IC–PROLOG LANGUAGE FEATURES

K.L. Clark F.G. McCabe S. Gregory

Imperial College of Science and Technology,
Department of Computing,
180 Queen's Gate, London, SW7 2BZ

ABSTRACT

In this short paper we introduce the principal features of IC-PROLOG mainly through examples. IC-PROLOG differs from PROLOG in not providing extra logical primitives such as the slash ("/") and "isvar", nor does it allow the addition and deletion of clauses during a query evaluation. On the plus side, negation and set expressions are primitives of the language, and there is a rich set of control facilities. For example, a programmer can:
(1) make the evaluation order of the calls of a procedure dependent upon its mode of use,
(2) initiate the (pseudo) parallel evaluation of a set of calls in which shared variables are one way communication channels.
The control is specified by annotations which have no effect on the declarative semantics of the program. The evaluation of an IC-PROLOG program is genuinely a controlled deduction.

1. PROCEDURES AND QUERIES

An IC-PROLOG procedure is an implication of the form:

$$B \leftarrow L_1 \& .. \& L_k,\ k \geq o$$

where B is an atomic formula and each L_i is a literal, i.e. an atomic formula or its negation. Atoms are of the form $R(t_1,..,t_n)$ where R is a relation name and each t_i is a term.

As in DEC-10 PROLOG (Pereira et al. 1978) the syntax can be

modified by declaring operators. This enables one to use infix form for binary relations, using r R t' instead of R(t,t').

The declarative reading of the procedure is as an implication universally quantified with respect to all the variables of the procedure[1]. The procedures describe a set of relations over terms which are the data structures of the program.

A query is of one of the three forms:

(i) : $L_1\&..\&L_k$
(ii) t: $L_1\&..\&L_k$
(iii) {t: $L_1\&..\&L_k$}

in which t is a term and each L_i is a literal. Let $x_1,...,x_n$ be all the variables of the conjunction $L_1\&..\&L_k$ and let $y_1,..,y_k$ be the subset of these that do not appear in t. The queries are read:

(i) For some $x_1,..,x_n$, $L_1\&..\&L_k$
(ii) A t such that for some $y_1,..,y_k$, $L_1\&...\&L_k$
(iii) All the t's such that for some $y_1,..,y_k$, $L_1\&..\&L_k$

Query Evaluation. Both queries and procedures can be annotated in ways that will shortly be exemplified in order to control the query evaluation process. The default evaluation of a completely unannotated program and query is that of standard PROLOG. Literals (calls) of the query are selected one at a time in left to right order. Failure to prove the literal L_i with the variable bindings generated by the preceding calls $L_1\&..\&L_{i-1}$ causes backtracking. Set queries of the form (iii) are answered by backtracking after each successful evaluation until there are no untried proof paths. The before/after order of the program procedures is the order in which they will be tried during backtracking.

Negated Atoms. Negated atoms are evaluated using the negation as failure proof rule. That is, -A is assumed true if all attempts to prove A fail. The call -A fails (i.e. is assumed false) only if there is a proof of A that does not bind any variable of the atom. If A can only be proved with some variable bound, the evaluation terminates with an error message. As explained in (Clark, 1978), this proof rule is

[1]There is an exception to this rule if the procedure contains set equalities. See section 2.

sound on the assumption that the relation R of atom A, and any auxiliary relations used to describe R, are relations that are completely defined by the procedures of the program. The programmer implicitly declares that this is the case by using a negation involving R.

Example Use of Negation. Suppose that we want to find unmarried children of Bill. We need to pose the query:

All the x such that Bill is the father of x and it is not the case that, for some y, x is married to y.

The only way we can express this in IC-PROLOG is to introduce the auxiliary relation, married(x), whose definition is the condition

"for some y, x is married to y"

that we wish to negate. We therefore add the procedure:

married(x) <- x married-to y .

We can now pose the query as:

{x: Bill father-of x & -married(x)}

2. PRIMITIVE RELATIONS

IC-PROLOG has primitive relations for natural number arithmetic, for reading and writing files, and for constructing a list of all the solutions of a query. The reading and writing of files we shall describe in section 4. Here we illustrate the use of the arithmetic relations and the set expression.

Arithmetic Relations TIMES, PLUS, LESS. The arithmetic relations are unusual in that there are no input/output restrictions on their use. For example, the primitive TIMES relation can be used to multiply, to divide, to find all the pairs of divisors of a number, even to generate all the tuples of natural numbers which lie in the relation. This generality of use allows one to write elegant arithmetic programs that are just the obvious definitions of the relations they compute.

Abstractly, the arithmetic primitives can be viewed as though defined by a data base of assertions that gives all the

instances of each relation. The natural number arguments can be denoted by successor terms or the usual decimal numerals. Thus "s(s(0))" and "2" are synonymous, and the term "s(s(x))" denotes any number greater than 1. The query:

```
{<s(s(x)), y>:TIMES(s(s(x)),y,36) & LESS(s(x),y)}
```

returns the set: {<2,18>,<3,12>,<4,9>,<6,6>}.

```
even(y) <- TIMES(x,2,y)
```

defines the property of being even. It can be used for testing or generating.

```
x divides z <- TIMES(x, y, z)
```

defines the divides relation. It can be used for testing, for finding divisors, or for finding multipliers.

```
has-divisor(z) <- s(s(x)) divides z & -s(s(x))=z
```

defines the property of having a proper divisor. Used for generating, as in the query:

```
{z: has-divisor(z)}
```

it will generate the infinite set of properly divisible natural numbers: {4, 6, 8, 9, 10,....}. (More exactly, it generates the set until the query evaluation is interrupted or the numbers exceed the range handled by the host computer.)

Finally,

```
prime(z) <- LESS(1, z) & - has-divisor(z)
```

defines the property of being a prime and

```
x prime-divisor-of z <- x divides z & prime(x)
```

defines the prime divisor relation. The query:

```
{u: prime(u)}
```

gives all the primes, and the query:

```
{u: u prime-divisor-of 100}
```


gives all its prime factors of 100. The definitions of "prime" and "prime-divisor-of" are really specifications of these relations. IC-PROLOG enables these specifications to be used, somewhat inefficiently, for computing instances of the relations.

Set Constructor. The set constructor is an equality of the form:

```
x = {t: A},    t a term, A an atom                    (1)
```

Used in a query or procedure it is logically equivalent to the non-atomic condition:

$$\forall u(u \text{ in } x \leftrightarrow \exists y_1,..,y_k(u=t \& A)). \qquad (2)$$

where $y_1,..,y_k$ are 'local' variables that only appear in t:A. As an example, the procedure:

```
p mother-of-children l <- female(p) & l =  q:q child-of p
```

is equivalent to:

```
p mother-of-children  l <- female(p) & ∀u(u in l <->
                                        Eq(u=q & q child-of p))
```

The p is not existentially quantified since it appears in another condition of the query. It is not 'local' to the set expression. Universally quantified equivalences such as that above can be expressed directly in the logic language of Hansson et al. (1982).

The set constructor (1) is used to generate a binding for x. This is its only use. It cannot be used to generate a binding for any other free variable of the equivalence (2), or to test that some list satisfies the condition.

Every member of the list generated for x is an instance ts of the term t. Here, s is a set of bindings for the local variables $y_1,..,y_k$ such that As is true. The list constructed by finding all the solutions to the query t:A . Each solution instance ts becomes an element on the list x. As with negation as failure, the evaluation method is sound on the assumption that the relation of A is completely described by the program. However the evaluation method does not necessarily generate the smallest list x that satisfies condition (2). This is because different evaluation paths of the query t:A may give rise to the same answer instance t' of

t. In this case more than one copy of t' appears on the list binding for x.

Example use

(a) {<s, l>: student(s) & l = {u: s takes u}}

This finds all the pairs <student, list of courses the student takes>. Using the expansion rule given above, it is equivalent to:

{<s, l>: student(s) & ∀u(u in l <-> s takes u)}

(b) l is-list-of-prime-factors-of x <-
l = u:{u prime-divisor-of x}

This defines the relation that holds between a number x and the list l of its prime divisors. It is equivalent to:

l is-a-list-of-prime-factors-of x
<- ∀u(u in l <-> u prime-divisor-of x)

Because of the restrictions on the use of the set constructor the procedure can only be used for finding the list of prime factors of a given number x.

3. LINKING CONTROL WITH USE

As we have already mentioned, the default control of IC-PROLOG is left to right evaluation of the conjunction of conditions of a query. This rule also applies to the preconditions/calls of a procedure. Unfortunately it is not always possible to find an ordering of the preconditions that is appropriate for every use.

As an example, consider the following definition of the "has-descendant" relation:

```
x has-descendant y <- x parent-of y
x has-descendant y <- x parent-of z & z parent-of y
x has-descendant y <- x parent-of z & z has-descendant w
                      & w parent-of y
```

Used to find descendants, in a query such as

{y: Tom has-descendant y},

the ordering of the three procedures and their preconditions results in a backtracking search which starts with the given "Tom". The first procedure finds all the children of "Tom". The second procedure generates the grandchildren of "Tom" by finding the children of his children. Finally, the last procedure will find all other descendants by finding each child of a descendant of one of his children.

Used to find ancestors, in a query such as:

{x: x has-descendant Bill},

the ordering of the preconditions results in a very inefficient search. Thus, the second procedure will generate the grandparents of Bill not by finding the parents of his parents, but by searching through all the parent-child pairs until a parent of Bill is found. The more efficient search would require the preconditions of the procedure to be in the reverse order. The last procedure also requires a reverse order of its preconditions if its use to find ancestors is to be an efficient search.

In standard PROLOG, one way round the problem is to define the complement relation, has-ancestor, with the ordering of preconditions appropriate to the finding of ancestors. Then this relation is used instead of the has-descendant relation for calls intended to generate ancestors. But this involves introducing a logically redundant relation, and it detracts from the invertibility property that is unique to logic programming. Another solution, again in standard PROLOG, is to use the meta-level primitive isvar which tests whether a variable is bound. The second procedure for the has-descendant relation is then expanded to the two procedures:

```
x has-descendant y <-
                    isvar(y) & x parent-of z & z parent-of y
x has-descendant y <-
                    -isvar(y) & z parent-of y & x parent-of z.
```

The major drawback of this solution, as with the use of the other meta-level primitives of PROLOG, is that it affects the declarative reading of the procedures. The procedures are no longer implications that can be read as simple definitions of relations.

In IC-PROLOG, no such pollution of the declarative semantics is allowed. Meta-level conditions, such as isvar(y), and all other issues of control are expressed in a separate language

of program annotations. The above pair of procedures become the annotated control alternatives:

```
[x has-descendant y^ <- x parent-of z & z parent-of y,
 x has-descendant y? <- z parent-of y & x parent-of z]
```

The "^" on the y of the first procedure expresses the control condition that y must be unbound on entry to the procedure. The "?" annotation of the second procedure is the control condition that it should be bound to a non-variable. These are the exact equivalents of isvar(y) and -isvar(y) conditions. Finally, the bracketing together of the procedures tells the evaluator that they are control alternatives, not logical alternatives. A similar pair of control alternatives can be given for the last has-descendant procedure. The program will then result in reasonably efficient search for all modes of use.

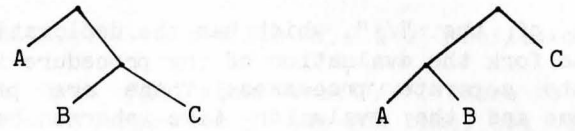
For further discussion of control alternatives, and for a description of the semantics of the head annotations, we refer the reader to (Clark and McCabe, 1979b). Head annotations are similar to the mode declarations of Dec-10 PROLOG (Pereira et al. 1978). However the annotations allow the expression of more complex input/output modes, and unlike Dec-10 PROLOG, the mode declaration for a procedure is translated into a runtime test. In the logic programming language proposed by (Hansson et al. 1982), different orders of evaluation of the preconditions of a procedure are generated automatically for different modes, the ordering being based on a topological sort in which calls which contain input arguments have priority. This removes the responsibility from the programmer, but does not necessarily result in the most efficient order of evaluation. It also does not cover the case when the control for different modes is not just a different order of sequential evaluation, but is a different mix of sequential/non-sequential evaluation.

4. NON-SEQUENTIAL EVALUATION

Various forms of non-sequential evaluation can be specified in IC-PROLOG. We shall exemplify them by considering the classic illustration of the benefit of non-sequential evaluation, the problem of checking that two binary trees have the same leaf profile.

The following procedures are a logic program that is essentially a specification of the "sameleaves" relation on

trees. The different trees:



have the same leaf profile. We use "t(x,y)" to denote a tree with subtrees x and y and "l(u)" for a tree with just the label u. The term "u.x" denotes the list with head u and tail x, "Nil" is the empty list.

```
sameleaves(x,y) <- w profile-of x & w profile-of y

u.Nil profile-of l(u)
u.z profile-of t(l(u),y) <- z profile-of y
w profile-of t(t(x,y),z) <- w profile-of t(x,t(y,z))
```

Let us now consider the test use of the program. A sequential execution of the "sameleaves" procedure means that the profile of the tree x is generated first and this is then tested against the profile of y. (So we have both a generate and test use of the procedures defining "profile-of".) If the trees have the same profile, both trees need to be traversed, and a sequential execution is as good as any. But if not, it is wasteful to generate the leaf profile of x beyond the point at which they differ. We need to specify a control that will ensure this early cut-off.

First, let us notice that the evaluation of the call w profile-of x will generate the output binding for w as a series of partial approximations. Thus, suppose that x is the tree t(l(A),t(l(B),l(C))). The second procedure for "profile-of" is the only one that applies to the call "w profile-of t(l(A),t(l(B),l(C)))" and its use will bind w to A.z. Here z is the profile of t(l(B),l(C)) yet to be generated. The next step in the evaluation will actually bind z to B.z', which implicitly binds w to the next approximation A.B.z'. Thus, the generation of the profile of the tree is such that each leaf label is 'made available', through the binding of w, as soon as the leaf is visited. There are three evaluation strategies for the sameleaves procedure that can exploit this label by label generation of the output binding for w.

Unsynchronised Parallel Evaluation. The simplest strategy is to execute the two "profile-of" calls in parallel. This is specified in IC-PROLOG by replacing the "&" by "//".

```
sameleaves(x,y) <- w profile-of x // w profile-of y
```

The effect of the "//", which has the declarative reading "and", is to fork the evaluation of any procedure in which it appears into separate processes. These are placed on a process queue and the evaluator time shares between the processes by rotating the queue. In each time slice the process at the head of the queue, if not suspended for some reason (see below), is given a time slice sufficient for at least one resolution step. Notice that this means that only one process can bind a variable. It is never the case that two processes try to bind the same variable at the same time. After a process has bound a variable, all other processes must read and agree with that binding.

In our example, this means that the first "profile-of" process to bind w is the one which has the tree with the shortest path to the leftmost label. Thereafter, the other process must 'read' this binding. The evaluations proceed with labels added to and read from w in an order determined by the shapes of the trees. As soon as both processes reach a mismatched label, the parallel evaluation fails.

Parallelism with Directed Communication. We can restrict the parallel evaluation so that only one process is allowed to generate the binding for the shared variable w. We do this by either annotating the occurrence of w in the producer process with a "^", or by annotating it in the consumer process with a "?". Thus,

```
sameleaves(x,y) <- w profile-of x // w^ profile-of y
```

makes the second call the producer of the leaf profile. In the parallel evaluation, the first consumer call becomes suspended if it tries to add a label to w. It becomes a read-only process on the binding of w. Each time the second process finds a new leaf, the first process is reactivated in order to check the label.

Data Triggered Coroutining. The above direction of communication constraint on w prevents the consumer process from doing unnecessary work visiting labels that occur after the mismatch. It does not prevent the producer generating extra labels. To prevent this, we need to specify a control in which the producer process is suspended wherever it finds a new label, and is only reactivated when the consumer process is resuspended because it needs the next label. This is specified by retaining the producer annotation on the variable, but by reverting back to the "&" connective.

```
sameleaves(x,y) <- w profile-of x & w^ profile-of y
```

Because we have "&" rather than "//", there is no forking. Only one process is active at any one time, but there is an alternation between the evaluations of the two calls. The "^" annotation makes the second call a lazy producer of the binding for the shared variable w.

The above example has served to illustrate three kinds of non-sequential control that one can specify using the annotations of IC-PROLOG. The unconstrained parallel control is the simplest. It simulates a parallel evaluation of conjunctive conditions in which the only constraint is that only one process is allowed to bind a variable. This is the form of parallelism discussed by Hogger (1982). Parallelism with designated producer processes for shared variables enables one to simulate networks of parallel processes with one way communication channels. This is the parallelism of the Kahn and McQueen model (1977). It is also treated in the logic programming context by van Emden and de Lucena (1982) and by Hansson et al. (1982). Finally, coroutining with a designated producer corresponds to lazy evaluation in a functional language (Friedman and Wise 1976).

Non-Sequential Control with Backtracking. Because the different forms of non-sequential control are specified explicitly with annotations they can be mixed. Coupled with backtracking search, this allows a rich variety of control strategies to be specified.

As an example we give the top level procedures of a solution to the eight queens problem. This is a slight modification of a purely coroutining solution to the problem given in (Clark and McCabe, 1979b). The procedures that complete the program are given in that paper. The candidate solutions to the problem are permutations of the list of the numbers 1 to 8, the i'th number in the permutation being the column position of the queen in the i'th row.

```
Queens-sol(x)<- Safe(x) & x^ perm-of 1.2....8.Nil
Safe(u.x)<- u cannot-take-any-of x // Safe(x)
```

The "^" annotation in the "perm-of" call makes this a lazy producer of the permutation. The relation can be defined so that the permutation is generated as a stream of partial approximations, as with the leaf profile of a tree. Each new

number placed on x is a new queen that is immediately checked by the Safe(x) condition. The evaluation of this is a forking parallel computation. For each queen placed on the board there is a new process generated to check that it cannot take any of the queens yet to be placed. Failure of any of these checks on the new queen results in backtracking to find a different placing. Thus, each candidate partial solution becomes a phalanx of parallel processes which grows and shrinks with the backtracking evaluation of the "perm-of" call. This sophisticated algorithmic behaviour results from simple control annotations attached to a program that is close to a specification of the problem that it solves.

Other Control Annotations. There are two other ways of controlling a parallel evaluation. There is a delay primitive which is a "!" immediately following a variable in a call of a procedure. If a process invokes the procedure, and its evaluation reaches the "!" annotated call, the process will be suspended until the variable is bound by some other process. The annotation is ignored if every other process is suspended. The following example is an annotated version of a program given in (Kowalski, 1979a). It defines an admissible list of pairs of numbers as one in which the second number of each pair is double the first, and in which the first number of the next pair is three times the second number of the preceding pair.

```
Admissible(l) <- Double(l)//Triple(l)
Double(Nil)
Double(<x,y>.l) <- TIMES(x!,2,y) & Double(l)
Triple(<x,y>.Nil)
Triple(<x,y>.<z,w>.l) <- TIMES(y!,3,z) & Triple(<z,w>.l)
```

The program will test or generate a list of admissible pairs by a parallel evaluation of the two conditions that it must satisfy. One generate use is particularly efficient. This is for calls of the form "Admissible(<N,x>.l)" in which the N is given. The delays on the "TIMES" calls of the "Double", "Triple" procedures mean that the remaining numbers on the list will be generated by a deterministic sequence of multiplications of the seed N.

The last control annotation is ":". This can be used to make the evaluation of the first call of a procedure act as a guard (cf. Dijkstra, 1976) on its use. The effect of the ":" in a procedure

$B \leftarrow G: A_1 \& \ldots \& A_m$

is to make the unification with head B and the evaluation of the guard atom G an indivisible unit during a parallel or coroutined evaluation. It is most commonly used to delay the communication of variable bindings that result from the unification with B until after the successful evaluation of the guard. If the guard fails, the binding is not transmitted to the other processes. Guards are similar to the constraints of Bellia et al. (1982). The difference is that in IC-PROLOG a successful guard evaluation does not mean that the procedure is the only one that can be used for the call. It does not exclude the possibility that other procedures will unify with the call and have true guards.

5. STREAM I/O

Another unique feature of IC-PROLOG is stream I/O. The primitive READ(x) binds its argument variable not to a single character or term, but to the entire stream of characters that will be typed at the terminal. The programmer processes this stream as though it were a list of characters. The list is lazily produced by the evaluation of the READ(x) call. Thus, in a query of the form

```
y: READ(x) & P(x,y)
```

"P" must be a relation from a list of characters to the output y. Characters are read from the terminal as unifications in the evaluation of "P(x,y)" demand more of the list x. Because x is the list of all the characters typed there is no problem with backtracking. As characters are read in they are explicitly stored in the list binding for x that READ(x) is lazily producing. After backtracking, evaluation steps of the "P(x,y)" call get characters from this partially recorded list. Only when this is exhausted will new characters be read from the terminal. This processing of streams generated at the terminal is handled by a modified unification algorithm that 'knows' about special values that are pointers to the terminal buffer.

The primitive WRITE(y) will display the list binding for y at the terminal. As with the lists produced by READ(x), special constants are used to denote invisible characters. Thus the constant "LINE" will be 'displayed' by generating a carriage return.

WRITE(y) can also be used with its argument generated as a stream. Consider a query of the form

: READ(x) & WRITE(y) & R(x,y^),

in which "R" is a relation over lists of characters that generates the output binding for y as a stream as it consumes x. The evaluation of the query will interleave lines of output with lines of input. Each carriage return that is typed is a signal that allows the display of the next segment of y that ends with the "LINE" constant.

6. CONCLUDING REMARKS

IC-PROLOG is a pure logic based language that enables one to illustrate a wide variety of programming concepts. Using the set constructor and control alternatives one can develop general purpose deductive data bases. Using stream I/O and data triggered coroutining one can illustrate the idea of lazy evaluation. Finally, the parallel evaluation and the various ways of controlling it correspond to current ideas in the area of communicating processes. IC-PROLOG is therefore an ideal language for teaching these programming concepts. It has been used in this way at Imperial College and at Syracuse University with some success.

ACKNOWLEDGEMENTS

The research on IC-PROLOG was supported by the British Science & Engineering Research Council.

PROPERTIES OF A LOGIC PROGRAMMING LANGUAGE

Å. Hansson S. Haridi* S-Å. Tärnlund

UPMAIL
Computing Science Department,
Uppsala University, Uppsala, Sweden

*Department of Computer Systems,
The Royal Institute of Technology, Stockholm, Sweden

ABSTRACT

We have developed a logic programming system based on natural deduction. It consists of a class of statements which is a superclass of Horn clauses. We can run as programs logical statements that formerly have been considered specifications. For example, the language contains the logical constants negation, equivalence, universal quantifier and identity. We can define functions, as well as relations, infinite data structures and virtual classes. Computation rules provide control information. A demand driven computation rule results in computations on infinite data structures that terminate.

1. INTRODUCTION

Logic programming as in the Prolog systems (see Colmerauer et al. 1972; Pereira et al. 1978) is based on Horn clauses and a procedural reading of relations (see Kowalski, 1974). The logical system is resolution (Robinson, 1965). In contrast, our language is based on a natural deduction system (see Prawitz, 1965). Our procedures are special cases of co-operating agents. The language is first order and has the following features in common with Prolog: (1) general tree-like data structures, (2) non-determinate programs treated by automatic backtracking, (3) no distinction between input and output, (4) logical variables that enable programs to manipulate partially specified data structures, (5) tail recursion optimization.

Its additional features are: (1) truth functional semantics