon the successful execution of the test calls. We should therefore delay any data transfer until after the evaluation of these calls.

The annotation to request such a delay is a ':' used instead of the '&' which immediately follows the test calls. Thus, if we write

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

instead of

$P(f(u),g(v)) \leftarrow T(u) \& R(u,v)$

this will delay any data transfers that would normally immediately follow the evaluation step that uses this clause until after the evaluation of the call $\leftarrow T(u)$ is completed. We call the ':' the clause bar.

More formally, the clause bar can be inserted at any point in the sequence of procedure calls of a program clause, even after the last call. The effect of the bar in a clause

$B \leftarrow A_1 \& \ldots \& A_i : A_{i+1} \& \ldots \& A_n$

is to delay any data transfers out of this procedure until the evaluation of the call segment $A_1 \& \ldots \& A_i$ is complete. A data transfer *out of* the procedure is either a data return from a producer that is an ancestor of $A_1 \& \ldots \& A_n$ (i.e. the use of the clause was an evaluation step of the producer), or it is a data call of an eager consumer that is a sibling of some ancestor of $A_1 \& \ldots \& A_n$. The bar has no delaying effect on suspensions of the procedure evaluation due to data requests.

3.4 Using the Control

For further information on the syntax, and certain restrictions on the use of the producer/consumer annotations, we refer the reader to the complete IC-PROLOG reference manual. We shall content ourselves with two examples of logic programs that use the annotations, which should help to clarify the ideas.

Example-1. For our first example we shall look again at the clauses that describe the relation front(n,1,1′) which holds when 1 comprises the first n elements of the list 1. We made use of these in example program-2, in Section 1, to give an example of the use of answer substitutions that contain variables. This time we shall annotate them so that we get a co-routining construction of the list of the front n elements that simulates the counting down transfer of elements of 1 to 1′.

Leaving aside considerations of their computational use we can simply write down the implications of the program in any order, and with any ordering of the antecedent atoms. In general, this should be the first stage of writing a logic program; we should first concentrate on the logic component of the program, using and reading the clauses as first order statements about the input-output relation we want to compute. As a second stage we can concern ourselves with their computational use, with the control component. We do this by choosing a suitable ordering of the clauses and their antecedent atoms and by adding the appropriate annotation for the use we have in mind. At this stage we might conceivably revise the logic component, changing it to a more 'pragmatic' description should a viable computational use be impossible to achieve (we shall have to do this in the next example program). This two-stage program writing methodology, which is another novel feature of logic programming, is a consequence of the independence of the logic and the control.

Let us take the clauses

front(n,z,x) ←append(x,y,z) & length(x,n)

length(NIL,0)
length(u.x,s(n)) ←length(x,n)

append(NIL,x,x)
append(u.x,y,u.z) ←append(x,y,z)

as our first order description of the front relation.

We now address the computational use of these first order implications to strip off the first n elements of a list in answer to a call of the form ←front(n,1,x) in which x is the only variable. If the clauses were to be used as they stand the back-tracking search strategy will result in pairs of candidate decompositions of the input list 1 being generated by the non-deterministic evaluation of ←append(x,y,1). The first pair will be x = NIL, y = 1, the second x = a_1.NIL, y = 1′ where a_1 is the first element of 1 and 1′ its tail, and so on. Each of these candidate decompositions will be checked in turn by a complete evaluation of ←length(x,n). When this fails, because the length of x is not yet n, a back-track leads to a costly complete re-computation of the next candidate decomposition. This is not a viable computational use.

What we should be doing is counting down on the length parameter n as the recursive decomposition of 1 adds each new element to the partial output binding for x. When we have counted down to zero this partial output, which will be of the form $a_1.a_2 \ldots a_n.x'$, can be completed by a last step of the decomposition that binds x′ to NIL.

Exactly this behaviour will result if we use the annotated clause

front(n?,z?x^) ←append(x,y,z) & length(x?,n).

This makes the length call an eager consumer of the partial output bindings for the front sublist x, and restricts the use of this control regime to calls of the appropriate form.

The data calls of length(x?,n) actually serve two purposes. The sequence of the calls is best viewed as a sequence of pairs of calls. The first call of each pair follows each use of append(NIL,x,x)← in the non-deterministic evaluation of the append call. Following this data call the next step in the suspended length evaluation can only succeed if the length of the front sublist has reached n. This call is the 'loop' exit test. If this step fails only the last step in the append evaluation is undone. This is immediately followed by the use of the recursive append clause, and the second data call of length. On this second call only the recursive clause for length can be used and its use effectively decrements the length count by 1.

Example-2. Let us consider the problem of describing a solution to the eight queens chess problem using only Horn clause sentences.

First, we have to find some way of naming candidate solutions, configurations of eight queens distributed on a chess board. We can simplify our task if we only consider configurations in which each row and columns contains only one queen, a constraint that we know must be satisfied by any solution to the problem. A list of the numbers 1 to 8, in some arbitrary order, is a name for such a configuration if we take the ith number on the list as the column number of the queen in the ith row; with this naming convention, the set of permutations of the numbers 1 - 8 names the set of all the board configurations we are considering. A solution to the problem is one of these permutations that is safe, that is one which represents a configuration in which no queen is on a diagonal with any other.

The Horn clause:

Queen-sol(x) ←Perm(1.2 . . . . 8.NIL,x) & Safe(x) (1)

is the formal statement of the condition where 'Perm' names the list permutation relation and 'Safe' names the unary relation which is true of a list of numbers only if for any number on the list in position i, and any other number n on the list in position j, j > i, $|m - n| \neq j - i$. This ensures that the list of numbers names a queen configuration that is safe.

We now need to give the Horn clause descriptions, of the Perm and Safe relations. Trivially the empty list is a permutation of itself. A list u.x is a permutation of a list v.z if v is on the list u.x, and y is a permutation of u.x with v removed. Fig.6 illustrates this recursive definition.

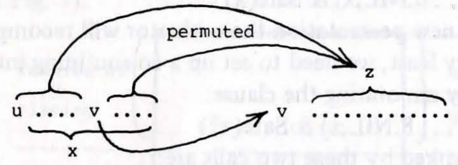

Figure 6. A permutation of lists

It is embodied in the two Horn clauses:

Perm(NIL,NIL)
Perm(u.x,v.z) ←delete(v,u.x,y) & Perm(y,z)

where 'delete' names the relation that is true when y is the list u.x with v removed. The clauses:

delete(v,v.x,x)
delete(v,u.x,v.y) ←delete(v,x,y) (2)

are a Horn clause description of the delete relation. We leave the reader to check that they provide a correct description which covers all the cases.

We now turn to the description of the Safe condition. Clearly an empty list of numbers represents a safe placing of queens on consecutive rows. A constructed list u.x will be safe if the list x of queen positions is safe and, taking into account that it is the column position for a queen one row away from the x sequence, a column u is not on a diagonal with any of the consecutive columns positions of the list x. This gives us the clausal description:

Safe(NIL)
Safe(u.x) ← no-diagonal(u,x,1) & Safe(x) (3)

Finally we need to describe the no-diagonal relation. This is a relation satisfiied by a column position, u, a list of column positions for consecutive rows, x, and a number, n, when a queen placed in column u, n rows away, cannot take any of the queens placed according to the list x. It is:

```
no-diagonal(u,NIL,n)
no-diagonal(u,v.x,n) ← no-take(u,v,n) & no-diagonal(u,x,s(n))        (4)
no-take(u,v,n) ← v > u & v = u + w & w ≠ n
no-take(u,v,n) ← u > v & u = v + w & w ≠ n
```

Here, '>', '. . = . . + . .' and '≠' are assumed as primitive arithmetic predicates.

The clauses (1), (2), (3) and (4) constitute a Horn clause axiomatisation of the concept of an eight queens problem solution. Note that they are much closer to a specification than a program. The ability to use logic programs that are more like specifications than programs is one of the bonuses of having a rich control component. Kowalski (1979a) elaborates on this point. Let us now turn our attention to their computational use.

With a mind to this computational use we have already chosen a suitable try order for the clauses of each predicate and for the procedure calls of each clause. However the computational use of the clauses as written suffers from considerable redundancy. The major problem is the generate and test method for finding a safe permutation which results from the back-tracking use of the clause:

```
Queen-sol(x) ← Perm(1.2 . . . . 8.NIL,x) & Safe(x)
```

Each time we back-track to try a new permutation the evaluator will recompute an entire permutation. At the very least, we need to set up a co-routining interaction between these two calls, by annotating the clause:

```
Queen-sol(x) ← Perm(1.2 . . . . . 8.NIL,x) & Safe(x?)
```

The procedures that will be invoked by these two calls are:

```
Perm(u.x,v.z) ← delete(v,u.x,w) & Perm(w,z)
Safe(u.x)     ← no-diagonal(u,x,1) & Safe(x)
```

A data call to the Safe computation will take place as soon as the shared variable of Perm(1.2. . . . 8.NIL,x) and Safe(x?) is bound to the template 'v.z', before the v giving the first queen position has been computed by the delete(v,u.x,y) call. It is better if we delay the call until v is known. We do this by changing the '&' of the Perm procedure to a ':'.

There is another more serious redundancy which cannot be remedied using a control annotation. The problem is with the recursive description of Safe, which must be changed to a more computationally useful description. Our recursive description implicitly involves a double recursion, since the no-diagonal relation is itself recursively defined. Computationally this means that the evaluation of the no-diagonal(u,x,1) call of the Safe procedure will generate a sequence of data requests that results in the complete construction of a permutation. In other words, a complete permutation will be constructed in order to test that the first queen does not take any of the tail list of queens. Thus, when the third queen position is given on some data transfer from Perm, only the diagonal constraint with the first queen is checked. The diagonal constraint with the second queen is embedded in the recursive call of the Safe procedure, and will not be checked until all the diagonal constraints involving the first queen have been checked.

A richer control regime would enable us to 'extract' a viable computation from the program as it stands. We need to be able to signal the quasi-parallel execution of the no-diagonal(u,x,1) and Safe(x) calls of the Safe procedure. More precisely,

we want the Safe(x) evaluation to be started, and then run in parallel with the evaluation of no-diagonal(u,x,1), when its argument becomes bound. The use of annotations to constrain a parallel evaluation of the procedure calls of a logic program, rather than to liberate the strictly sequential execution, has been investigated by Bruynooghe & Clark (1979). There is currently no implementation of such a control regime.

To make do with the control facilities that are available in IC-PROLOG we must revise the logic of our program. Behaviourally, we want the new description of the Safe relation to be such that each newly generated queen position is checked against all the preceeding queens rather than all the queens yet to be positioned. Intuitively it exploits the fact that a queen list is Safe iff its reverse is Safe.

To get this behaviour we must re-describe the Safe relation using an auxiliary relation, Safe-pair. Safe-pair is true of pairs of queen lists y and x iff x is a Safe list and no queen in the x sequence can take any queen in the reverse of the y sequence, when the reverse of y followed x is taken as a queen board configuration (see Fig. 7).

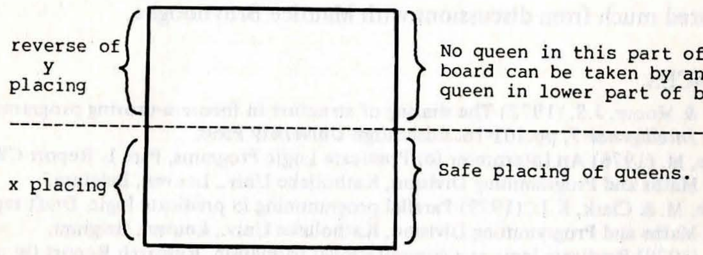
reverse of y placing
No queen in this part of board can be taken by any queen in lower part of board.
x placing
Safe placing of queens.

Figure 7. y and x are a safe-pair of queen placings

We leave the reader to check that, with this intended interpretation of the predicates 'Safe' and 'Safe-pair', each of the following clauses is a true statement:

```
Safe(x) ← Safe-pair (NIL,x)

Safe-pair(y,u.x) ← no-diagonal(u,y,1) & Safe-pair(u.y,x)
Safe-pair(y,NIL)
```

They embody a conceptualisation of the Safe relation as a special case of the Safe-pair relation, and a recursive description of Safe-pair.

Our final, revised and annotated program is:

```
Queen-sol(x) ← Perm(1.2.3.4.5.6.7.8.NIL,x) & Safe(x?)

Perm(NIL,NIL)
Perm(u.x,v.z) ← delete(v,u.x,y) : Perm(y,z)

delete(u,u.x,x)
delete(v,u.x,u.y) ← delete(v,x,y)
Safe(x) ← Safe-pair(NIL,x)
```

```
Safe-pair(y,NIL)
Safe-pair(y,u.x) ← no-diagonal(u,y,1) & Safe-pair(u.y,x)

no-diagonal(u,NIL,n)
no-diagonal(u,v.x,n) ← no-take(u,v,n) & no-diagonal(u,x,s(n))

no-take(u,v,n) ← v>u & v = u+w & w≠n
no-take(u,v,n) ← u>v & u = v+w & w≠n
```

The execution of the program corresponds to the simple-minded back-tracking algorithm which places the queens one at a time on successive rows. It always tries to place the next queen on the left-most free column (because of the try order for the 'delete' clauses). If it cannot place a queen on the next row in a position in which it cannot be taken by an earlier queen, it back-tracks and tries to move the previous queen to the right.

ACKNOWLEDGEMENTS

Our implementation of the control facilities in IC-PROLOG built upon the experience gained by Chris Stevens' pilot implementation (Stevens 1977). We have also benefited much from discussions with Maurice Bruynooghe.

REFERENCES

Boyer, R.S. & Moore, J.S. (1972) The sharing of structure in theorem-proving programs. *Machine Intelligence 7*, pp.101-16. Edinburgh University Press.

Bruynooghe, M. (1976) An Interpreter for Predicate Logic Programs, Part 1. Report CW 10, Applied Maths and Programming Division, Katholieke Univ., Leuven, Belgium.

Bruynooghe, M. & Clark, K.L. (1979) Parallel programming in predicate logic. Draft report, Applied Maths and Programming Division, Katholieke Univ., Leuven, Belgium.

Clark, K.L. (1979) Predicate logic as a computational formalism. Research Report (in preparation), Imperial College, London

Clark, K.L. & McCabe, F. (1979) *IC–PROLOG Reference Manual.* CCD Rep. 79/7, Imperial College, London.

Friedman, D.P. & Wise, D.S. (1976) CONS should not evaluate its arguments. *Automata, Languages and Programming*. Third International Colloquium. Edinburgh University Press.

Futo, I., Darvas, F. & Cholnoy, E. (1977) Practical application of an AI Language 2. *Proceedings of the Hungarian Conference on Computing*, Budapest, pp.385-400.

Green, C.C. (1969) Theorem-proving by resolution as a basis for question-answering systems. *Machine Intelligence 4*, pp.183-205. Edinburgh University Press.

Hayes, P.J. (1973) Computation and deduction. *Proc. 2nd MFCS Symp.*, Czechoslovak Academy of Sciences, pp.105-18.

Henderson, P. & Morris, J. (1976) A lazy evaluator. *3rd. Symp. on principles of programming languages*. Atlanta, pp.95-103.

Hill, R. (1974) LUSH Resolution and its completeness. DCL Memo No. 78, University of Edinburgh, School of Artificial Intelligence, August 1974.

Hoare, C.A.R. (1973) Recursive data structures. Computer Science Dept., Stanford University, STAN-CS-73-400.

Kowalski, R.A. (1974) Predicate logic as programming language. *Proc. IFIP 74*, pp.569-74. North Holland Publishing Co., Amsterdam.

——(1979a) Algorithm = Logic + Control. To appear in CACM.

——(1979b) Logic for Problem Solving. To be published by North Holland, N.Y.

Robinson, J.A. (1965) A machine-oriented logic based on the resolution principle. *J.A.C.M. 12* (January 1965) 23-41.

——(1979) *Logic: Form and Function*. Edinburgh University Press.

Roberts, G. (1977) An implementation of PROLOG. MSc Thesis. University of Waterloo.

Schwarz, J. (1977) Using annotations to make recursion equations behave. Research Memo, Dept. of Artificial Intelligence, University of Edinburgh.

Steven, C. (1977) The application of call-by-need to automatic theorem-proving. MSc Thesis. Department of Computing and Control, Imperial College, London.

Van Emden, M.H. & Kowalski, R.A. (1976) The semantics of predicate logic as a programming language. *J.A.C.M. 23*, no.4, 733-42.

Warren, D.H.D. (1977) Implementing PROLOG. Res. Rep. 39, 40, Dept. of A.I., Univ. of Edinburgh.

Warren, D.H.D., Pereira, L.M. & Pereira, F. (1977) PROLOG – The language and its implementation compared with LISP. Proc. Symp. on AI and Programming Languages, SIGPLAN Notices, vol.12, no.8, and SIGART Newsletters no.64, August 1977, pp.109-15.