PARLOG: a tutorial introduction. †

David Gilbert

Department of Computing
Imperial College
180, Queen's Gate
London SW7 2BZ
UK

ABSTRACT

Concepts connected with parallel logic programming are discussed in the light of "traditional" developments in the field of logic programming. The language used to illustrate these topics is PARLOG. Applications of PARLOG are introduced, in particular those of systems programming and object oriented programming.

1. Declarative Programming & Logic Programming

The present generation of computers which conform to the von Neumann type of architecture has begun to reach its performance limits, due basically to their sequential design. Early programming languages evolved to exploit this architecture, and became heavily reliant on the execution mechanism of the underlying machine. This was a constraining influence and inhibited their orientation towards the language users. Despite radical improvements in language design during the last two decades, resulting in languages such as Pascal, Ada and Modula, the underlying imperative nature of these languages which utilised destructive assignment remains unchanged. The semantics of this class of languages can only be understood by reference to the abstract or real machine on which the languages are to be executed.

Recent years have seen the design of languages which are free from the constrictions imposed by sequential architectures, and in so doing have stimulated the design of radical machine architectures. These declarative languages differ from imperative languages in that they are based on an abstract formalism, thus divorcing their semantics from the machine on which they are run. The statements in such a language are declarative since they can be understood without reference to machine level terms, such as side-effects. For this reason, programs in a declarative language can act as a specification, since they can be regarded as the formal description of a problem [Kowalski82]. Other advantages of such languages are that programs can be developed and tested bit by bit, and that they can be systematically synthesised or transformed [Hogger81].

First-order predicate logic provides the formalism for one branch of the declarative languages, namely logic programming. Statements in such languages are sentences in this form of logic, usually Horn clauses. Kowalski's work [Kowalski74] gave these a procedural interpretation; the first logic programming language to be implemented was Prolog, designed at Marseilles [Roussel75] and is widely used today.

---

† This article forms the basis for a talk given by the author at a seminar entitled "Current trends in Parallel Processing and Supercomputing", organised by the Belgian Institute for Automatic Control (November 19-20, 1987, Antwerp).

2. Logic programming, Prolog and PARLOG.

2.1. Committed choice parallel logic programming languages

PARLOG is a language which belongs to the family of committed choice parallel logic programming languages. Other such languages are Concurrent Prolog [Shapiro83a], and GHC [Ueda86]; all three are descended from the Relational Language [Clark81]. There are significant differences between PARLOG and Prolog. Committed choice non-determinism in PARLOG is implemented by the use of guards which ensure that committal is made to only one clause, unlike the backtracking ability of Prolog. Also the language can explicitly express both parallelism in OR and stream-AND forms, as well as sequentiality. These features enable the non-determinism inherent in the language to be exploited by the programmer, and permit the writing of programs which require the separate use of both sequential and parallel evaluations.

2.2. Execution strategies.

A pure logic programming language does not possess rules which detail the order of evaluation of predicates, and thus gives no indication of the manner in which the and-or tree determined by a computation should be traversed. Because such a language is non-deterministic, a logic program may possess a branched computation tree, and its execution may result in a variety of computations. Branching is due to

(i) The existence of several calls capable of activation.

(ii) The existence of several clauses capable of responding to a particular call.

In an implementation of a logic language, the execution of a logic program is governed by the control mechanism built into the program evaluator [Hogger84]. Thus, although the programmer has control over the general way in which the computations are developed, the program evaluator must manage the binding history. The standard strategy controlling the program evaluator's progress consists of combining the standard computational rule (call selection) with the standard search rule (clause selection). PARLOG and Prolog differ in the standard control strategy they employ.

The the standard computational rule of Prolog is said to be *left-to-right, depth first*, with *backtracking*. Prolog selects the first goal in textual order in each clause, and does not develop any other computations until the current one has been concluded. The search tree that is developed is expanded depth first, and if a terminal node with a failure is reached, then the program evaluator backtracks through the path in order to find the most recent node which gives rise to other unexplored branches. PARLOG's standard computational rule does not employ automatic left-to-right evaluation, calls in an AND-conjunction being evaluated in parallel; a depth-first strategy is not used to explore the search tree, but rather all possible paths are explored concurrently and the first successful answer terminates the search. Thus in a call:

? a(X) , b(Y).

PARLOG would evaluate the calls to a(X) and b(X) *concurrently*, while Prolog would attempt to find a solution to a(X) *before* attempting solutions to b(X).

The standard search rule employed by a sequential implementation of Prolog evaluates clauses within a relation in the order in which they appear in the text, using backtracking to provide an "all solutions" set of answers by attempting to completely traverse the search tree. In practice there are situations in which Prolog's search strategy is incomplete, a problem related to its left-to-right depth-first strategy. If a branch in a computational tree is infinite in depth, due for example to a non-terminating loop, then Prolog will not explore the branches to the right of that branch. The computation may be aborted when the computer runs out of workspace, or by intervention from the operator; alternatively some method may be implemented to enforce backtracking in such a case. Either way, the computation is incomplete in that the aborted path may eventually have terminated successfully.

The PARLOG program evaluator does not explore clauses within a relation definition in the strict textual order in which they are encountered, but chooses *one* at random to evaluate, and only pursues the paths which arise from that computation. The search strategy of PARLOG is thus incomplete, but preserves partial correctness. In this sense, PARLOG employs "don't care" non-determinism, rather than the "don't

know" non-determinism that backtracking brings to Prolog.

Consider the following logic program:

```
a <- a.
a.
```

and the query

```
? a.
```

Prolog would attempt solve the first clause textually and would enteran infinite loop, *never* returning an answer. PARLOG non-deterministically commits to one of the clauses, and thus *could* eventually return a successful answer. Recently progress has been made in the implementation of Or-parallel Prologs in an attempt to avoid the bottleneck of the sequential search rule, and thus benefit from speedup in execution [Warren87].

2.3. The syntax of a PARLOG program

2.3.1. Explicit AND operators

The PARLOG programmer may use AND operators which are explicitly parallel or sequential. The parallel AND conjunction operator is the comma ',' and the sequential-AND operator is '&'. These two operators may be used together in the same clause, and bracketing can be employed in order to clarify which calls are to be explored in parallel, and which in sequence. Thus the calls in the query

```
? p , q
```

will be evaluated in parallel, but in the case of

```
p & q
```

q will only be evaluated after the successful completion of the call to *p*. In the query:

```
( p , q ) & r
```

the calls to *p* and *q* are executed in parallel, and only after the successful completion of both is the call to *r* executed. The sequential '&' is useful in several instances. These may include, for instance, the correct ordering of input and output (programming with side effects) and the control of the degree of parallelism in algorithms.

2.3.2. Explicit OR searches

OR searches may be designated as being explicitly either parallel or sequential. Clauses which are to be searched in parallel are terminated with a period, and those to be evaluated sequentially are terminated with a semi-colon ';'. If the sequential and parallel OR search operators are used in the same predicate, clauses to searched using the same strategy are separated by the relevant operator, the flow of control being read from the first clause in textual order. The last clause of a predicate is always terminated by a period. PARLOG's sequential OR search strategy is similar to that of Prolog in that clauses are tried in textual order. Thus

```
p ;
p ;
p ;
p.
```

are to be tried in the sequence 1,2,3,4 but

```
p .
p .
p ;
p.
```

designates that the first three clauses are to be attempted in parallel, and if none of these succeeds then

clause 4 is to be explored. The colon at the end of clause 3 separates the subsequent clause from the parallel OR-search of 1, 2 and 3.

2.3.3. Clausal Form

The general form of a PARLOG clause is:

<head> <- <guard> : <body>

Note that <head> is in the form *name(a1,..,ak)* , where *name* is the relation name, and *a1,..,ak* its arguments. The logical implication symbol is ' <-' and ':' the guard operator. Both the guard and the body can be a conjunction of calls, or empty, and the calls are separated by the sequential-AND or parallel-AND operators. By convention, if a clause has a guard but no body, the constant **true** is takes the place of the missing body, thus PARLOG clauses may also have the form:

<head> **<-** <body> **.**
<head> **<-** <guard> **: true .**

2.4. Forms of parallelism in logic programs, with reference to PARLOG.

An implementation of a logic language may allow parallel evaluation in the form of And-parallelism (the computational rule where calls are evaluated concurrently), and as Or-parallelism (the search rule where clauses are tried concurrently).

2.4.1. AND-parallelism

AND-parallelism is implicitly present in a query such as:

? q(X) , r(Y).

since the query will result in the evaluation of the multiple goals q and r. The logic programming paradigm places no restriction on the order of evaluation of q and r, and in the ideal case they could be called concurrently. If the calls are working on unrelated solutions, one particular form of parallelism, **all-solutions and-parallelism** [Gregory85a] is an easy paradigm to implement. On a multi-processor machine, the two calls could be allocated to different processors and the computation would be able to proceed efficiently.

In many computations which involve AND-parallelism calls may not be completely independent but share logical variables. **Restricted AND-parallelism** where mutually dependent calls are evaluated sequentially could be used, but this would still not exploit the full potential of a parallel processor and is inefficient. In order to permit the concurrent evaluation of calls that share a variable, **stream AND-parallelism** has been introduced into PARLOG; this allows the expression of the communication which occurs in such cases.

A language employing a left-to-right, depth-first computational rule would not be able to exploit the potential And-parallelism present in a program. Prolog in the presently implemented form throws away all And-parallelism which it encounters.

2.4.2. Directional and non-directional logic programs.

Prolog is a non-directional logic programming language in that no distinction is made between which arguments of a predicate can be used for input and which can be used for output. This is often held to be one of the great advantages of the language in that one predicate can be used to perform apparently different tasks. However, in practice many Prolog programs do not make use of this facility. Indeed, the non-logical "cut" operator is often employed in order to prevent this kind of program behaviour by pruning away all unexplored branches from the search tree after the cut has been encountered. Programmers use the cut for several reasons, including increased speed of execution since unwanted solutions are not explored, and saving memory since backtracking points do not have to be recorded for later examination.

Programs in PARLOG are directional: the use of of mode declarations means that relations cannot be used "in reverse", unlike the case in Prolog. Combined with the committed choice search strategy, this makes the implementation of And-parallelism more practical because bindings do not have to be undone since

there is no backtracking. The advantages that have been gained in terms of stream communication are enormous, enabling PARLOG to be used constructively in applications where concurrency and process state are of importance.

2.4.3. Communicating processes in PARLOG.

Although in a logic program a relation call cannot actively change state, but only reduce to other calls, a relation which calls itself recursively with different arguments can be viewed as a long lived process which changes its state [Gregory85b]. Since PARLOG allows more than one call to be evaluated concurrently, the language permits computations which comprise the execution of several concurrent processes, and is able to express the concepts of concurrency which have been of importance in the areas of systems programming, and more recently in applications work. Prolog's search strategy, on the other hand, only allows one relation at a time to be evaluated, so that only one such process can be regarded as being in existence at any point.

2.4.4. Stream communication

PARLOG uses the data flow model of communication, information being passed between concurrently executing PARLOG processes using *stream communication.* PARLOG's ability to perform **stream-AND-parallelism** gives the language the power to express stream communication in a very simple manner. Shared variables act as communication channels and messages can be sent by the incremental construction of partly instantiated data structures such as lists of terms. The communication channels are effectively unbounded queues. Output variables are produced in an asynchronous manner, but input variables are processed synchronously if an argument contains a partially instantiated term.

In fact, the use of streams permits more complex forms of communication, for example back-communication (see below). One advantage of stream-AND-parallelism is that solutions are communicated incrementally, a disadvantage is that its use admits the computation of only one solution. Since stream-AND-parallelism is a more primitive form of parallelism in terms of which other schemes can be implemented, it can be used to implement a multi-solution mode using incremental construction of a solution list [Gregory85a].

2.4.4.1. Synchronisation and suspension

Process synchronisation is an important component in communication, and is achieved in PARLOG by suspension of **input matching** until a variable is bound. The notion of "modes" is provided in PARLOG so that communication between concurrently executing calls may be expressed. The "**?**" declaration when used to annotate an argument indicates that a non-variable term appearing in that position in the head of a clause may be used only for input matching. The call to a goal which contains an uninstantiated variable in an input mode position may suspend until that variable has become instantiated. For example, given a relation declaration:

```
mode is_list(?).
is_list( [H|T] ).
```

a call *? is_list([X|Y])* will **succeed**, a call *? is_list(foo)* will **fail**, and a call *? is_list(X)* will **suspend**. Note that the list notation **[H|T]** is equivalent to the data structure **'.'(H,T)**, where the top-level functor name is '.'. Suspension thus occurs in order to prevent the variable *X* in the call *? is_list(X)* from being bound to **'.'(H,T)**. Deadlock occurs if no progress can be made with any call, so that all calls are suspended. Mode declarations mean that in a situation involving communication only one PARLOG process can be the producer binding a particular shared variable, whilst there may be one or more consumers of the communication stream.

Output arguments are indicated by a "^" mode annotation, and output is performed by **unification** after the clause has been selected. Thus, given a PARLOG program:

```
mode foo(^,^).
foo(f(X),X).
```

- a call *? foo(A,B)* will result in the bindings **A=f(Z) , B=Z**.
- a call *? foo(g(X),Y)* will fail,
- a call *? foo(f(a),Y)* will succeed, with the binding **Y=a**.

2.4.4.2. Producers and consumers.

The basic paradigm of stream communication is that of producers and consumers. As an illustration, consider the call

? eager_producer(Stream) , lazy_consumer(Stream).

where the programs for *eager_producer* and *lazy_consumer* are:

Example 1

```
mode eager_producer( Stream^ ).
eager_producer( [ Item | Stream ]) <-
  eager_producer( Stream ).


mode lazy_consumer( Stream? ).
lazy_consumer( [ Item | Stream ] ) <-
  lazy_consumer( Stream ).
```

Note firstly that we do not define termination clauses for either relation, so that the call given above is non-terminating. Secondly, we do not concern ourselves here with the nature of **Item**, which is a variable in the above program sketch.

The *eager_producer* process may run ahead of the *lazy_consumer* by an arbitrary amount, and we assume the existence of system buffers to permit this. However, the *lazy_consumer* is constrained by the rate of production of **Item** slots, ie by the instantiation of the list functor. If the *eager_producer* for some reason instantiated **Stream** at a slower rate than the *lazy_consumer* could consume it, the progress of the latter process would be constrained by suspension on each recursion until **Stream** was instantiated at the top level to a list.

The PARLOG programmer is, however, able to utilise the completely synchronous communication facilities offered by the language. This is achieved by *back-communication* which can take two forms.

Firstly, the mode of the producer may be reversed, causing it to become "lazy". In this case, the consumer has the output mode on the shared variable, and the producer is declared input on that argument. The consumer than sends the producer a list of variables to be instantiated as messages; unable to run ahead of the consumer, the producer's eagerness is constrained by that process.

The second, analogous to "rendez-vous", is that of the co-operative construction of binding terms. The producer sends a stream (eg an incrementally constructed list) of tuples, each of which contains two arguments, one the data item to be sent and the other a variable. The example code below illustrates this; we use **msg(Item,Reply)** for the message tuple.

Example 2

```
mode synchronous_producer( Stream^ ).
synchronous_producer([ msg(item,Reply) | Stream ])<-
      data(Reply)&
      synchronous_producer(Stream).


mode synchronous_consumer( Stream? ).
synchronous_consumer([ msg(Item,Reply) | Stream ])<-
      Reply = ok ,
      synchronous_consumer(Stream).
```

As an example, a non-terminating invocation would be:

? synchronous_producer(Stream) , synchronous_consumer(Stream).

In the above, **synchronous_producer/1** may only produce one message tuple at a time, and is forced to wait until the **Reply** argument has been ground by the call *Reply = ok* in **synchronous_consumer/1**. The overall sequencing of the communication is achieved by the sequential-AND operator in **synchronous_producer/1** between the call to **data/1** and the self recursive call.

2.4.5. OR-parallelism

A relation defined by more than one clause possesses the potentiality of supporting a parallel search of all its member clauses in order to determine which one(s) result in success. This has been termed OR-parallelism, which refers to

"concurrency in the search for solutions to a single relation call" [Gregory87a].

OR-parallelism can be combined with any of the schemes for AND-parallelism, as well as with an AND-sequential search strategy, such as that employed by Prolog. In the following example, only OR-parallelism is illustrated:

```
q(X) <- r(X).
q(X) <- s(X).
```

In evaluating a call to q

? q(X)

both clauses of q are invoked concurrently. The way in which solutions are obtained depends on the implementation of the logic language that is being used. PARLOG will pick one of the clauses at random (in the absence of input-matching or guards) and attempt to explore that one: the result of success or failure will be reported, but no backtracking is employed, so that no other clause will subsequently be chosen for evaluation.

2.4.6. Committed choice non-determinism

PARLOG employs "don't care" non-determinism as a means of obtaining a solution, rather than the backtracking strategy used by Prolog to implement "don't know" non-determinism. The former strategy can be viewed as a form of intelligent backtracking which eliminates unnecessary searching [Kowalski79]. Don't-care non-determinism formed the basis of Dijkstra's language of guarded commands [Dijkstra76].

Production of one solution to a query is useful in many applications, especially that involving the description and programming of systems which in real life possess the property of **committed choice non-determinism**. As an example, although it may be possible to write a file in several ways, it usually desirable to implement only one solution. In system programming *achieving* a specified goal is normally of

more consequence than the *method* of achievement.

Although in some cases the PARLOG programmer really does not care which clause in a predicate is chosen for evaluation, it is more usual to specify the conditions under which a clause may be selected. The two methods open to the programmer are *guards* and *input matching*.

2.4.7. Guards

A clause containing a guard is evaluated first with respect to the guard; only if the guard succeeds will the evaluation be committed to the body, after which there is no going back. Where a relation is defined by several clauses, each clause will be evaluated with respect to its guard, the evaluation proceeding in parallel down the search tree. As soon as one guard is successfully evaluated, the body of that clause is committed to, and the evaluation of all the other clauses is terminated. During the search for a candidate clause, no variables in the call are bound, and no output binding is made to variables of the call, until the evaluation commits to the use of some clause. Since there is no backtracking on the choice of the candidate clause, bindings to variables in the call are never retracted.

An example of a program which makes use of guard and non-determinism is **partition/4**, which partitions a list of items (integers or constants) according to the value of a partition element into two lists. This program is used in the quicksort program. The PARLOG program for **partition/4** is:

Example 3

```
mode partition( partition_element? , input_list,greater_eq^ , less_eq^ ).
partition( Pe , [] , [] , [] ).
partition( Pe , [H|T] , [H|Ge] , Le )<-
        Pe =< H :
        partition( Pe , T , Ge , Le ).
partition( Pe , [H|T] , Ge , [H|Le] )<-
        H =< Pe :
        partition( Pe , T , Ge , Le ).
```

Note that if the list element being compared is equal in value to the partition element, the above program non-deterministically allocates it to either the **Greater_eq** list or the **Less_eq** list. This is intentional, and is due to the fact that the guards in the second and third clauses are not mutually exclusive.

2.4.8. Input matching

PARLOG employs input matching as a means of selecting a candidate clause for evaluation. This standard technique of logic programming can be used by the PARLOG programmer in order to write programs without guards, thus making the programs more succinct.

A standard PARLOG program which exploits input matching and suspension is **merge/3**, which merges two streams to produce one output stream. The program is:

Example 4

```
mode merge(A?,B?,Out^).
merge( [Item|A] , B , [Item|Out] )<- merge( A , B , Out ).
merge( A , [Item|B] , [Item|Out] )<- merge( A , B , Out ).
merge( [] , Out , Out ).
merge( Out , [] , Out ).
```

An example call is:

? eager_producer(L1), eager_producer(L2) , merge(L1,L2,L3).

Note that the merging of the two lists will be partly determined by the rate at which each is produced. The call to **merge/3** will result in a parallel clause search: if neither input stream is empty, the first two clauses are the candidates. Their selection will be suspended until L1 or L2 is bound to a partial list of the form [H|T] by one of the producer processes. If, for example, the second stream **B** is produced more slowly than the rate it can be consumed by **merge/3**, the call will commit to the first clause. Thus the call will result in a time dependent merge, a requirement of systems programming where a communal resource is shared and accessed by clients. Messages from different clients can be merged and passed through to the resource as they arrive.

If both L1 and L2 are partly instantiated, the call to **merge/3** will non-deterministically commit to one of the first two clauses The order of items within each stream is preserved in the output stream, but the program is not a fair merge, since it may repeatedly select the same clause in such a case. Fair and efficient merges can be written in PARLOG and similar languages [Clark85].

2.4.9. Kernel PARLOG

A PARLOG clause can be translated into a standard form, **Kernel PARLOG**, which has no mode declarations. All head arguments are distinct variables, input matching and output matching being done by explicit calls to PARLOG unification primitives in the guard and body of the clause respectively. The primitives[1] used to perform input matching is **<=** (one way unification), while **=** (full unification) is used for output.

As an example, the first clause of the **merge/3** relation (above) is presented in kernel form:

Example 5

```
merge(X,B,Z)<-
        [Item|A] <= X :
        Z = [A|Out] ,
        merge(A,B,Out).
```

A more detailed discussion of the implementation of input matching and output assignment, which is related to the kernel form of PARLOG, may be found in [Gregory87a].

2.4.10. Safe guards

In PARLOG, the evaluation of a guard must not be allowed to bind variables in the call, otherwise even if the call were to eventually fail, such bindings might remain. Output unification must be made in the *body* of a clause, after commitment to it. Even if variables in input positions have been instantiated by input matching, these bindings will not affect output arguments until commitment.

It is important that evaluation of a guard does not bind variables in an input position, a restriction known as *the safe guard condition*. Thus a guard may only bind variables appearing in the guard, body and output positions in the head of the clause. The mode declarations allow the implementation of a guard safety check at compile time, which may be over restrictive in that it declares unsafe some guards which are actually safe. The current implementation of PARLOG uses a simpler mechanism [Foster86] [Gregory87b], and the programmer is responsible guard safety.

Concurrent Prolog [Shapiro83a] does not impose any constraint on the form of guards, but requires an elaborate run-time mechanism to ensure the locality of bindings made by guards. Guarded Horn Clauses [Ueda86] uses a simpler mechanism which still requires an expensive run-time check, since the attempt to bind a call variable during either input unification *or* a guard call is the suspension mechanism.

[1] See the Appendix

3. The Semantics of PARLOG

The semantics of PARLOG has been given an interpretation in terms of a modified version of Milner's Calculus of Communicating Systems [Milner80] by [Hussey87]. This work follows on from that done by [Ellis86], and is an advance on the work presented in [Beckman86]. Hussey's approach was to model PARLOG processes as agents in a form of CCS modified to allow global scope of variables, unification and suspension.

4. Applications of PARLOG

4.1. Metalevel programming

PARLOG provides a metalevel programming facility, as do Prolog and LISP. A metacall similar to that of Prolog exists in PARLOG: in its simple form the **call/1** predicate has mode declaration:

mode call(goal?).

The logical reading of **call(goal)** is the same as that of **goal**. A call *call(Goal)* suspends until **Goal** is instantiated to a term denoting a PARLOG clause body (a relation call or conjunction). A program can thus evaluate calls which are determined at run-time. Considering the query:

? call(X) , X=write(foo).

the call *call(X)* will suspend until **X** is bound to the atom **write(foo)**; the entire query is equivalent to:

? write(foo)

However PARLOG currently has a more powerful metacall facility, **call/4**, derived from the three argument metacall described in [Gregory87a], having the mode declaration:

mode call(Database?,Goal?,Statusˆ,Control?).

The Control argument is an input stream of messages which can be used by a supervisor or monitor program to control the evaluation of the Goal. The acceptable messages on the control stream are: **stop**, **suspend** and **continue**.

The Status argument, a variable at the time of the call, is instantiated by the call to a stream of messages reporting key states in the evaluation of the call. The last message will be one of: **failed**, **succeeded** or **stopped**, indicating the form of termination; **stopped** indicating premature termination due to an input **control** message on the **Control** stream. Before termination, the messages **suspend**, **continue** may be output on the **Status** stream when such messages are received on the **Control** stream. Finally, **exception** messages may also be output on the **Status** stream, signaling exceptions such as deadlock, overflow or a call to a relation undefined in **Database**. A monitor program can then handle the exceptions; the exception message for the undefined relation is of the form:

exception(undefined,Undefined_call,Var)

where **Var** is an unbound variable for back communication. The metacall suspends on **Var** waiting for it to be bound, and then continues with the term to which **Var** was bound in place of the undefined call. By this means the monitor program can substitute some other call for the undefined call, and by binding Var to the goal **evaluate(Other_database,Undefined_call)** it can cause the call to the relation undefined in **Data_base** to be evaluated in **Other_database**.

The PPS, a PARLOG programming system written in PARLOG [Foster87] depends heavily on the use of the metacall. Details of the use of the metacall by the PPS can be found in [Clark86a]. An example program is given below, which implements a Unix style shell that accepts a stream of user commands to call programs in a particular database. The shell program has been modified from that given in [Clark86b]. The user programs are to be executed as either background commands, a command term of the form **bg(Db,Call)**, or foreground commands, **fg(Db,Call)**. When a foreground command arrives, the shell suspends all current background processes until the foreground command terminates, priority being given to foreground processes. The second argument of the **pri_shell** program is a control message stream

consumed by each of the metacalls running a background process.

Example 6

```
mode pri_shell(User_commands? , Background_control^).
pri_shell( [] , Control ).
pri_shell( [ bg(Db,Call ) | Commands ]  ,Control )<-
        call( Db , Call , Status , Control ) ,
        pri_shell( Commands , Control ).
pri_shell( [ fg(Db,Call) | Commands] , [ suspend | Control ] )<-
        call( Db , Call , Status , New_control ) &
        ( Control = [continue|New_control] ,
        pri_shell( Commands , New_control )  ).
```

The use of the sequential conjunction in the last clause ensures that the sending of the **continue** message to the suspended background calls is delayed until the completion of the foreground call.

4.2. Advanced Man Machine Interfaces and PARLOG

PARLOG's ability to express concurrency, stream determinism and non-determinism greatly simplify the development of programs on a modern high performance workstation. Such work requires the use of highly sophisticated man-machine interfaces. Recent work [Barnes87] has assessed in detail the suitability of PARLOG for the construction of such interfaces, and has further enhanced features of the PPS [Foster87] (see above). This includes a graphical interface to a browser, a tool which creates its own window and then allows the user to access all the programs that have been created. Many functions have been added, including the ability to edit relations, purge and restore versions and return to the latest version of a program. This facility makes the task of program development and maintenance easier, and the graphical interface further simplifies the interaction between the programmer and the PPS due to the model of the PPS that is presented to the user.

4.3. Concurrent systems programming in PARLOG

PARLOG can be used to describe communicating systems, in a form which is both a specification and a program. In the following example, a synchronous buffer, use is made of the **synchronous_producer** (above). We describe a one-place buffer which accepts message tuples of the form **msg(Item,Reply)** on an input channel, and grounds **Reply** to the constant 'ok'. This buffer is recursive, but will not accept the next message until the process waiting on the output channel has acknowledged receipt of the Item. Note that the buffer terminates when the **In** channel becomes the empty list.

Example 7

```
mode buffer(in? , out^).
buffer( [] , [] ).
buffer( [ msg(Item , Reply) | In ] , [ msg(Item , Reply1) | Out]) <-
        Reply = ok ,
        data(Reply1)&
        buffer(In , Out).
```

A buffer of any size may be built by recursively spawning one-place buffers, and connecting the output channel of one to the input channel of the next:

Example 8

```
mode buffer_n(size? , in? , out^).
buffer_n(0 , Out , Out);
buffer_n(Size , In , Out)<-
        Size1 is Size - 1 ,
        buffer(In , Mid) ,
        buffer_n(Size1 , Mid , Out).
```

The communication of a synchronous producer-consumer pair via a ten-place buffer would be invoked by:

? synchronous_producer(A), buffer_n(10,A,B), synchronous_consumer(B).

We can also connect several producers to one consumer, noting that the producer-consumer paradigm may be either synchronous or asynchronous:

? producer1(A) , procucer2(B) , producer3(C) , merge_all([A,B,C],All) , consumer(All).

Note that the producers-consumer paradigm above may be either synchronous or asynchronous; we can imagine that the producers are users who wish to access a shared resource such as a printer, represented by the **consumer**. In this case, the producers may send a variable which is a reference to some text file to be printed. A more sophisticated version of the algorithm would make all the producers and the consumer synchronous, with an n_place buffer between the **merge_all** process and the printer (**consumer**): the **Item** in the message tuple **msg(Item,Reply)** would represent the text file to be printed:

? synchronous_producer(A), synchronous_producer(B), synchronous_producer(C), merge_all([A,B,C],D), buffer_n(20,D,E), printer(E).

The **merge_all/2** relation accepts as input a list of streams to produce one stream using **merge/3** and may be defined thus:

Example 9

```
mode merge_all(streams? , all_stream^).
merge_all([] , []).
merge_all( [ Stream | Streams ] , All_Stream)<-
      merge_all(Streams , Streams1),
      merge(Stream, Streams1 , All_stream).
```

PARLOG can also be used to translate formal description techniques, for example those written in LOTOS, into executable code [Gilbert87a] [Gilbert86]. An important feature of a translation of this kind is that LOTOS is used to describe concurrent systems, and the resultant PARLOG code is capable of concurrent execution where appropriate.

4.4. Temporal Logic and PARLOG

PARLOG programs can be given a natural interpretation in terms of networks of communicating processes. Systems programming requires the use of synchronisation, and classical logic languages do not express tractably the constraints needed to control program execution. One form of logic that can be used is the temporal approach [Gabbay85] [Gabbay87] and work has recently been done to implement such a system in PARLOG [Shah87]. Temporal constraints can be specified at a high level using a subset of temporal logic, and automatic source-to-source translation is used to convert the specification into a PARLOG program which executes in the required manner.

4.5. Using PARLOG for simulations

The way in which PARLOG can be used for discrete event simulation has been investigated where systems are modelled by a collection of communicating processes running in parallel [Broda84]. PARLOG is a natural specification language for communication protocols due to its ability to express concurrency and communication, and initial work to this end is reported in [Gregory85c] and the specification of a telephone switching system has been made by extending PARLOG to incorporate the concept of a real time clock[Elshiewy86].

4.5.1. Digital circuit simulation and debugging

The use of PARLOG as a computer hardware description language is a relatively new topic. The use of and-parallelism, ability to express processes and stream communication allows hardware circuits to be described and simulated both naturally and efficiently [Chan87]. PARLOG can specify sequential machines directly from the physical configuration due to the parallel nature of the language, whilst the use of Prolog requires a program transformation before the circuit component can be specified. In order to perform circuit debugging, backtracking is simulated in PARLOG by the use of stacks and deep guards. The analogy between digital circuits and the and-or tree representation of logic program reduction indicates an application of circuit debugging algorithms to program debugging.

4.5.2. Specification and simulation of parallel system architectures in PARLOG

A declarative Computer Hardware Description Language (CHDL) has been developed in PARLOG to specify parallel system architectures and the CHDL produced used to specify aspects of the FAIM-1 (Fairchild AI Machine) system architecture [Steer87].

Simulation tools designed to be used on the SUN-3 workstation were produced and were designed; the speed at which algorithms were coded and tested allows the user to generate and test their own specifications efficiently. The work also showed that PARLOG can be used directly to write functional specifications.

4.6. Object Oriented Programming in PARLOG

Logic programming and object-oriented programming are formalisms which are suitable for combination. Logic programming gives a logical meaning to objects, allowing them to be verified and transformed easily, and provides a clean way of representing the relations between objects. Also logic programming languages are more powerful than object oriented programming languages due to their use of unification. Concurrent logic programming has the advantage of being able to express concurrent objects. The method of representing objects in concurrent logic programming languages was first explored in [Shapiro83b], and has been further developed in [Kahn86] and [Davison87] [Davison88]. The latter author has developed an object-oriented language called POLKA, which is implemented on top of PARLOG. It contains features found in languages such as Smalltalk, Simula and Hewitt's Actor languages. In addition, it allows meta level programming using objects. This is similar to ideas found in Bowen's MetaProlog [Bowen85]. The language supports encapsulation of data, synchronous and asynchronous message passing, inheritance (single, multiple and dynamic), self communication and dynamic creation of objects . The language allows the use of all the features of PARLOG, including AND and OR parallelism, the logical variable and committed choice non-determinism. This results in new programming techniques such as message peeking, object splitting and partially defined objects. PARLOG programs for representing problems involving networks of communicating processes are especially easy to express in POLKA. Such code is shorter than the PARLOG equivalent since process recursion, process arguments and message streams are hidden. It is also possible to execute PARLOG procedures directly from within POLKA.

Applications of POLKA include :

(i) a window manager on the SUN 3

(ii) simulation of simple computer hardware

(iii) bank account objects

Future applications will include the coding of a large subset of the speech recognition system HEARSAY, to investigate POLKA's use for blackboards, and also the programming of petri nets.

What are objects in PARLOG?

- An object can be implemented as a tail recursive PARLOG process, with its internal state held as unshared arguments.
- PARLOG objects communicate by partially instantiating shared variables which act as message streams and determine the network. Thus the sending of the message **M** on the stream **X** is represented by the binding **X = [M|X1]**.
- An object process is suspended until its message streams are partially instantiated with a message, following PARLOG's normal suspension rules.
- Inheritance in objects can be performed by connecting a message stream from an object to its inheritor. Unrecognised messages are passed from an object lower down in the inheritance hierarchy to its superior.
- Responses to messages may be passed using back communication rather than via an explicit return stream.

The basic ideas underlying the representation of objects in PARLOG is given by the following example. An object represents a bank account which has one state for the amount of money that it contains. The kinds of messages that it can receive are:

(i) balance(A) returns the balance in the account

(ii) credit(Val) credits 'Val' to account

(iii) debit(Val) debits 'Val' from account

The PARLOG code for this, and the POLKA version is given below:

Example 10

```
mode account(Message_stream?).
account(Message_stream)<-
        account(Message_stream , 100).


mode account(Message_stream? , Amount?).

account([balance(A)|Stream] , Amount)<-
        A = Amount ,
        account(Stream,Amount).

account([credit(Val)|Stream] , Amount)<-
        New_amount is Amount + Val ,
        account(Stream , New_amount).

account([debit(Val)|Stream] , Amount)<-
        New_amount is Amount - Val ,
        account(Stream , New_amount).

account([] , Amount);

account([Message|Stream] , Amount)<-
        write('account does not understand ')&
        write(Message)&
        account(Stream,Amount).
```

Example 11

```
name account
variables invisible  Amt state  <= 100

clauses
    balance(A)   -->   A = Amt.

    credit(Val)  -->   Amt becomes Amt + Val.

    debit(Val)   -->   Amt becomes Amt - Val.

end.
```

To execute the object it must be called with its argument bound to a stream of messages which are incrementally instantiated:

? account(Stream), Stream = [balance(A),.....]

Note that **name**, **variables, invisible**, **clauses, state, become** and **end** are reserved words in POLKA. All the clauses have a format of the form:

messages --> actions

and the message can be any PARLOG term, including a variable. The action is a sequence of operations which may be calls to PARLOG relations or a POLKA cliche. There is no need to include recursive calls to the 'account' object. Note that the use of **becomes**, which looks like destructive assignment, is restricted to changing a variable only once in a clause. The general form of **becomes** is

Var1, Var2, ... VarN **becomes** Expr1, Expr2, ... ExprN

A POLKA program is compiled into PARLOG, and the **becomes** cliche is converted into an argument replacement operation, thus giving it a simple declarative meaning.

It is not necessary to define the input stream to an object, and termination and error clauses are added by default to the PARLOG code by the POLKA-PARLOG compiler.

Other recent work has attempted to use PARLOG in an object-oriented style to encode problems [Buckley87]. This approach has to represent the problem directly in PARLOG without first writing the code in an OOP language; the domain investigated was the taxi scheduling algorithm.

5. Implementations of PARLOG

A sequential implementation of PARLOG, the Sequential PARLOG Machine (SPM), using an emulator written in C, is the major system currently used by the PARLOG research group [Foster86] [PARLOG Group87a] and is based on an abstract PARLOG machine design [Gregory87b]. The PPS, a PARLOG programming environment, has been developed to run on the SUN-3, allowing the user to interact with and control multiple evaluations using windows and mouse [Foster87]. In this system a program is a collection of databases (modules) with local name spaces, programs in one database being able to call relations defined in another database. Built into the system is an integrated editor which automatically retrieves old versions of programs. Various program development tools are being constructed, for example debuggers [Huntbach87], type checkers [Gilbert87b], and static analysis tools [PARLOG Group87b]. Licences for the SPM implementation on Hewlett Packard 9000, Sun or Vax are available from the PARLOG Research Group.

A new implementation of PARLOG is being developed which will execute on multiprocessors faster than the SPM, and should be available in 1988. It is being developed on a Sequent Balance 8000 six-processor and the emulator has been designed in such a way that it ports to other multi-processors with a similar (Unix based) parallel programming environment. The initial results on the Sequent show good parallel speed-ups running on up to six processors (the maximum available) [Crammond88].

An implementation scheme [Gregory87c], designed to map PARLOG onto DACTL [Glauart87] is under way, with a view to executing PARLOG on the Alvey Flagship machine [Watson87].

Acknowledgements

This tutorial paper has been compiled whilst the author was employed on Alvey grant "Implementation and Applications of PARLOG", Project number 043/098.

I would like to thank all the members of the PARLOG group, and students past and present, whose work has contributed to this paper.

6. Appendix

6.1. The syntax of a PARLOG program.

The PARLOG syntax used in this article is given in the Table below.

Table 1 PARLOG syntax

Symbol	Meaning
<-	logical implication
&	sequential-AND
	parallel-AND
;	sequential-OR
.	parallel-OR
:	guard operator
?	input mode annotation
^	output mode annotation

6.2. Unification primitives in PARLOG :

=/2 : full unification

<=/2 : one way unification

==/2 : test unification, which does not bind any variables.

6.3. Modes and suspension

Mode declarations are used to specify communication constraints on shared variables, which are declared to be either **?** ("input") or **^** ("output"), thus acting as communication channels. These declarations are made once for each relation, and each argument of the relation is annotated:

mode name(a_1?,..,a_k^,..)

Messages sent along these channels are incrementally constructed from partially determined data structures, usually consisting of lists of terms acting as message streams. Suspension is the method used to synchronise concurrently evaluating calls, and thus a PARLOG call may succeed, fail or be in suspended state waiting for information on a communication channel to be further instantiated.

6.4. Negation

Negation is implemented in PARLOG as negation by failure [Clark78].

References

Barnes87.
R. G. T. Barnes, ‘‘Advanced Man Machine Interfaces in PARLOG,’’ Report for MSc degree in Engineering, Imperial College of Science and Technology, London, September 1987.

Beckman86.
L. Beckman, ‘‘Towards a formal semantics for concurrent logic programming languages.,’’ *Third International Conference on Logic Programming, 1986*, pp. 335-349, Springer-Verlag, London, UK, 1986.

Bowen85.
K.A. Bowen, ‘‘Meta-level Programming and knowledge representation,’’ *New Generation Computing*, vol. 3, pp. 359-583, 1985.

Broda84.
Krysia Broda and Steve Gregory, ‘‘PARLOG for Discrete Event Simulation,’’ *2nd International Logic Programming Conference, Uppsala, July 1986*, Dept of Computing, Imperial College, London. UK, March 1984.

Buckley87.
Michael Buckley, ‘‘Process Modelling in PARLOG,’’ MSc. Thesis, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, September 1987.

Chan87.
C.K. Chan, ‘‘PARLOG for Digital Circuit Simulation and Debugging,’’ MSc Thesis, Department of Computing, Imperial College, London, UK, September 1987.

Clark78.
Keith Clark, ‘‘Negation as failure,’’ in *Logic and Databases*, ed. Gallaire, H. and Minker, J., pp. 293-322, Plenum Press, New York, 1978.

Clark81.
Keith Clark and Steve Gregory, ‘‘A relational language for parallel programming,’’ *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architectures*, pp. 171 - 178, Portsmouth, NH, October 1981.

Clark85.
Keith Clark and Steve Gregory, ‘‘PARLOG: Parallel Programming in Logic,’’ DOC 84/4, Dept of Computing, Imperial College, London. UK, June 1985.

Clark86a.
Keith Clark and Ian T. Foster, ‘‘A Declarative Environment for Concurrent Logic Programming,’’ *Proc. TAPSOFT 87, Pisa, Italy.*, Dept of Computing, Imperial College, London. UK, 1986.

Clark86b.
Keith Clark and Steve Gregory, ‘‘PARLOG: Parallel programming in Logic,’’ *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 1, Jan 1986.

Crammond88.
Jim Crammond, ‘‘Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors,’’ PhD Thesis (in preparation), Department of Computer Science, Edinburgh, UK, 1988.

Davison87.
Andrew Davison, ‘‘POOL : A PARLOG Object Oriented Language,’’ PAR 87/5, PARLOG Group, Department of Computing, Imperial College, London, UK, April 1987.

Davison88.
Andrew Davison, ‘‘POLKA : A PARLOG Object Oriented Language,’’ (in preparation), PARLOG Group, Department of Computing, Imperial College, London, UK, 1988.

Dijkstra76.
E.W. Dijkstra, *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, NJ, USA, 1976.

Ellis86.
Mark R. Ellis, ‘‘A Relational Language into ECCS,’’ M.Sc, Imperial College of Science & Technology, London. UK, 12th September 1986.

Elshiewy86.
N. A. Elshiewy, *Extended PARLOG: Logic Programming of Real Time Systems,* Computer Science Laboratory, Ericsson Telecom, Sweden, 1986.

Foster86.
Ian Foster, Steve Gregory, Graem Ringwood, and Ken Satoh, ‘‘A Sequential Implementation of PARLOG,’’ *3rd International Conference on Logic Programming, London. July 1986*, Dept of Computing, Imperial College, London. UK, March 1986.

Foster87.
Ian Foster, ‘‘The PARLOG Programming System (PPS),’’ manual, Imperial College, London, May 1987.

Gabbay85.
Dov. M. Gabbay, *Temporal Logic and Computer Science,* DoC, Imperial College, London, UK, May 1985.

Gabbay87.
D. M. Gabbay, *Executable Temporal Logic for Interactive Systems,* DoC, Imperial College, London, UK, March 1987.

Gilbert86.
David Gilbert, ''Implementing LOTOS in PARLOG,'' MSc Thesis, Department of Computing, Imperial College, London, UK, September 1986.

Gilbert87a.
David Gilbert, ''Executable LOTOS: Using PARLOG to implement an FDT,'' *Proceedings of IFIP Protocol Specification, Testing and Verification: VII, Zurich, Switzerland, 5-8 May 1987*, Elsevier Science, North-Holland, Amsterdam, Netherlands, 1987.

Gilbert87b.
David Gilbert, *PTCP Polymorphic Type Checker for PARLOG: USER GUIDE,* The PARLOG Group, Dept. of Computing, Imperial College, London., London, March, 1987.

Glauart87.
J.R.W. Glauart, J.R. Kennaway, and M.R. Sleep, ''DACTL: a computational model and compiler target language based on graph reduction,'' *ICL Technical Journal*, vol. 5, no. 3, pp. 509-537, ICL, Manchester, UK, May 1987.

Gregory85a.
Steve Gregory, ''Design, Application and Implementation of a Parallel Pogic Programming Language,'' Phd thesis, London, UK, 1985.

Gregory85b.
Steve Gregory, Rob Neely, and Graem A. Ringwood, ''PARLOG for specification, verification and simulation,'' IC Research report DOC 85/7, London, UK, 1985.

Gregory85c.
Steve Gregory, Rob Neely, and Graem Ringwood, ''PARLOG for specification, verification and simulation,'' *7th International Symposium on Computer Hardware Description Languages and their Applications, Tokyo, August 1985.*, Dept of Computing, Imperial College, London. UK, May 1985.

Gregory87a.
Steve Gregory, *Parallel Logic Programming in PARLOG: The Language and its Implementation,* Addison-Wesely, London, UK, 1987.

Gregory87b.
Steve Gregory, Ian T. Foster, Alastair D. Burt, and Graem A. Ringwood, ''An Abstract Machine for the Implementation of PARLOG on Uniprocessors,'' Draft, PARLOG group, Dept of Computing, Imperial College, London. UK, January 1987.

Gregory87c.
Steve Gregory and Melissa Lam, ''An Implementation of PARLOG on DACTL,'' Draft Report, PARLOG Group, Department of Computing, Imperial College, London, UK, October 1987.

Hogger81.
Hogger, C.J., ''Derivation of Logic Programs,'' *Journal of the Association for Computing Machinery*, vol. 28, no. 2, pp. 372-392, 1981.

Hogger84.
C. J. Hogger, *Introduction to Logic Programming,* Academic Press, 1984.

Huntbach87.
Matthew M. Huntbach, ''Algorithmic PARLOG Debugging,'' *Proceedings 1987 Symposium on Logic Programming*, pp. 288-297, IEEE, Washington, DC, USA, September 1987.

Hussey87.
Charlie Hussey, ''Interpreting PARLOG Programs as CCS agents,'' Report for MSc degree in Engineering, Imperial College of Science and Technology, London, September 1987.

Kahn86.
Kenneth Kahn, Eric Dean Tribble, Mark S. Miller, and Daniel G. Bobrow, ''Objects in Concurrent Logic Programming Languages,'' *OOPSLA'86 Proceedings*, Knowledge Systems Area, Intelligent System Laboratory, Xerox Palo Alto Research Center, Palo Alto, USA, 1986.

Kowalski74.
R.A. Kowalski, ‘‘Predicate Logic as a programming language,’’ *Proceedings of the IFIP Congress 1974*, pp. 569-574, 1974.

Kowalski79.
R.A. Kowalski, ‘‘Algorithm = Logic + Control,’’ *Communications of the ACM*, vol. 22, no. 7, pp. 424-436, 1979.

Kowalski82.
R.A. Kowalski, ‘‘Logic as a computer language,’’ DOC 82/24, Department of Computing, Imperial College, London, UK, 1982.

Milner80.
Robin Milner, *A Calculus of Communicating Systems,* Lecture Notes in Computer Science, 92, Springer-Verlag, Berlin, 1980.

PARLOG Group87a.
The PARLOG Group, *PARLOG Predicates Reference Manual,* PARLOG Group, Dept of Computing, Imperial College, London. SW7, 1987.

PARLOG Group87b.
The PARLOG Group, *PARLOG Tools and Utilities,* The PARLOG Group, Department of Computing, Imperial College, London, 1987.

Roussel75.
Roussel, P., *PROLOG: Manuel de reference et d’utilisation,* Groupe d’Intelligence Artificielle, U.E.R. de Luminy, Universite Aix, Marseille, France, 1975.

Shah87.
Neeshmal N. Shah, ‘‘Using Temporal Logic Specifications to Control the Behaviour of PARLOG Programs,’’ Report for MSC degree in Engineering, Imperial College of Science and Technology, London, September 1987.

Shapiro83a.
Ehud Shapiro, ‘‘A Subset of Concurrent PROLOG and Its Interpreter,’’ TR-003, ICOT, Tokyo, 1983.

Shapiro83b.
Ehud Shapiro and Akikazu Takeuchi, ‘‘Object Oriented Programming in Concurrent Prolog,’’ CS83-08, Department of Applied Mathematics, Weizmann Institute of Sciences, Rehovot, Israel, June 1983.

Steer87.
K. G. Steer, ‘‘Specification and Simulation of Parallel System Architectures in PARLOG,’’ Report for MSc degree in Engineering, Imperial College of Science and Technology, London, September 1987.

Ueda86.
Kazunori Ueda, ‘‘Guarded Horn Clauses,’’ Thesis for Doctor of Engineering, University of Tokyo, Tokyo, Japan, 1986.

Warren87.
David H.D. Warren, ‘‘The SRI Model for Or-Parallel Execution of Prolog - Abstract Design and Implementation Issues,’’ *Proceedings 1987 Symposium on Logic Programming*, pp. 92-102, IEEE, San Francisco, USA, August 31 - September 4, 1987.

Watson87.
I. Watson, J. Sargeant, P. Watson, and V. Woods, ‘‘Flagship computational models and machine architectures,’’ *ICL Technical Journal*, vol. 5, no. 3, pp. 555-574, ICL, Manchester, UK, May 1987.