PARLOG: A PARALLEL LOGIC PROGRAMMING LANGUAGE

Keith L. Clark & Steve Gregory

Research Report DOC 83/5
May 1983

ABSTRACT

PARLOG is a logic programming language in the sense that nearly every definition and query can be read as a sentence of predicate logic. It differs from PROLOG in incorporating parallel modes of evaluation. For reasons of efficient implementation, it distinguishes and separates and-parallel and or-parallel evaluation.

PARLOG relations are divided into two types: and-relations and or-relations. A sequence of and-relation calls can be evaluated in parallel with shared variables acting as communication channels. Only one solution to each call is computed.

A sequence of or-relation calls is evaluated sequentially but all the solutions are found by a parallel exploration of the different evaluation paths. A set constructor provides the main interface between and-relations and or-relations. This wraps up all the solutions to a sequence of or-relation calls in a list. The solution list can be concurrently consumed by an and-relation call.

The and-parallel definitions of relations that will only be used in a single functional mode can be given using conditional equations. This gives PARLOG the syntactic convenience of functional expressions when non-determinism is not required. Functions can be invoked eagerly or lazily; the eager evaluation of nested function calls corresponds to and-parallel evaluation of conjoined relation calls.

This paper is a tutorial introduction and semi-formal definition of PARLOG. It assumes familiarity with the general concepts of logic programming.

KEY WORDS AND PHRASES

Logic programming, applicative programming, non-deterministic programming, parallel programming, and-parallelism, or-parallelism.

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

Telephone: 01-589 5111 Telex: 261503
1. INTRODUCTION

PARLOG is a logic programming language in the sense that nearly every definition and query can be read as a sentence of predicate logic. Moreover, the results of a computation are logical consequences of the program definitions. It differs from PROLOG in incorporating parallel modes of evaluation. For reasons of efficient implementation, it distinguishes and separates and-parallel and or-parallel evaluation.

This paper is a tutorial introduction and semi-formal definition of PARLOG. It assumes familiarity with the general concepts of logic programming.

Brief history

PARLOG is a successor to the logic based parallel programming language described in our earlier paper [CIG 81] which was itself a derivative of IC-PROLOG [CMG 82]. The parallelism of the original language was restricted to the concurrent evaluation of different conditions in a query or program clause. The concurrent evaluations could communicate in one direction only via incrementally produced list bindings of shared variables. This is the analogue of the parallel evaluation of functional expressions returning list structures. The main difference between the relational language of [CIG 81] and a functional parallel programming language was non-determinism. A call returned some instance of a relation rather than the value of a function for given arguments. Which instance was returned was not determined by the program but depended on the evaluation paths taken by concurrent communicating relation calls.

A major feature of our earlier language, which distinguished it from other logic programming languages, was that only one solution to a relation call was ever returned and the allowed modes of use of every relation definition had to be declared. As in DEC-10 PROLOG [War 77], the mode declarations defined the input/output role of the argument patterns in the head of a clause but, because of the one-way communication, they also enabled the mode of every call in the body of a clause to be determined by a compile time analysis. Thus the mode declarations facilitated the compilation of very efficient code. The single solution property arose because only one matching candidate clause was ever used to evaluate a call. This is "committed choice" non-determinism, which we adapted from Hoare's CSP [Hoa 78].

Committed choice selection of a candidate clause combined with the parallel evaluation of conjoined calls (which is and-parallelism) is the appropriate form of parallel evaluation for applications of logic programming to concurrent process forms of parallel programming and it has since been adopted by Shapiro for Concurrent PROLOG [Sha 83]. It enables one to achieve the behaviour of message sending or object oriented programming languages, such as Smalltalk, using programs which describe the logic of the communication.

However, committed choice non-determinism is not suitable for concurrent search forms of parallelism, the form of parallel evaluation needed to find all solutions to a database query. The logic programming analogue of this is or-parallelism, the concurrent search for all of the different solutions to a relation call.

PARLOG incorporates both and-parallelism and or-parallelism.
PARLOG and-relations and or-relations

PARLOG relations are divided into two types: and-relations and or-relations. A sequence of and-relation calls can be evaluated in parallel with shared variables acting as communication channels. Again only one solution to each call is computed.

A sequence of or-relation calls is evaluated sequentially but all the solutions are found by a parallel exploration of the different evaluation paths. A set constructor provides the main interface between and-relations and or-relations. This wraps up all the solutions to a sequence of or-relation calls in a list. The solution list can be concurrently consumed by an and-relation call.

PARLOG and-parallelism generalizes that of the earlier language in several respects. The communication between concurrent relation calls is not restricted to incremental lists or streams, but can be any incrementally constructed data structure. For example, it can be a tree structure being constructed by a forking producer computation. The communication can also be bi-directional. A process can communicate a data structure containing unbound variables that are given values by some consumer process.

Back communication using variables in data structures gives to the and-parallel component of PARLOG some of the power of the use of the logical variable, a feature that is much used in sequential PROLOG and Concurrent PROLOG. It enables one to describe many communicating process computations as the task of constructing a single agreed stream of messages. However, the actual process of constructing the agreed stream is a two-way communication down a shared channel.

The modes of use of each and-relation definition are still declared but the form of declaration now allows reverse communication to be specified. The mode declarations retain the property that the mode of each and-relation call can be determined at compile time. There are no mode declarations for or-relations, they can be queried with any pattern of given arguments.

Conditional equations

The and-parallel definitions of relations that will only be used in a single functional mode can be given using conditional equations. This gives PARLOG the syntactic convenience of functional expressions when non-determinism is not required. Functions can be invoked eagerly or lazily; the eager evaluation of nested function calls corresponds to and-parallel evaluation of conjoined relation calls.

Structure of paper

The structure of the rest of the paper is as follows. It is divided into four main sections. In section 2 we describe the and-relation subset of PARLOG. In section 3 we describe the use of conditional equations and in section 4 the use of or-relations. Section 5 briefly describes two metalevel features of PARLOG. Section 6 is an extended example of a PARLOG program for a UNIX-style operating system. In each of the sections describing language features we begin with some examples, then give a formal description of the syntax and a semi-formal definition of the
operational semantics, and usually end with some more examples. In each case the formal sections can be skipped on first reading.

State of the language

A prototype compiler for PARLOG has been written both in PROLOG and in PARLOG. It compiles PARLOG to the ALICE Compiler Target Language [DaR 81]. It was the writing of the compiler in PARLOG that made us consider the inclusion of variables in data. We were further convinced of their utility when we saw their effective use in Concurrent PROLOG.

2. AND-RELATION DEFINITIONS

Each and-relation definition must be preceded by a declaration of its modes of use. The definition is separately compiled for each mode. The communication conventions are such that the mode of any call to an and-relation can be determined at compile time. An and-relation call is therefore compiled into a call to the compiled code for that mode. Only one solution to each and-relation call can be computed.

Example program 1

This program defines the relation append(x,y,z): z is list x concatenated to list y:

relation append(?,?,^)
relation append(?,^,?)
relation append(?[^],?)
append([],y,y).
append([u|x],y,[u|z]) :- append(x,y,z)

The program is preceded by three mode declarations because it can be used in all three modes. In the mode declaration, '?' means input and '^' means output. So the mode append(?[^],?) corresponds to the normal appending use and the mode append(?[^],?) the use to strip off a front sublist of a given list. For the appending mode the pattern [u|z] in the head of the second clause is compiled into code that constructs a new list cell with the given u as the head element. For the stripping mode it compiles it into code that decomposes the given list into a head u and a tail z.

Notice that the use to find all possible splittings of a list is not possible because 'append' is defined as an and-relation. Even if we preceded the definition with the mode

relation append(^[^],?)

a call in this mode would produce only one splitting. Which one is computed is not determined by the program, but depends on which of the two clauses is selected to evaluate the call. To use an 'append' program to find all decompositions, it has to be defined as an or-relation and called inside a set expression such as

{ <x,y> : append(x,y,[1,2,3]) }

which evaluates to the list of different splittings. We shall return to this example in section 4.
Example program 2

The relation $\text{merge}(x,y,z)$ means that $z$ is an arbitrary interleaving of lists $x$ and $y$. It too can be used in different modes, e.g.:

\[
\text{relation merge}(?,?,^*) \\
\text{relation merge}([^*],^*) \\
\text{merge}([^],[^],[^]) . \\
\text{merge}([u|x],y,[u|z]) :- \text{merge}(x,y,z) . \\
\text{merge}(x,[v|y],[v|z]) :- \text{merge}(x,y,z)
\]

The call $\text{merge}([1,2,3],[4,5,6],x)$ will bind $x$ to one interleaving of the two list arguments. Which one is not determined by the program.

Similarly, the call $\text{merge}(x,y,[1,2,3],[4,5,6])$ will bind $x$ and $y$ to a pair of lists that corresponds to some possible order preserving splitting of the given list. Again, the program does not determine which one.

Example program 3: set

Consider a relation $\text{set}(c,x,z)$: $z$ is the set $x$ after items have been added and deleted according to messages in the message stream $c$. $c$ is a list of 'Add' and 'Delete' messages, e.g.

\[[\text{Add}(2),\text{Add}(5),\text{Delete}(2),...]\]

Sets $x$ and $z$ are represented by some data structure, such as an ordered tree or an ordered list.

The 'set' relation can be recursively defined in terms of an 'insert' relation on sets:

\[
\text{relation set}(?,?,^*) \\
\text{set}([^],x,x) . \\
\text{set}([\text{Add}(u)|c],x,z) :- \text{insert}(u,x,y^*), \text{set}(c,y,z) . \\
\text{set}([\text{Delete}(u)|c],x,z) :- \text{insert}(u,y^*,x), \text{set}(c,y,z)
\]

$\text{insert}(u,x,y)$ means that $y$ is set $x$ with item $u$ added. Notice the inverted use of 'insert' for deleting an element. To use it in this way we must define 'insert' so that it can only be used to add an item not already in the set. The inverted use is then guaranteed to delete the element provided it is in the set.

In evaluating the call

\[\text{set}([\text{Add}(2),\text{Add}(5),\text{Delete}(2),...],x,z)\]

the number of processes grows dynamically as shown in Figure 1.
The additions and deletions may be performed concurrently provided that 'insert' starts to produce its output before completely consuming its input. The degree of concurrency depends on the amount by which the output of 'insert' lags behind its input. This in turn depends on the data structure chosen to represent the set.

To initialize the set, we define the single argument relation 'set':

\[
\text{set}(c) \text{ means that } c \text{ is a valid list of messages to an empty set.}
\]

\[
\text{relation set(\(?)

\text{set}(c) := \text{empty}(x^\text{''}), \text{set}(c,x,z)
\]

Later, we shall define the 'empty' and 'insert' relations, which are dependent on the chosen set representation.

Syntax of and-relation definitions

An and-relation definition is a clause group headed by one or more relation declarations.

A relation declaration takes the form

\[
\text{relation } R(m_1,\ldots,m_n)
\]

where \(R\) is the name of an \(n\)-ary and-relation and each \(m_i\) indicates the mode of the \(i\)-th argument. '?' means that the argument is input while '"' means output.

There may be more than one relation declaration if the relation is to be used in several modes. A clause group can only be used in one of its declared modes, but different clause groups, i.e. different defining programs, can be given for different modes of the same relation. If different programs are given for different modes, they should be equivalent descriptive definitions.

A clause group is of the form
Intuitively, ',' is a parallel 'and' connecting the clauses and ';' is a sequential 'and'. The difference is only in the operational semantics. The combination of ',' and ';' clause connectives defines the search strategy used to find a clause to apply to a call. We shall return to this later. The operators ',' and ';' have the same precedence so brackets must be used to indicate groupings. A clause group is terminated by the next mode declaration or by the end of the program.

A clause unit is

(<clause group>) |
<clause>

A clause is of the form

H :- G | B

where the guard G and body B are relational expressions. The head of the clause, H, is an atom. If G is absent the '!' may be omitted. If both G and B are absent the ':' may also be dropped. The '!' is read as 'and', but has a special significance in the operational semantics, as we shall see. Later we shall see that the variables of the head can be annotated in order to refine the mode declaration at the head of the clause group. Similarly, variables in the guard and body can be annotated to override certain default assumptions concerning the mode of calls in the guard and body.

A relational expression is of the form

<atom group> |
<atom group> , <relational expression> |
<atom group> & <relational expression>

Intuitively, ',' is a parallel 'and' and '&' is a sequential 'and'. Again the difference is in the operational semantics. The ',' and '&' combination in the body of a clause determines the degree of parallelism used to evaluate the body of the clause when the clause is invoked.

An atom group is

(<relational expression>) |
<literal>

A literal is

<atom> |
- <atom>

An atom is of the form

R(t1,...,tn)

where R is a relation name and t1,...,tn are terms.

A term is a number, a variable (beginning with a lower case letter), a tuple <t1,...,tn> or a term of the form F(t1,...,tn) where t1,...,tn
(n>0) are terms and F is a constructor function. If n=0 the parentheses are omitted. A constructor function is either a name beginning with an upper case letter or an arbitrary string enclosed in single quotes (e.g., John, '$275'). Variable free terms are the data structures of PARLOG.

The above definition of term is provisional because we shall extend it in the next section to include evaluable function calls. But it simplifies our description of the operational semantics if we first consider only terms as defined here, which are really a special class of PARLOG term called data terms.

We use a special notation for lists, whereby the following pairs of terms are equivalent:

\[
\begin{align*}
[] & \quad \text{Nil} \\
[1,2] & \quad \text{Cons}(1,\text{Cons}(2,\text{Nil})) \\
[1|x] & \quad \text{Cons}(1,\text{x}) \\
[1,2|x] & \quad \text{Cons}(1,\text{Cons}(2,\text{x}))
\end{align*}
\]

'Cons' and 'Nil' are primitive list constructor functions of PARLOG.

The head of a clause must have the same name and number of arguments as the relation declaration(s) preceding the clause group in which it appears.

**Operational semantics**

An evaluation of a relational expression

\[
\text{AG} \& \text{RE}
\]

is reduced to an evaluation of AG sequentially followed by RE. An evaluation of

\[
\text{AG} , \text{RE}
\]

is reduced to a parallel evaluation of AG and RE.

In solving a single call \( R(t_1, \ldots, t_n) \) to an and-relation \( R \), the mode of the call will have been determined at compile time, so it will be matched against the clauses of the group for that mode. There are three categories of clause:

1. The candidate clauses (the head matches the call and the evaluation of the guard relational expression succeeds).
2. The non-candidate clauses (the head does not match the call or the guard evaluation fails).
3. The input suspended clauses (the attempt to unify the head of the clause with the call tries to bind a variable of an input argument of the call to a non-variable, i.e. the input data structure expected in this argument position is not yet sufficiently instantiated for a read only match).

The search for candidate clauses is controlled by the '.' and ';' structure of the clause group for R:

Structure is \( \text{CU} . \text{CG} \)
Find candidate clauses from CU and CG in parallel. Select the first one found.

Structure is CU ; CG

Find a candidate clause in CU if possible. If all clauses in CU are non-candidate clauses, then search in CG for a candidate clause.

When a candidate clause is found, the bindings to output arguments of the call become public (available to other concurrently executed calls) and the call is reduced to the evaluation of the clause body relational expression. There is no backtracking on this choice of clause to use an alternative candidate clause.

If there are no candidate clauses but there is at least one input suspended clause, the call is suspended. If there are no candidate clauses and no input suspended clauses, the evaluation of the call fails. If the clause body is empty, the evaluation of the call succeeds.

If the call is of the form

\[ R(t_1, \ldots, t_n) \]

the evaluation of the call is converted into the evaluation of \( R(t_1, \ldots, t_n) \) with every argument input. If the evaluation of the unnegated call succeeds (fails) the evaluation of the negated call fails (succeeds). This is the "negation as failure" operational semantics. It is valid provided the definition of \( R \) and the definitions of all auxiliary relations used to define \( R \) are complete definitions of the relations [Cla 78]. Notice that the evaluation of \( \neg R(t_1, \ldots, t_n) \) will suspend if there is an attempt to bind any variable in the call. It will wait until some concurrent evaluation generates a binding for the variable.

If the evaluation of a relational expression is reduced to the evaluation of a call that fails, its evaluation fails. The evaluation of a relational expression succeeds if all of the calls to which it is reduced succeed.

**Example program 4: quicksort**

We first need a definition of the relation partition\((u, x, y, z)\): \( y \) and \( z \) are lists of those numbers on list \( x \) which are less than or not less than \( u \) respectively, e.g. partition\((5, [3, 1, 7, 2], [3, 1], [7])\).

\[
\text{relation partition(?,, ,^,^)} \\
\text{partition(u,[],[],[])} . \\
\text{partition(u,[v|x],[v|y],z) :- v\less u | partition(u,x,y,z) .} \\
\text{partition(u,[v|x],[v|y],[v|z]) :- v\less u | partition(u,x,y,z)} 
\]

We can use this relation together with 'append' in a definition of a sort relation called 'quicksort', the evaluation of which is a parallel quicksort.
relation Quicksort(?,-)
Quicksort([],[]).
Quicksort([U|X],Z) :-
    partition(U,X,X1,X2),
    Quicksort(X1,Y1),
    Quicksort(X2,Y2),
    append(Y1,[U|Y2],Z)

The list to be sorted is partitioned into two lists X1 and X2. Each of
these is itself sorted to give Y1 and Y2 respectively, which are
concatenated. Here, all four processes proceed in parallel constrained
only by the "data flow" through the shared "channel" variables X1, X2, Y1
and Y2. The 'partition' call is the producer of X1 and X2, and the
'append' call is the consumer of the Y1 and Y2 generated by the two
'Quicksort' calls. The producers are implicitly specified by the left-
right order.

Alternative control

The relational syntax allows a choice of control strategies to be
imposed on a given logic program. For example, different mixes of
sequential and parallel execution for 'Quicksort' give radically different
behaviours:

Quicksort([],Z) :-
    partition(U,X,X1,X2) &
    ( Quicksort(X1,Y1), Quicksort(X2,Y2) ) &
    append(Y1,[U|Y2],Z)

Here, the lists X1 and X2 are completely constructed before they are
sorted, in parallel. The sorted lists Y1 and Y2 are completely constructed
before concatenation. There are no channel variables between the two
parallel processes. The completion of the 'partition' process is a fork
and the starting of the 'append' process is a join.

Bounded buffers

Normally, when we have two parallel processes connected by a shared
variable, there is no limit to the amount by which the (eager) producer can
run ahead of the consumer, i.e. buffers are unbounded.

We can limit this run-ahead by assigning a finite size to a channel
buffer: the maximum difference between the "depth" of the structure
produced and the depth of the structure consumed. We do this by writing
the size after a '^-1' annotation on the producer occurrence of the variable.

In the special but quite common case where lists are communicated, the
buffer size corresponds to the maximum number of elements of the list that
can be in the channel at any time.

For example, we can constrain the behaviour of our 'Quicksort' program
as follows:

Quicksort([U|X],Z) :-
    partition(U,X,X1^-1,X2^-1),
    Quicksort(X1,Y1^-1),
    Quicksort(X2,Y2^-1),
    append(Y1,[U|Y2],Z)
Now, each time the 'partition' call "produces" an item of list x1 or x2 by a partial binding, it will suspend until the item is "consumed" by the corresponding 'quicksort' call. This is because channels x1 and x2 have unit size. Similarly, the 'quicksort' calls cannot run ahead of the 'append' call due to the bounded buffers on channels y1 and y2.

In our earlier paper [CLG 81] we allowed the limiting case of a bounded buffer where the buffer size is zero, resulting in synchronized communication. However, synchronized communication coupled with non-deterministic evaluation is much more difficult to implement than buffered communication. This is because a buffer, if not full, will always accept another communication from the producer, whereas a non-deterministic consumer may not be in a position to accept the message at the point that it is sent. We have therefore abandoned the zero-buffer in the present version of PARLOG.

**Refinement of the operational semantics**

Because of buffer constraints there is actually a fourth category of clause for a particular call. This is the output suspended clause. It is a candidate clause for which the output argument bindings cannot be made public without overflowing an output channel buffer.

The call will be suspended if there are no candidate clauses but there is at least one input suspended or output suspended clause.

**Mode analysis**

The mode declarations associated with and-relation definitions specify very strong input/output constraints, much stronger than those of the DEC-10 PROLOG compiler [War 77]. If an argument position is input, the default assumption is that the evaluation of the call will only access the components of a data structure passed in that argument position and that the evaluation will not in any way contribute to the construction of the data structure. Conversely, if an argument position is output, the default assumption is that the evaluation of the call will completely construct the data structure that is passed out in that position.

These assumptions enable a compiler to generate access-only code for each input argument pattern in the head of a clause and to generate construct-only code for each output argument pattern. Coupled with other default assumptions about the flow of information through shared variables in the body of a clause, they also enable the input/output mode of each call to be determined at compile time. The mode analysis is as follows.

First of all, the input/output role of each occurrence of a variable in a call of the clause is determined. Any variable appearing in an input argument term in the head of the clause is input to all calls in the clause. Each other variable is such that some call in the body of the clause generates an output binding for the variable. In the absence of an explicitly designated output occurrence, signalled by a "#" annotation on the variable, the first occurrence of the variable is deemed the output occurrence. All other occurrences of the variable are assigned the input role.
Example of input/output analysis for variables

In the clause

\[ \text{relation } c(?,^) \\
    c(k(w),x) := (p(w,x) \& q(x,y)), (r(y^) \& s(y)) \]

the occurrence of \( x \) in \( p(w,x) \) is output while its occurrence in \( q(x,y) \) is input. The occurrence of \( y \) in \( q(x,y) \) would be output except that the \('^'\) annotation on its occurrence in \( r(y^) \) makes this the output occurrence. All other occurrences of \( y \) are input. The \( w \) in \( p(w,x) \) is input because it is a variable in the input argument of the head of the clause.

With these rules, the compiler can determine for each occurrence of a variable in a call whether it is input or output to the call. It then determines the mode of the call as follows. For each argument, if the argument contains no variables then the argument has input mode. If it contains only input variables then the argument has input mode. Finally, if it is a single variable, which is output, the argument has output mode. Any other case gives rise to a compile time error message. Thus a non-variable argument term which contains an output variable is not legal. A compile time error is also signalled if this analysis does not assign input mode to every argument of a negated call.

Example of illegal use of modes

\[ \text{relation } c(?,^) \\
    c(G(x),F(y)) := R(T(x,y)) \]

This is not legal because \( x \) is input to \( R \) and \( y \) is output and the mode of the single argument \( T(x,y) \) of \( R \) cannot be determined. \( R \) needs to be redefined so that it has two arguments, and the call then has \( x \) as input argument and \( y \) as output.

Overriding the read-only/construct-only assumptions

A powerful feature of logic programming is the ability to place variables in data structures. The data structure is then passed on and further instantiated by some consumer call. Different calls cooperate to construct an agreed data structure.

To allow this style of programming in PAMLOG, assumptions that will be made by the compile time mode analysis need on occasions to be explicitly overridden. This is done by attaching annotations to the argument terms in the head of the clause.

For example, in the definition

\[ \text{relation } p(?,^) \\
p([u;x],[<v,w?>;y]) := q(u,v), r(w,x,y) \]

the \('?\) annotation on the variable \( w \) of the output term \([<v,w?>;y]\) signals that \( w \) will not be bound by the evaluation of the body of the clause, but by some consumer of the partial output structure \([<v,w?>;y]\). The compiler therefore assigns the mode \( r(?^2,?) \) to the \('^'\) call of the clause rather than the mode \( r(?^2,?) \) which would be assigned in accordance with the default assumption that the evaluation of the clause
will completely instantiate its output arguments. So that the consumer can pick up the partial structure and instantiate the variable \( w \), it must have a "\( ^\wedge \)" annotation in its corresponding argument term. Thus, suppose that \( p \) communicates with \( c \) as in

\[ p(x,y), c(y) \]

The definition of \( c \) must have the form

\[
\text{relation} \quad c(?) \\
c([<v,w> | y]) : - s(v,w), c(y)
\]

**Example use of variables in data**

Returning to our earlier set example (example program 3), let us consider a representation of the set as an ordered list of pairs, terminated by a special "infinity" element. The empty set is represented by the singleton list \([\infty]\). We need to define 'empty' and 'insert'.

\[
\text{relation} \quad \text{empty}(\wedge) \\
\text{empty}([\infty])
\]

We shall take the first item of each pair to be a key. The ordering relation 'less' compares the keys, so there may be no more than one member of the set with a given key.

\[
\text{relation} \quad \text{insert}(?,?,\wedge) \\
\text{relation} \quad \text{insert}(?,\wedge,?) \\
\text{insert}(u,[v|x],[u,v|x]) : - \text{less}(u,v) \ldots \\
\text{insert}(u,[v|x],[v|y]) : - \text{less}(v,u) \ldots \text{insert}(u,x,y)
\]

\[
\text{relation} \quad \text{less}(?,?) \\
\text{less}(x,\infty) \\
\text{less}(<k,i>,<k1,i1>) : - k < k1
\]

Notice that the 'less' definition does not require the second item of the pair to be instantiated. One useful method of using the set is to send 'Add' messages and 'Delete' messages of the form \( \text{Delete}(<k,x>) \) with \( k \) given and \( x \) a variable. This means that an item can be deleted from the set by giving just the key; see Figure 2.

![Figure 2](image-url)
Notice the "back communication". When the message Delete(<2,x>) is consumed by 'set', the item <2,F> is deleted and x is bound to F. For this program, the back communication does not need to be explicitly signalled by a head annotation because it does not conflict with any of the compile time assumptions made by the mode analysis.

We now introduce a new type of message 'In' to test for membership of a given item in the set. If a message of the form In(<k,x>) is received by 'set', x will be bound as with the 'Delete' message, except that the item is not deleted. If the item is not in the set, x is bound to NOTFOUND.

To implement the 'In' message, we add a new clause to the 'set' definition and define the 'member' relation:

```prolog
set([In(u)[c]],x,z) :- member(u,x), set(c,x,z)
```

```prolog
relation member(?,?)
member(u,[u|x]).
member(u,[v|x]) :- less(u,v) \ notfound(u) .
member(u,[v|x]) :- less(v,u) \ member(u,x)
```

```prolog
relation notfound(?)
notfound(<k,NOTFOUND^>)
```

Notice the '^^' head annotation in the 'notfound' clause. It is required to override the default assumption that the input argument term <k,NOTFOUND> will be used for a read only match. The '^^' says that the second component of the given argument pair will be a variable which is to be bound to NOTFOUND. If the annotation were omitted, 'notfound' would suspend indefinitely, waiting for the variable to be given a value by some other process.

**Example program 5: set**

Here we give an interesting alternative program for the 'set' relation defined in example program 3. In this program, each member of the set is represented by a process. 'Add' and 'Delete' messages cause the dynamic creation and deletion of processes. This example was taken from a CSP program in [Hoa 78] and was previously defined as an IC-PROLOG program in [Gre 80].

The 'set' relation is defined in terms of 'item', where item(n,c,d) means that c is a valid history for an initial set {n} with respect to messages for numbers less than or equal to n, and d is the subsequence of c containing ignored messages for numbers greater than n.

```prolog
relation set(^)
set(c) :- item(\infty,c,d)
```
relation item
  item(n,[Add(m)|c],[Add(m)|d]) :- less(n,m) !
  item(n,c,d).
  item(n,[Add(m)|c],d) :- less(m,n) !
  item(m,c,x^y), item(n,x,d).
item(n,[Delete(m)|c],[Delete(m)|d]) :- less(n,m) !
  item(n,c,d).
item(n,[Delete(n)|c],c).
item(n,[In(m)|c],[In(m)|d]) :- less(n,m) !
  item(n,c,d).
item(n,[In(n)|c],d) :-
  item(n,c,d).
item(n,[In(m)|c],d) :- less(m,n) !
  notfound(m), item(n,c,d)

Figure 3 shows the processes involved before and after adding 2 and 5 to an empty set, and then deleting 2.

\[ \text{Add}(2), \text{Add}(5), \text{Delete}(2), \ldots \] → set
\[ \downarrow \]
\[ \text{[Delete}(2), \ldots] \] → item2 → item5 → item∞
\[ \downarrow \]
\[ \ldots \] → item5 → item∞

3. CONDITIONAL EQUATIONS

The result of an and-relation is functionally determined by the values of the other arguments, and the relation will only be used in the mode to find this argument, the relation can be defined as a function using conditional equations rather than clauses. This is not merely syntactic sugar; there are important differences between the evaluation of functions and relations.

Example program 6: quicksort function

We can reexpress our 'quicksort' program as a function, as follows:

\begin{verbatim}
function quicksort(?)
  quicksort([]) = [] .
  quicksort([u|x]) = append(quicksort(x1),[u|quicksort(x2)])
                 if partition(u,x,x1,x2)

function append(?,?)
  append([],y) = y .
  append([u|x],y) = [u-append(x,y)]
\end{verbatim}
The 'quicksort' definition makes use of the 'partition' and-relation which we defined earlier. It also uses a functional definition of 'append'.

Example program 7: primes

This program is a definition of the function 'primes' which evaluates to the (infinite) list of all prime numbers. It uses the familiar "sieve of Eratosthenes" algorithm.

```
function primes()
primes() = sift(intsfrom(2))

function sift(?)
sift([u|x]) = [u|sift(filter(u,x))]

function filter(?,?)
filter(u,[v|x]) = filter(u,x) if divides(u,v) |
filter(u,[v|x]) = [v|filter(u,x)] if ¬ divides(u,v) |

function intsfrom(?)
intsfrom(u) = [u|intsfrom(u+1)]
```

Syntax of function definitions

A function definition consists of a function declaration followed by an equation group.

A function declaration includes the function name (which begins with a lower case letter) and a list of '?' symbols to indicate the number of arguments of the function:

```
function f(?,?,...,?)
```

An equation group is of the form

```
<equation unit> |
<equation unit> . <equation group> |
<equation unit> ; <equation group>
```

An equation unit is

```
(<equation group>) |
<equation>
```

An equation is of the form

```
LHS = RHS if G ; B
```

where LHS and RHS are terms. The guard G and body B are relational expressions.

We now introduce a new kind of term, the evaluable term. This is written as f(t1,...,tn) where f begins with a lower case letter and t1,...,tn (n>0) are terms. If n=0 the parentheses must still appear, to distinguish the term from a variable.

So the full definition of term is that a term is a number, a variable, a
tuple, a constructor function term of the form \( F(t_1, \ldots, t_n) \) or an evaluatable term of the form \( f(t_1, \ldots, t_n) \), where the \( t_i \) are any terms.

Our previous definition of term identified a special subclass of term which we shall call a **data term**. Data terms are terms that do not include any evaluatable terms. Variable free data terms are PARLOG data structures.

**Restriction on equations**

The LHS of an equation must be an evaluatable term with the same name and number of arguments as the heading function declaration, and the arguments must be data terms. There is no restriction on the type of term that can appear on the RHS of an equation.

**Operational semantics**

An evaluatable term (a function call) is evaluated by a series of rewrite steps determined by the function definitions in the program.

A single rewrite step for a function call \( f(t_1, \ldots, t_n) \) consists of the following actions:

1. Find the applicable equation among the equation group for function \( f \). There must be exactly one such equation.

2. Replace the function term by the RHS of the applicable equation, and at the same time begin the parallel evaluation of the relational expression comprising the body of the equation. The evaluation of this relational expression must succeed.

The search for the applicable equation, in step 1, is controlled by the '.' and ';' structure of the equation group for \( f \):

Equation group has the structure  \( EU \; EG \)

Search for an applicable equation concurrently in \( EU \) and \( EG \). There should not be more than one; any other equations should eventually be inapplicable.

Equation group has the structure  \( EU \; EG \)

Search for an applicable equation in \( EU \). If all equations are inapplicable then search in \( EG \).

An equation of the form \( LHS = RHS \) if \( G : B \) is applicable if the LHS matches the evaluatable term and the guard relational expression \( G \) succeeds. An equation is inapplicable otherwise.

If there is no applicable equation, the evaluation is aborted.

An equation is input suspended if the match of the LHS with the call would result in a variable of the call being instantiated, or result in an attempt to unify a data term in the LHS with an evaluatable function term in the call. Both a variable and an evaluatable function term in the call represent values for arguments of the call that have yet to be computed. The evaluatable function term is an "inner" call which has not yet been expanded.
Evaluable terms as relation arguments

Evaluable terms may be used in the guards and bodies of clauses anywhere where a data term is allowed. However, as with equations, evaluable terms must not appear in input arguments in the head of a clause, i.e. those arguments corresponding to a '?' in a relation declaration for that relation.

This gives us one more condition for a clause to be input suspended. It is input suspended if there is an attempt to unify a data term in the head with an evaluable term in the call.

Eager and lazy evaluation

The default evaluation mode of a function call is eager, i.e. all of the function call arguments are evaluated in parallel with their values consumed by a concurrent evaluation of the top level function call. When function calls appear as arguments to relations, the default evaluation mode is also an eager evaluation. The function calls in the arguments of the relation are evaluated in parallel with their values consumed by a concurrent evaluation of the relation call.

Consider the query

: printlist(primes()) .

which causes the infinite list of primes to be printed. Here 'printlist' is a primitive relation that checks that its argument is a list and displays each element as it is available. For this query, primes() will be evaluated eagerly and its output list of integers will be printed as it is incrementally produced.

Alternatively a function may be evaluated lazily, in which case the rewrite steps are performed in a demand driven, outermost first order.

A lazy function call is only rewritten when its value is needed by an input suspended equation or an input suspended clause. In other words there has been an attempt, in the evaluation of some other call, to unify the lazy function call with a data term. In contrast, an eager function call is rewritten as soon as there is an applicable equation.

A lazy evaluation is specified by a '!' annotation following the function name in the evaluable term. For example:

: printlist(primes!()) .

Here, the 'primes' function will not be evaluated until triggered by a demand from 'printlist'. When primes() evaluates to a list [2,...] it will suspend until the next element of the list is demanded. Figure 4 shows a trace of this lazy evaluation.
primes()
sift(intsfrom(2))
sift([2|intsfrom(3)])
[2|sift(filter(2, intsfrom(3)))]
SUSPEND
[2|sift(filter(2, [3|intsfrom(4)]))]
[2|sift([3|filter(2, intsfrom(4))])]
[2,3|sift(filter(3, filter(2, intsfrom(4))))]
SUSPEND
.
.
.

Figure 4

Notice that the lazy evaluation is caused by an annotation in the call rather than by the use of a special constructor function in the definition, as in lazy LISP [FrW 76] and HOPE [BMS 80]. This means that the same function definition can be used in either an eager or a lazy evaluation. The lazy evaluation mode is inherited. All function calls generated by a lazy function call are themselves lazy.

Termination

Another advantage of the use of the functional notation is that the evaluation of a function call can be correctly terminated when there are no references to it, i.e. when no other call needs its value. This is due to the fact that in first order logic functions are assumed to be total, i.e.

\[ \forall x_1... \forall x_n \exists y: y = f(x_1,...,x_n) \]

When combined with lazy evaluation this allows one to program with infinite functions without incurring any overhead. For example, we can obtain the first 100 primes by the query

: printlist(first(100,primes!)) .

where the 'first' function is defined as follows:

function first(?,?)
first(0,x) = [] .
first(n,[u|x]) = [u|first(n-1,x)] if n>0 |

The logic programming system of Hansson et alia [HHT 82] also includes functions. In their paper they discuss the benefits of using functions for handling potentially infinite data structures.

Comparison with other languages

The functional language outlined in this section has some features in common with HOPE [BMS 80] and Turner's KRC [Tur 81].

PARLOG, like KRC, is untyped whereas HOPE has strong polymorphic typing. Both of these languages have a much more general higher order capability than PARLOG. On the other hand, PARLOG allows quite general control of the mode of evaluation of functions. KRC only allows lazy evaluation, while in HOPE the only lazy function is one which evaluates to a list.

18
4. OR-RELATION DEFINITIONS

For a query involving only and-relations, it is not possible to obtain more than one solution due to the don't care non-determinism. This makes it unsuitable as a database language. However, we can get all solutions to a query by means of a set expression that refers to or-relations. A set expression is a special case of an evaluable term. It can appear in any place that an evaluatable term can appear.

Example program 8

The relation 'append' can be defined as an or-relation as follows. This definition can be used to find all solutions to the relation.

or-relation append
append([],[],[])
append([u|x],y,[u|z]) :- append(x,y,z)

We can now define an and-relation allsplits(s,t) which means t is a list of all possible pairs of lists resulting from splitting list s:

relation allsplits(?;?)
allsplits(s,t) :- t=[<x,y>:append(x,y,s)]

In the call allsplits([1,2],z) z will be computed as the list

[[[],[1,2]],[[1],[2]],[[1,2],[]]]

or the list

[[[1],[2]],[[],[1,2]],[[1,2],[]]]

or any other list containing the same tuples in some order. The order of the tuples in the list is not determined by the program.

Another use of the or-relation 'append' is to split a list on a given element. In the call

append(x,[3|y],[1,2,3,4,5])

x is bound to [1,2] and y to [4,5]. This is a use that would flaunt the mode constraints if 'append' were defined as an and-relation because the argument [3|y] is partly input and partly output.

Single calls to or-relations, like the above, can be used in and-relation definitions and queries. The values assigned to the variables are the values of the first found solution. However, since it is an or-relation, the variable bindings of the solution are only made public when the complete solution is found (they are not incrementally made available).

Example program 9

The following two or-relations constitute a database which represents the departments of an organization and the employees in each department:
or-relation dept
dep(D1)
dep(D2)

or-relation employee
employee(D1,Smith)
employee(D1,Jones)
employee(D2,Brown)
employee(D2,Green)

The following term evaluates to a list of all departments and the employees in each:

{ <x,y> : dept(x) & y={z:employee(x,z)} }

i.e. [D1,[Smith, Jones]> ,D2,[Brown, Green]> ] or some permutation of this.

Syntax of or-relation definitions

An or-relation definition is a single or-relation declaration followed
by an or-clause group.

An or-relation declaration is of the form

or-relation <relation name>

Notice that there is no mode in the declaration. Or-relations can be
called in any mode. The definition is compiled in such a way that it can
be used to find all solutions to any call.

An or-clause group is of the form

<or-clause> |<or-clause> <or-clause group>

An or-clause is of the form

<atom> |<atom> :- <sequential expression>

A sequential expression is

<literal> |<literal> & <sequential expression>

Finally, a set expression takes the form

{ <term> : <sequential expression> }

Declarative reading

A set expression

{ T : SE }

is read:
a list of all T such that
SE for some x1,...,xk

where x1,...,xk are all the local variables of the sequential expression
SE. These are the variables that only appear in SE. The global variables
of SE are the variables that occur outside the set expression within the
clause or query in which it is used. The global variables must have been
assigned values before the expression is evaluated.

A major difference between or-relations and and-relations is that there
are no guards in the clauses and no separators between clauses of or-
relations. An explicit separating connective is not needed, since the way
in which the different clauses are tried in order to find all solutions to a
call is determined by the mode of evaluation of the set expression in
which the call appears. This will be explained further below. Guards are
not needed, since all clauses that match a call are deemed candidate
clauses which may produce a different solution to the call. Finally, calls
to and-relations can be used in set expressions and in the bodies of or-
relation clauses. When they are evaluated, only one solution is returned.

Operational semantics

Or-relations are usually called from set expressions. We therefore give
the operational semantics for set expressions.

The evaluation of a complex set expression

\[ s = \{ T : L \& SE \} \]

where T is a term, L is a literal and SE is a sequential expression, is
first reduced to the evaluation of the simple set expression

\[ u = \{ V : L \} \]

where V is the tuple of variables in L.

If u evaluates to the empty list then s is empty. If u contains p
solutions V1,...,Vp (p\geq1), where each Vi is a tuple of terms giving
solution bindings for the V variables in L, then the value of s is found by
the evaluation of

\[ s = \text{merge}( \{T[V/V1]:SE[V/V1]\}, ..., \{T[V/Vp]:SE[V/Vp]\} ) \]  (1)

Here, 'merge' is a generalization of the and-parallel relation defined
above, used as a "non-deterministic function". It evaluates to a list
resulting from merging p input streams. Duplicate solutions are not
removed.

T[V/Vi] and SE[V/Vi] are substitution instances of T and SE in which
variables of V are replaced by their corresponding bindings in Vi. As each
solution Vi in u is found, the evaluation of the corresponding (i-th) set
expression argument of (1) is started. The solution lists of all p set
expressions are merged into the list s in the order in which they become
available.

Intuitively, the first condition L of the sequential conjunction L \& SE
becomes a generator of values that are passed on to the rest of the
conjunction as soon as they are found. The evaluation of SE is not started
until the first solution of L is found. Each found solution to L causes a
new parallel evaluation of SE to be immediately started. The solutions of all of these parallel evaluations of \{T:SE\}, modified in accordance with some solution to L, are merged into a single solution list s as and when they are found.

To evaluate a simple set expression

\[ s = \{ T : R(t_1,...,t_n) \} \]

all of the candidate clauses are found by a parallel search of the clauses for R. The candidate clauses are the clauses whose heads match the atom \( R(t_1,...,t_n) \). If there are no candidate clauses then s is empty. If there are q candidate clauses:

\[ H_1 :\text{-} CB_1 \\
... \\
H_q :\text{-} CB_q \]

q\(\geq 1\), then s is found by evaluating

\[ s = \text{merge} (\{T(1):CB_1(1)\}, ..., \{T(q):CB_q(q)\}) \quad (2) \]

where \( T(i) \) and \( CB_i(i) \) are substitution instances of \( T \) and \( CB_i \) resulting from the unification of \( R(t_1,...,t_n) \) with clause head \( H_i \).

If a clause body \( CB_i \) is empty, the i-th argument of (2) is \( \{T(i)\} \) which evaluates to the singleton list \( \{T(i)\} \).

All q set expression arguments of (2) are evaluated concurrently and their solution lists are merged into list s as soon as they become available.

Finally, to evaluate a negative set expression

\[ s = \{ T : \neg R(t_1,...,t_n) \} \]

the set

\[ s' = \{ <> : R(t_1,...,t_n) \} \]

is evaluated. As with negated calls in and-relation definitions, the atom \( R(t_1,...,t_n) \) must be completely instantiated. If not, the evaluation aborts with an error.

Then

\[ s = [T] \text{ if } s' = [] \]
\[ s = [] \text{ if } s' = [<>] \]

Again this is an application of the "negation as failure" rule used to evaluate negated conditions for and-relations.

Lazy set evaluation

As described above, a set expression is evaluated in a highly parallel eager manner. Alternatively, a PROLOG-like lazy evaluation can be specified by writing a '!' annotation after the set brackets:

\[ s = \{ T : L \text{ & SE } \}! \]
As with the eager evaluation, this is first reduced to the evaluation of the simple set expression

\[ u = \{ V : L \}! \]

and \( s \) is then found by the evaluation of

\[ s = \text{append}( \{ T[V/V1]:SE[V/V1] \}!, \ldots, \{ T[V/Vp]:SE[V/Vp] \}! ) \] (1')

where \( V1, \ldots, Vp \) are the solutions in \( u \), and 'append' is a generalization of the binary 'append' function defined earlier.

Now, as each solution \( Vi \) in \( u \) is found, the evaluation of

\[ \{ T[V/Vi]:SE[V/Vi] \}! \]

is started only if the evaluation of

\[ \{ T[V/Vi-1]:SE[V/Vi-1] \}! \]

has finished. This means that on the list \( s \), all solutions to \( SE \) generated by the \( i \)-th solution of \( L \) appear as a contiguous sublist of elements sandwiched between the solutions to \( SE \) generated by the \( (i-1) \)-th solution to \( L \) and those generated by the \( (i+1) \)-th solution to \( L \).

The evaluation of a simple set expression

\[ s = \{ T : R(t1,\ldots,tn) \}! \]

is reduced to the evaluation of

\[ s = \text{append}( \{ T(1):CB1(11) \}!, \ldots, \{ T(q):CEq(q) \}! ) \] (2')

where \( T(i) \) and \( CBi(i) \) are the substitution instances of \( T \) and of the body of the \( i \)-th candidate clause for the call.

Because of the way 'append' is defined, the lazy set expression arguments of (2') are evaluated sequentially in left to right order and their solutions are merged into the list \( s \) in this order. In other words, all of the solutions given by the first candidate clause precede the solutions given by the second candidate clause and so on. Moreover, solutions given by the second candidate clause are only sought when all solutions given by the first have been found. The candidate clauses are ordered by the before-after order in which they appear in the program.

The evaluation of a negative set expression

\[ s = \{ T : \sim R(t1,\ldots,tn) \}! \]

is converted to the evaluation of the lazy set

\[ s' = \{ \langle\rangle : R(t1,\ldots,tn) \}! \]

Then, as for the eager evaluation method,

\[ s = [T] \text{ if } s' = [] \]

\[ s = [] \text{ if } s' = [\langle\rangle] \]
Evaluation of a single call

A single call $R(t_1,\ldots,t_n)$ to an or-relation outside a set expression is converted to an evaluation of the lazy set

$$\{ \text{V : } R(t_1,\ldots,t_n) \}!$$

where $\text{V}$ is a tuple of the variables of the call. The evaluation will be terminated as soon as the first solution is found and the variables of $R(t_1,\ldots,t_n)$ are bound to this found solution.

The evaluation of the single call therefore corresponds to the sequential backtracking evaluation of PROLOG. The difference is that once a solution is found, there is no backtracking to find an alternative solution. PROLOG's use of backtracking to search for a solution to a conjunction of calls must be achieved in PARLOG by using a lazy set expression to find the list of solutions to the conjunction of calls.

Example use of lazy sets

We can invoke a lazy, sequential evaluation of the 'allsplits' program from example program 8 by using the '!' annotation:

$$\text{allsplits}(s,t) :- t=[\langle x,y\rangle: \text{append}(x,y,s)]!$$

In this case a call $\text{allsplits}([1,2],z)$ will compute $z$ as the list

$$[[[],[1,2]],[[1],[2]],[[1],[]]]$$

Here, the solution tuples are guaranteed to appear in the order determined by the order of the 'append' clauses in the program.

The evaluation of

$$\{ \langle x,y \rangle : \text{employee(D2,x)} \& \text{salary(x,y)} \& y>10000 \}!$$

will be a PROLOG style backtracking sequential search.

Comparison

Set expressions have long been included in PROLOG systems, such as DEC-10 PROLOG [War 82] and IC-PROLOG [CMG 82].

Our use of set expressions compares with Turner's KRC [Tur 81] and the functional database language of Buneman and Nikhil [BuN 81].

5. METALEVEL FEATURES

Higher order functions

A metacall facility in PARLOG allows a form of higher order programming: a metavariable $f$ denoting a function name can be applied to arguments $x_1,\ldots,x_n$ using the primitive operator '$\theta$', as in the expression $f\theta(x_1,\ldots,x_n)$. This is illustrated by the following definition of 'f_maplist':

24
function f_maplist(?,?)
f_maplist(f,[]) = [].
f_maplist(f,[u;x]) = [fθ(u)|f_maplist(f,x)]

The term fθ(u) is a function application. f is a variable which must be bound to a constant naming an evaluable function.

To call 'f_maplist', with the first argument the name of a 'sqrt' function for example, we give the name of the function as a constant:

f_maplist('sqrt',[1,4,16,36])

which evaluates to [1,2,4,6]. The metacall fθ(u) will be equivalent to a function call sqrt(u).

A similar feature is provided for relations. For example, the relation 'r_maplist' takes a list and a constant naming a relation and produces a list of the same length:

relation r_maplist(?,?,?,^)
r_maplist(r,[],[]).
r_maplist(r,[u|x],[v|y]) :-
            rθ(u,v) & r_maplist(r,x,y)

We can use this to compute a list z of square roots:

r_maplist('sqrt',[1,4,16,36],z^)

This time, the metacall rθ(u,v) will be equivalent to a relation call sqrt(u,v) in mode (?^). The compiler computes the mode of a metacall in the same way as for a normal relation call.

Program manipulation

In common with most logic programming languages, PARLOG has primitives that can be used to manipulate programs. These facilitate the writing of front-end systems in PARLOG to aid the construction of PARLOG programs. They can also be used to manipulate or-relation definitions used as databases.

For example, the following top-level query command will add the definition of a new or-relation 'salary'. This relation defines the salary of each employee:

: addcl(salary(Smith,8560)) &
   addcl(salary(Jones,9800)) &
   addcl(salary(Brown,10150)) &
   addcl(salary(Green,12010)) .

To delete a clause, say the second clause for the 'employee' relation, we can use the query command

: delcl('employee',2) .

or the equivalent

: delcl(employee(D1,Jones)) .

The clauses added to and deleted from an or-relation are not restricted

25
to assertions; they can be any clauses.

Or-relations can be used as 'global' data structures that are destructively manipulated using 'addcl' and 'delcl'. This is the non-applicative component of PARLOG. Uses of 'addcl' and 'delcl' are the only forms of destructive assignment. As with all assignment, it should be used in a controlled way to preserve the comprehensibility of the program. An example of the controlled use of a database or-relation is given in example program 10 below.

Example program 10: set

This is yet another program to implement the 'set' relation of example program 3. Each member of the set is represented by a clause for a 'member' relation. The set is updated by addition and deletion of clauses.

relation set(?)
set(c) :- kill('member') & set1(c)

relation set1(?)
set1([Add(m);c]) :- member(m) | addcl(member(m)) & set1(c).
set1([Delete(m);c]) :- member(m) | delcl(member(m)) & set1(c).
set1([In(m);c]) :- member(m) | set1(c).
set1([In(m);c]) :- member(m) | notfound(m) & set1(c).

The sequential 'and' ('&') is essential here to ensure the sequential modification of the shared database.

6. A SIMPLE OPERATING SYSTEM

The following example illustrates most of the features of PARLOG. It shows how a UNIX-style operating system can be implemented in PARLOG.

Consider the operating system depicted in Figure 5. It comprises a main 'system' process connected to a 'filestore' process, a 'spooler' process and two 'term' processes via a 'merge' process. Each 'term' process services a terminal user, and is connected to a 'keyboard' and a 'screen' process which produce and consume lists of characters respectively. The 'keyboard' process is a lazy function: it only sends characters to the 'term' process when prompted.
The dotted lines in the above diagram indicate back communication.

The concurrent evaluation of the system is represented by the PARLOG parallel conjunction:

: screen(s1), screen(s2),
  term(keyboard(1), s1^, m1), term(keyboard(2), s2^, m2),
  merge(m1, m2, m), system(m, fs, p), filestore(fs), spooler(p).

The 'keyboard' function and the 'screen' relation are assumed as primitives. All other relations are defined by PARLOG programs.

The 'term' process comprises the 'shell' command processor and a 'merge' process which merges the 'shell' responses with the keyboard input to form the screen output of 'term'. The 'shell' also produces m, a stream of messages to the system. This is the stream of messages that 'term' sends to the system.

relation term(?, ^, ^)
  term(k, s, m) :- shell(k, s1, m), merge(k, s1, s)

This clause can be read: m is the sequence of system commands and s is the display character sequence for input sequence k if m is the sequence of system commands generated by the shell process, s1 is the sequence of characters of shell responses, and s is a merging of s1 and k.

The 'shell' process parses the input list of characters into a list of commands and data strings and passes these to the 'cshell' relation which actually does the work and generates the shell responses and the system commands.

relation shell(?, ^, ^)
  shell(k, s, m) :- cmds(k, c), cshell(c, s, m)

Below, we describe how the input character sequence is parsed into a sequence of shell commands and how these commands are processed by
'cshel'.

The user command language

We will use a Unix-like command syntax as follows.

A user command consists of a sequence of characters giving up to four command fields: program name(s), input specifier, output specifier and an optional '&' to indicate that the command is to run in the background. The command is terminated by a cr.

If more than one program name is given, the list of program names is written:

P1 | P2 | ... | Pn

This indicates that they are to be run concurrently with the output of each program Pi fed to Pi+1.

The input specifier indicates the source of the input to the first program in the list. There are three possibilities. If absent then there is no input. '<F' means that input is the file with name 'F'. The '|' symbol on its own (i.e. followed by a separator character) means that input is from the "standard" input, i.e. it is the string of characters following the command and terminated by the next EOF character. Normally, standard input would not be specified in the command in this way. This feature is introduced here in order to simplify our example.

There are two kinds of output specifier. '>F' means that output is stored in a file named 'F'. If the output specifier is omitted, the output is "standard", i.e. sent to the same stream as the responses to the commands.

Let getcmd(x,u,z) be the relation: u is the parsed form of the command at the front of string x and z is the remainder of x following the command. This representation of an initial sequence as the difference between one character list and another is the standard method for representing parsing programs in logic.

We will use a string notation for lists of characters, e.g. "PRINT" stands for the list [P,R,I,N,T]. The following are some instances of the 'getcmd' relation:

getcmd("PRINT <ForP|Q < ...",
     "([PRINT],FILE(F),STD,SEQ),
      "P|Q < ...")

getcmd("P|Q <orR >G & ...",
     "([P,Q],STD,STD,SEQ),
      "R >G & ...")

getcmd("R >G &or...",
     "([R],NONE,FILE(G),PAR),
      "...")

In the first of these examples, the tuple

"([PRINT],FILE(F),STD,SEQ)"

26
is the parsed form of the substring "PRINT <For" that begins the entire command sequence string. The remainder of this command sequence string is "P|Q < ..." which is the third argument of the 'getcmd' relation.

In general, the parsed form of each command is a four-tuple of the form

< [piped-processes-list, input-file, output-file, mode-indicator]>

The mode-indicator is 'PAR' if the command is to be run as a concurrent background process. It is 'SEQ' if it is to be run as a foreground process and completed before any later command is commenced.

As an example, consider the following character string:

"PRINT <ForP|Q <cr>crWYcrZeofDIRcr...

The 'cmds' relation must parse this into the following command sequence:

[<[PRINT],FILE(F),STD,SEQ>,
 <[P,Q],STD,STD,SEQ>,
 "WXYcrZ",
 <[DIR],NONE,STD,SEQ>, ...
 ]

The "WXYcrZ" is the data for the sequentially executed second command. The end of this data is marked by the eof.

The 'cmds' relation can be defined in terms of such a 'getcmd' relation and a similar 'getdata' relation as:

relation cmds(?,^)
cmds([],[]).
cmds([v|x],[u|y]) :-
    getcmd([v|x],u,z), getdata(u,z,z1,y,y1), cmds(z1,y1^)

'cmds' parses a command from the front of the input character sequence and adds this to the command sequence. If the command requires standard input, 'getdata' is used to find any data sequence of characters that follows the command and to add them as the next element of the command sequence. getdata(u,z,z1,y,y1) means that the data for command u is the sequence of characters of z up to z1 and that this data sequence is the difference between command sequence y and command sequence y1. 'getdata' is defined as follows:

relation getdata(?,^,^,?,?)
getdata(v,NONE,w,x>,z,z,y,y) .
getdata(v,FILE(f),w,x>,z,z,y,y) .
getdata(v,STD,w,x>,z,z1,[u|y1],y1) :-
    append(u,[eof|z1],z)

'append' is the or-relation of example program 8. The call returns the first solution found by a backtracking evaluation. Thus u is the front segment of z up to the first eof character.

The 'getcmd' relation can be similarly defined using 'append'. We leave it as an exercise for the reader in logic program parsing.

We must now define the 'cshell' command evaluation relation. We lead up to the general definition that can deal with any sequence of commands by first defining a more restricted relation that can only handle three types of command. Its definition has three clauses.
relation cshell(?,^,^)

cshell([<\{p1\},NONE,STD,SEQ>|c],s,[Get(p1,pr1?)|m]) :-
  ( execute(p1,pr1,\[],s1,m1) & cshell(c,s2,m2) ),
  merge(s1,s2,s), merge(m1,m2,m)

This first clause deals with a single command to be run in the foreground. A system message 'Get' is generated to obtain the program file named by the command. This program is executed with an empty input list, due to the 'NONE' specifier. When the execution terminates, the 'cshell' process resumes. Two 'merge' processes are run concurrently with this so that any system messages or user responses generated by either 'execute' or 'cshell' are passed on immediately.

Notice the back communication signalled by the '?' annotation on pr1. This tells the compiler that pr1 is input to the 'execute' call. Its value will be given by the file server that picks up the 'Get' message.

cshell([<\{p1,p2\},STD,STD,SEQ>,st|c],s,
  [Get(p1,pr1?),Get(p2,pr2?)|m]) :-
  ( ( execute(p1,pr1,st,x,m1), execute(p2,pr2,x,s1,m2) ) &
    cshell(c1,s2,m3) ),
  merge(s1,s2,s), merge(m1,m2,m3,m) )

This second clause is for a foreground pair of piped commands. The input to the first command p1 is the string following the command in the command sequence. The output of p1 is a "pipe" channel which is input to p2. When both processes have terminated, 'cshell' resumes. We have used a general 'merge' relation, with any number of arguments.

cshell([<\{p1\},FILE(ifn),FILE(ofn),PAR>|c],s,
  [Get(p1,pr1?),Get(ifn,if?),Replace(ofn,of)|m]) :-
  execute(p1,pr1,ifn,ofn,m1), cshell(c,s,m2),
  merge(m1,m2,m)

This last clause deals with a background command which uses files for input and output. System messages are generated to 'Get' the program file and the input file and to 'Replace' the output file in the filestore. The execution of the command proceeds in parallel with the continuation of 'cshell'. Again, the annotation on if tells the compiler that if is input to the 'execute' call.

If we assume that a program name is the name of some key relation defined in the program the call execute(p,pr,in,out,m) is equivalent to p(in,out,m) where p is the relation defined by program pr.

If we assume a primitive 'addprogram' relation that adds a set of definitions to the run time system, then 'execute' can be defined as

execute(p,pr,in,out,m) :- addprogram(pr) & p@\(in,out,m)\)

which makes use of the metalevel 'apply' relation to execute the added program. Notice that the relation p must be defined as a relation that consumes a string of characters (the in argument) and which generates a string of characters (the out argument) and an optional list of system messages.

The general definition of 'cshell' is as follows. As a simplification, we assume that all files required are present in the filestore. Without
this assumption, we would need to have additional clauses to deal with 'NOTFOUND' replies from the filestore.

'cshell' has one clause for a background command and another for a foreground command. Each clause has four main parts: connecting the input 'if' to the specified source, connecting 'of' to the specified output destination, running the command program(s) and resuming the 'cshell'. The only difference between the two clauses is that in the background case the commands are executed concurrently with the 'cshell' while foreground commands are executed before control returns to 'cshell'.

\[
\text{relation cshell}(?,^,^) \\
cshell([],[],[]). \\
cshell([\langle ps, input, output, PAR\rangle \{ c \}, s, m]) :- \\
\text{progs}(ps,prs,m1), \\
\text{infil}(input,if,c,c1,m2), \\
\text{outfil}(output,of,s1,m3), \\
\text{run}(prs,if,of^,m4), \\
cshell(c1,s2,m5), \\
\text{merge}(s1,s2,s), \\
\text{merge}(m1,m2,m3,m4,m5,m). \\
cshell([\langle ps, input, output, SEQ\rangle \{ c \}, s, m]) :- \\
\text{progs}(ps,prs,m1), \\
\text{infil}(input,if,c,c1,m2), \\
\text{outfil}(output,of,s1,m3), \\
( \text{run}(prs,if,of^,m4) \& \text{cshell}(c1,s2,m5) ), \\
\text{merge}(s1,s2,s), \\
\text{merge}(m1,m2,m3,m4,m5,m) \\
\]

The 'progs' relation obtains from the filestore the program files named in the list of commands.

\[
\text{relation progs}(?,^,^) \\
\text{progs}([],[],[]). \\
\text{progs}([\langle ps, [\langle p, pr\rangle \{ prs \}, [\text{Get}(p,pr?)\{ m \}] \} :- \\
\text{progs}(ps,prs,m) \\
\]

The 'infil' relation definition has three cases. If the command input specifier is 'NONE', the input file is the empty list. If 'STD' is specified, the input file is the next string in the parsed command sequence. For a 'FILE' specifier the input file is obtained by a filestore message to the system.

\[
\text{relation infil}(?,^,?,^,^) \\
infil(\text{NONE},[],c,c[,[]). \\
infil(\text{STD},if,[[if]\{ c \},c,[]]) . \\
infil(\text{FILE}(ifn),if,c,c,[\text{Get}(ifn,if?)]) \\
\]

The 'outfil' relation connects the output file to the list of responses if the output specifier is 'STD'. If it is 'FILE', the output file is placed in a 'Replace' message to the system.

\[
\text{relation outfil}(?,?,^,^) \\
\text{outfil}(\text{STD},of,of,[[]). \\
\text{outfil}(\text{FILE}(ofn),of,[[]),[\text{Replace}(ofn,of)]) \\
\]

The 'run' relation executes a list of command programs, passing the output of each command to the input of the next. Any system messages generated by the commands are merged into a single list.
relation run(?,?,^,^)
run([],if,if,[]).
run([<p,pr>|ps],if,of,m) :-
    execute(p,pr,if,x,m1),
    run(ps,x,of,m2),
    merge(m1,m2,m).

The complete system was described by

: screen(s1), screen(s2),
term(k1,keyboard(s1),m1), term(k2,keyboard(s2),m2),
merge(m1,m2,m), system(m,fs,p), filestore(fs), spooler(p).

We have defined the 'term' relation. It remains to define 'system',
'filestore' and 'spooler'. We can define the filestore as a set. The
'set' relation has been defined earlier:

relation filestore(?)
filestore(fs) :- set(fs)

Channel m is a list of messages of the forms Get(fn,f), Replace(fn,f),
Print(fn), p is a list of files to be printed. The 'system' process
passes 'Get' and 'Replace' messages to the filestore as 'In' and 'Add'
messages and also handles Print messages, which might be generated by
commands such as 'PRINT'. When the 'system' process receives a message of
the form Print(fn) it sends an In(<fn,f>) message to the filestore and
simultaneously sends f to the spooler. When f is bound to a file by the
filestore, this is communicated to the spooler.

relation system(?,^,^)
system([Get(fn,f)|m],[In(<fn,f?>)|fs],p) :- system(m,fs,p).
system([Replace(fn,f)|m],[Add(<fn,f>)|fs],p) :- system(m,fs,p).
system([Print(fn)|m],[In(<fn,f?>)|fs],[f|p]) :- system(m,fs,p).

The 'spooler' relation was discussed in detail in [CLG 81]. For
example, a spooler with two printers can be defined as follows. The
argument list of files p is split into two lists p1 and p2 such that no
more than one file will accumulate on the input queue for either printer.

relation spooler(?)
spooler(p) :- merge(p1~1,p2~1,p), printer(p1), printer(p2)

6. CONCLUDING REMARKS

Required architectures

Our original paper [CLG 81] defined a language with a restricted amount
of parallelism and channels restricted to streams, which is suitable for a
loosely coupled parallel architecture. The more powerful version of PARLOG
described here requires a tightly coupled architecture, allowing many
parallel activities.

Work in progress

The and-relations and functions of PARLOG, described in sections 2 and
3, have been implemented on the ALICE machine [DaR 81]. This is a
reduction machine designed primarily for functional languages and has
proved to be well suited to PARLOG. The intention is to use PARLOG to implement the higher level components of the ALICE operating system.

This pilot implementation consists of a compiler from PARLOG to an abstract machine code [Gre 83a] and then to ALICE CTL (Compiler Target Language), together with a small run time system [Gre 83b]. The compiler is itself written in PARLOG and also in micro-PROLOG [Hoc 83] and MPROLOG [SZK 82]. It is hoped to eventually bootstrap the PARLOG version on the ALICE simulator.

We are also beginning to investigate the implementation of PARLOG on other parallel architectures such as the Manchester dataflow machine [GuW 80].

Related work

Several other systems have recently been developed for the parallel execution of logic programs. These differ in the degree of parallelism and the language features, if any, provided to control the concurrency. Perhaps the most important difference between parallel logic programming systems is the way in which they handle and-parallel calls which share variables.

The system closest to PARLOG is Shapiro's Concurrent PROLOG [Sha 83]. This is similar to the and-relation subset of PARLOG in its use of don't care nondeterminism. Every clause has a guard and a body and no more than one clause is selected to solve a call.

The principal difference between PARLOG and Concurrent PROLOG is that the latter uses a non-inherited read-only annotation on variables instead of PARLOG's inherited modes, which are not used in Concurrent PROLOG. The great advantage of this is that partially instantiated messages need no special treatment (such as PARLOG's head annotations). The read-only annotation has been used in several other systems, such as IC-PROLOG [CNM 82], EPILOG [Wis 82] and the language of Dausmann et alia [DFW 79]. It is a lower level concept than the inherited direction of communication. Consequently, although it is powerful, great care must be taken to include the annotation in the correct places. For this reason Concurrent PROLOG programs are such that they can usually only be used in one mode (although if back communication is used this is also true of PARLOG).

It is worth noting that Concurrent PROLOG is being used by the Japanese ICOT to define the operating system of their Sequential Inference Machine.

Wise's EPILOG is another system which makes use of annotations to control parallel execution. EPILOG includes a variety of such annotations.

Most other systems do not make use of don't care non-determinism. Con Par's paper [CoK 81] described a system with or-parallelism and they have since extended it to allow limited and-parallelism. It appears that whenever calls share a variable, they are evaluated sequentially thus precluding stream communication. Their system is designed on the principle that the programmer should not have any control over the evaluation of a program. Instead, the order of evaluation of calls and clauses is determined automatically.

PRISM [KKM 83] consists of a language and an abstract architecture designed specifically for parallel logic program execution. The architecture has been implemented on a multi-processor machine ZMOB. The
language allows programmer control of sequential or parallel execution of calls and selection of clauses in a way similar to PARLOG. Or-parallelism and limited and-parallelism is implemented by PRISM but when and-parallel calls share variables, they are not evaluated concurrently but sequentially on the same processor.

Pollard's scheme [Pol 81] is altogether more ambitious than those mentioned above. His system is based on the completely parallel evaluation of the AND-OR tree for a program. Separate "reconciliation" processes run concurrently with this. These processes inspect the partial bindings of variables and detect any incompatibility between groups of nodes in the tree. As more incompatibilities are detected, some nodes may be found to be impossible (cannot contribute to a solution) and the evaluation of the subtree rooted at such nodes is aborted. This could be regarded as a parallel "join" which works on the partial solutions returned by conjoined calls.

ACKNOWLEDGEMENTS

We are grateful to the following people for helpful comments and discussions: Kryspia Broda, Tony Hecare, Frank McCabe, Jack Minker, George Pollard, Ehud Shapiro, David Warren.

The research described in this paper has been supported by the Science and Engineering Research Council.

REFERENCES


