Parallel Logic Programming

K. L. CLARK
Department of Computing, Imperial College, London SW7 2BZ

Most research on parallel logic programming divides into (1) or-parallel implementations of Prolog, and (2) and-parallel implementations of committed choice variants of Prolog, the so-called concurrent logic languages. The reason is implementation efficiency: it is extremely complex to implement a combined and/or parallel system. This paper introduces the research on the concurrent and-parallel languages and their extensions, with an emphasis on Parlog. However, all the main concurrent languages are introduced and compared, and set in a historical context of precursor research. Recent work on extensions of the languages is described, particularly the Parlog extensions: Parlog++ and Polka for object-oriented programming, and Pandora for constrained search.

Received August 1990

1. INTRODUCTION

Research on parallel logic programming divides into two main branches: (1) work on parallelising Prolog, which is primarily research into efficient implementation of or-parallel execution, for example Aurora;\(^3\) (2) work on the design and implementation of concurrent, and-parallel, variants of Prolog, for example Parlog.\(^7\) In this brief overview we shall concentrate only on (2), for programming in a concurrent logic language is very different from programming in Prolog, even one that has a parallel implementation.

Primarily this is because the concurrent languages incorporate the concept of committed choice (or ‘don’t care’) non-determinism, as in CSP.\(^{19}\) Each program clause is divided into a guard component and a body component. Given a call to a procedure, the guard components of each clause of the procedure are evaluated in parallel and the evaluation commits to any clause whose guard component successfully terminates. The clause is called a candidate clause for the call. There is no backtracking on the choice of clause, even if there are other candidate clauses. During a guard evaluation, no bindings for variables in the call are generated. In consequence, only a single solution to any call is computed, by the body component of the selected candidate clause. One must therefore program in such a way that any candidate clause for a call will generate a solution for the call, if there are solutions.

In contrast, Prolog has ‘don’t know’ non-determinism. Successful unification of a call with a clause head, the guard computation for a Prolog clause, does not guarantee that the clause will generate a solution. Other unifying clauses must also be tried. Prolog tries each clause in turn using backtracking, while an or-parallel Prolog tries each unifying clause in parallel. \textit{Cut} is used in Prolog to commit the evaluation to a particular clause. Some or-parallel Prologs have introduced a symmetric cut, which commits the evaluation to a particular clause even if earlier clauses, being tried in parallel, have not yet generated all their solutions to the call. This is similar to the commitment to a candidate clause in a concurrent language, but with one major difference. In the concurrent languages no bindings for variables of the call are generated before commitment. In an or-parallel Prolog complete bindings may have been generated by the use of other clauses before execution of the symmetric cut. So even in programs in which each clause has a symmetric cut, multiple bindings for variables need to be maintained, the major complexity of or-parallel Prolog implementation.\(^{34}\)

In a concurrent language only one binding for each variable can be generated, and this binding is never rescinded. This is the key to efficient and-parallel implementation, in which calls with shared variables are evaluated in parallel. A variable becomes a shared-memory location which is assigned a single value by one of the calls. All the concurrent languages enable the programmer to specify whether or not a call can generate the binding or whether it can only consume a binding generated by some other call. The consuming is done by the guard evaluation of some clause for the consuming call. If the binding has not yet been generated, the guard evaluation for the clause suspends until the binding is generated. If a call has a suspended clause, and there is as yet no candidate clause, the call suspends. Suspension of a call waiting for one or more variable bindings to be generated, by one or more other calls, is the primary control component of the concurrent languages. They are \textit{data flow} languages.

Allowing multiple bindings and and-parallel evaluation of calls with shared variables requires an extremely complex implementation. Hence the major split of parallel logic programming research mentioned above. The or-parallel research focuses on allowing multiple bindings but no parallel evaluation of calls with shared variables. The concurrent logic programming research concentrates on allowing and-parallel evaluation of calls with shared variables; using committed choice and binding restrictions on guard evaluations to ensure that only one binding for each shared variable is ever generated. Some or-parallel Prolog implementations allow parallel evaluation of calls with no shared variables (\textit{independent and-parallelism}) and even and-parallel evaluation of calls for which there is only one unifying clause (\textit{deterministic and-parallelism}). The Andorra\(^{38}\) system has deterministic and-parallelism, a control concept introduced in Ref. 35. But each of these extensions is such that the parallel calls either do not communicate, or only communicate a single binding for a variable. (The fact that there is only one unifying clause for a call must be determined without generating any bindings for variables in the call. If this is not possible, the call is deemed to be non-deterministic.)

We shall use Parlog as the main exemplar of the
PARALLEL LOGIC PROGRAMMING

of one call to that of another when some shared variable of the two calls is bound or is about to be bound.

The simplest implementation of this idea was the freeze predicate of Prolog-II.° freeze(X,Call) delays the evaluation of Call if X is unbound, until X is bound to a non-variable by some later call.

IC-Prolog° had higher-level constructs. For example, if a variable X was annotated with ? in a call C, the call became an eager consumer of X. When X is bound to some partial term, say [1|Y], by the evaluation of some call P preceding C, C is evaluated as the next call even if it does not immediately follow P. But C will automatically suspend if it tries to bind Y, for it is a consumer of the whole of X. (freeze(X,C) does not resuspend C when this happens.) Control now transfers back to the suspended evaluation of P for it to generate a non-variable binding for Y. If and when it does, say binding Y to [2|W], control jumps back and the evaluation of C is resumed. This alteration continues, with the X acting as a communication channel between P and C for a stream of values [1,2, . . .] generated by P and consumed by C. Control transfers from P to C when a value is placed on the stream, and back from C to P when the next value is needed by C.

2.1 From co-routines to multiple processes

In the concurrent logic languages, P and C evaluate concurrently and C suspends if it runs ahead of the stream of values being communicated by P. This multiple-process generalisation of data flow co-routining for logic programming was first examined by van Emden and de Lucena. In that model (the vEdL model) shared variables are one-way communication channels which are incrementally bound to lists of ground terms—the streams of values sent along the channel. Calls can be evaluated concurrently and consuming calls suspend if the next value is not available on one or more of their input channels. The model does not deal with non-determinism. Each process, when its input data is available, must have only one unifying clause for its next reduction step. In general a process is reduced to a parallel conjunction of sequential conjuncts, each sequential conjunction becoming a new sequential subprocess.

In addition to its co-routining facilities, IC-Prolog allows the programmer to specify that a conjunction of calls should be evaluated in pseudo-parallel, with strict alternation between the reduction steps of each call. In addition, all but one of the calls can be made the consumer of some shared variable X using the ? annotation. This can be used to emulate the vEdL model.

A pseudo-parallel call can also be made to suspend until some variable is bound, with no further suspension if the binding contains variables. The variable is annotated with ! instead of ?. c(X!) is just freeze(X,c(X)) is a parallel context. This control feature can be used to emulate bidirectional communication via a single shared variable, which does not allow.

IC-Prolog also has control facilities for selecting alternative clauses on the basis of which arguments of the call are unbound variables and which arguments are non-variables. Used with the pseudo-parallelism, they allow one to emulate non-stream communicating

2. THE PRECURSORS OF THE CONCURRENT LANGUAGES

The late 1970s saw attempts to incorporate co-routining ideas into logic programming implementation based upon the idea of transferring control from the evaluation of one call to that of another when some shared variable of the two calls is bound or is about to be bound.

of one call to that of another when some shared variable of the two calls is bound or is about to be bound.

The simplest implementation of this idea was the freeze predicate of Prolog-II.° freeze(X,Call) delays the evaluation of Call if X is unbound, until X is bound to a non-variable by some later call.

IC-Prolog° had higher-level constructs. For example, if a variable X was annotated with ? in a call C, the call became an eager consumer of X. When X is bound to some partial term, say [1|Y], by the evaluation of some call P preceding C, C is evaluated as the next call even if it does not immediately follow P. But C will automatically suspend if it tries to bind Y, for it is a consumer of the whole of X. (freeze(X,C) does not resuspend C when this happens.) Control now transfers back to the suspended evaluation of P for it to generate a non-variable binding for Y. If and when it does, say binding Y to [2|W], control jumps back and the evaluation of C is resumed. This alteration continues, with the X acting as a communication channel between P and C for a stream of values [1,2, . . .] generated by P and consumed by C. Control transfers from P to C when a value is placed on the stream, and back from C to P when the next value is needed by C.

2.1 From co-routines to multiple processes

In the concurrent logic languages, P and C evaluate concurrently and C suspends if it runs ahead of the stream of values being communicated by P. This multiple-process generalisation of data flow co-routining for logic programming was first examined by van Emden and de Lucena. In that model (the vEdL model) shared variables are one-way communication channels which are incrementally bound to lists of ground terms—the streams of values sent along the channel. Calls can be evaluated concurrently and consuming calls suspend if the next value is not available on one or more of their input channels. The model does not deal with non-determinism. Each process, when its input data is available, must have only one unifying clause for its next reduction step. In general a process is reduced to a parallel conjunction of sequential conjuncts, each sequential conjunction becoming a new sequential subprocess.

In addition to its co-routining facilities, IC-Prolog allows the programmer to specify that a conjunction of calls should be evaluated in pseudo-parallel, with strict alternation between the reduction steps of each call. In addition, all but one of the calls can be made the consumer of some shared variable X using the ? annotation. This can be used to emulate the vEdL model.

A pseudo-parallel call can also be made to suspend until some variable is bound, with no further suspension if the binding contains variables. The variable is annotated with ! instead of ?. c(X!) is just freeze(X,c(X)) is a parallel context. This control feature can be used to emulate bidirectional communication via a single shared variable, which does not allow.

IC-Prolog also has control facilities for selecting alternative clauses on the basis of which arguments of the call are unbound variables and which arguments are non-variables. Used with the pseudo-parallelism, they allow one to emulate non-stream communicating

THE COMPUTER JOURNAL, VOL. 33, NO. 6, 1990 483

Copyright © 1990 Oxford University Press, Inc. All rights reserved.
processes. For example, Ref. 20 shows how one can use them to implement an or-parallel search algorithm using communicating and-parallel calls, all of which terminate when one of the calls succeeds. This was perhaps the first study of this concurrent logic programming technique, which uses shared variables to send control signals between processes.

A third control concept that can be emulated in IC-Prolog is co-routining between a lazy non-deterministic producer and a parallel consumer, the form of control needed for many constrained search applications where many checks need to be made for each alternative increment of a partial solution to some problem. Program 1, taken from Ref. 6, is the top level of the IC-Prolog program for the 8-queens problem. // specifies pseudo-parallel evaluation of the two calls of the recursive safe clause, & is sequential conjunction in IC-Prolog.

**Program 1**

eight_queens(L) :-
    perm([1,2,3,4,5,6,7,8],L) & safe(L?).
    safe([]).
    safe([Q|R]) :-
        no_take(Q,R) // safe(R).
    perm([],[]).
    perm([U|L],[V|PermL]) :-
        delete(V,[U|L],List) &
        perm(List,PermL).
    delete(U,[U|L],L).
    delete(V,[U|L],[U|DelL]) :-
        delete(V,L,DelL).

(+ a suitable recursive definition of no_take)

safe is an eager consumer of the permutation of L which is lazily and incrementally generated by the perm call. The ith integer on L gives the column position of the queen in the ith row. The pseudo-parallelism of safe means that there is a strict alternation between the evaluation of the no_take calls (one for each queen position already generated and placed on L). A no_take call suspends when it reaches the end of the stream of queen positions so far generated by perm and control transfers back to perm for the placing of the next queen as soon as the first no_take call suspends. Fortunately, because of the strict alternation between the no_take calls, and because each when reactivated does the same computation with the new queen position it has received, the suspension occurs only when all no_take calls, and the recursive safe call, are suspended. That is, all the checking of the last queen to be placed has been done before the next queen is requested by the first no_take call.

perm is the only non-deterministic call. Failure of any no_take call causes backtracking. The last slice of the safe computation is undone as is the last step of perm. Note that failure can be handled because perm is lazy. It suspends until safe suspends or fails. If we had evaluated perm concurrently with safe, on failure of some no_take call it would be difficult to determine the state of the concurrent perm computation to which we must backtrack. Hence the two options we discussed in Section 1. Either we allow the full parallelism of the process model, but disallow don’t know non-determinism and hence the need to handle multiple bindings. Or, we suspend any non-deterministic call until the rest of the computation deadlocks. We then only fork from and backtrack to deadlocked computation states. This is, by accident, what the IC-Prolog program does. The co-routining transfer from the safe call occurs when its pseudo-parallel evaluation is ‘deadlocked’. Transfer on deadlock, rather than transfer when a single consumer wants more data, is the control concept that is needed when we have multiple, parallel consumers.

Andorra and Pandora incorporate this control concept. They allow co-routining between a lazy non-deterministic producer (in fact several such producers), and phalanxes of parallel consumers. They transfer control back to a non-deterministic producer only when all the parallel consumers are deadlocked, i.e. have processed all the data generated by the last non-deterministic step of the producer. In Section 5 we shall give the Pandora version of Program 1.

### 2.2 Introducing committed-choice non-determinism

The first committed-choice logic programming language was RL. The computational model of this language generalised that of vEdL in that it allowed the possibility of more than one clause unifying with a call, even after the data input constraints for the call were satisfied. In addition, it introduced the concept of guards on clauses, conjunctions of calls to test-only primitives (now called flat guards) which must also succeed before the clause can be used. It also adopted the CSP concepts of committed choice, and communication (the binding of call variables) on commitment (not before).

Given a call, any candidate clause for the call can be used to reduce the call. A candidate clause is a clause whose input constraints are satisfied and whose guard is true. For a given call there may be several candidate clauses. There is no backtracking on the choice of the candidate clause.

Communication between calls (processes) in RL is restricted to the incremental generation of streams of variable free terms, or to a single ground term. The input constraints for clauses, and hence the data flow through shared variables of calls, is specified by mode declarations. For example, the mode declaration of Program 2 species that in any call ord_merge(L1,L2,L) the first and second arguments L1 and L2 are input and the third L is output. More precisely, it says that each clause for ord_merge has an input constraint that prevents the clause being used unless the first two arguments of the call input match (are substitutions instances of) the first two argument terms in the head of the clause. This is the meaning of the ? mode.

**Program 2**

mode ord_merge(?1,?,^).
ord_merge([],S2,S2).
ord_merge(S1, [],S1).
ord_merge([U1|S1],[U2|S2],[U1|S2]) :-
    U1 =< U2: ord_merge(S1,[U2|S2],S).
    ord_merge([U1|S1],[U2|S2],[U2|S2]) :-
    U1 =< U2: ord_merge([U1|S1],S2,S).
The first clause cannot be used unless L1 is \[ \]. (The second argument L2 of the call can be any term, even an unbound variable, for this will input match S2.)

The second clause cannot be used unless L2 is \[ \].

The third and fourth clauses cannot be used unless L1 is a term of the form \[ N1|L1 \] and L2 is a term of the form \[ N2|L2 \]. The guards of the clauses (the \( = \) and \( = \) calls before the \( : \) in each clause) Further require N1 and N2 to be numbers, but both L1|L2 and L2 can be variables. The third clause will be the only candidate if N1 = \( N2 \), the fourth the only candidate if N1 \( \neq \) N2, and both will be candidates if N1 = N2. In the latter case, one will be selected and the evaluation will commit to that clause.

On comm.\: ment to some clause for the call, the output binding to the third argument L of the call is made. L must be an unbound variable, and the term to which it is bound must either be a ground term (e.g. \[ \] ) or a list term of the form \[ E|S \] in which E is a ground term. S can be unbound. This is the meaning of the \( ^\cdot \) mode of RL.

Let us examine a computational scenario of some parallel conjunction

\[
\text{prod1}(L1) // \text{prod2}(L2) // \text{ord:\:merge}(L1, L2, L)
\]

where L1 and L2 are being incrementally produced by prod1 and prod2 respectively and ord:\:merge is incrementally consuming L1 and L2 and incrementally generating L.

The input constraints and guards of the clauses for ord:\:merge mean that the call suspends until either prod1 binds L1 to \[ \], or prod2 binds L2 to \[ \], or prod1 binds L1 to a term of the form \[ N1|L1 \] and prod2 binds L2 to a term of the form \[ N2|L2 \] where N1 and N2 are numbers. Suppose that N1 = N2 = 2 and the third clause is selected. On commitment to use the clause, the variable L of the call is bound to \[ [2|S] \], a term of the required form, and the ord:\:merge call reduces to ord:\:merge(L1, [2|L2], S), which will again suspend if L11 is an unbound variable.

The ord:\:merge program is such that its output is uniquely determined by its input even though the computation path is not uniquely determined. It is actually logically equivalent to a program which has an extra clause

\[
\text{ord:\:merge}([U|S1], [U|S2], [U|S]) \leftarrow \text{ord:\:merge}(S1, S2, S).
\]

and strict inequalities in the other two recursive clauses.

The evaluation of this modified program is deterministic. A genuinely indeterminate program, whose output is not determined by the input, is the arbitrary merge program given in Ref. 5 (Program 3).

### Program 3

\[
\begin{align*}
\text{merge}(?, ?, ^\cdot). \\
\text{merge}([\cdot], S2, S2). \\
\text{merge}(S1, [\cdot], S1). \\
\text{merge}([U1|S1], S2, [U1|S]) \leftarrow \\
\quad \text{merge}(S1, S2, S). \\
\text{merge}(S1, [U2|S2], [U2|S]) \leftarrow \\
\quad \text{merge}(S1, S2, S).
\end{align*}
\]

By examining different evaluation scenarios for a parallel conjunction

\[
\text{prod1}(L1) // \text{prod2}(L2) // \text{merge}(L1, L2, L)
\]

you will see that the L that is generated may depend upon the rate at which prod1 and prod2 generate the streams of terms on L1 and L2. Because merge is running in parallel, if prod1 puts 3 terms t1, t2 and t3 on L1 before prod2 generates any term on L2, merge will almost certainly place these three terms at the head of L in front of any terms generated by prod2. In other words, the interleaving computing will depend on the generation behaviour of the producer processes feeding into merge. This is an example of a behaviour that cannot be expressed in the vEdL model.

RL also allows the programmer to specify sequential execution of a conjunction, using the \& conjunction operator. In general, queries and bodies of clauses (the non-guard components) can be parallel conjunctions of sequential components, as in the vEdL model.

### 3. THE MAIN CONCURRENT LOGIC LANGUAGES

The data flow constraints of RL are very strong. The stream elements communicated between processes must be ground terms and the communication must be unidirectional. A consumer of a partial stream \([2, 3, 4 | L]\) cannot switch to become the producer of the rest of the stream \(L\), and partly instantiated terms such as \( f(2, X) \) cannot be transmitted. A syntactic check using the mode declarations ensures that these two communication constraints are satisfied. They have the major merit that they allow simple and efficient implementation on message passing, even distributed architectures. Ref. 5 gives an abstract operational model that can serve as the basis of such an implementation.

It is, however, very restrictive. Usefully, and without too much implementation overhead, one could allow alternation of the generation of elements on a stream between two communicating processes, using the single stream for two-way communication. More usefully still, we could allow the stream elements to be partly instantiated terms. The consumer of the stream can then further instantiate the received terms. When the partly instantiated term contains both a message and a variable, the binding of the variable can be viewed as the consumer replying to the message. For example, on receiving a message term such as \( \text{on}(2, X) \) (viewed as a message that asks what value is associated with the key 2 in some data structure held by the consumer process) the consumer could bind \( X \), thereby giving the reply. This is the basis of the implementation of objects as processes in the concurrent logic languages.46 Sending messages which are completely uninstantiated, i.e. unbound variables, enables one to emulate lazy evaluation and bounded buffer communication; see Ref. 30.

Parlog and the other main concurrent logic languages all allow such generalisations of the inter-process communication. But they all incorporate the major and key ideas of RL: (1) committed choice; (2) guards and candidate clauses; (3) constraints on unification determining the allowed inter-process communication and process suspension; (4) binding of call variables only on
commitment to some candidate clause. As such they are all descendants of RL.

3.1 Parlog

The version of Parlog described in Ref. 7 generalises RL in three major respects.

1) An output binding made on commitment to a clause can be any term. The call still has to have an unbound variable in each output argument position to receive the binding, but the binding does not have to be ground term or a list of the form \([E][S]\) in which \(E\) is ground. Thus, processes can incrementally generate and communicate tree-structured terms, not just lists, and terms communicated between processes via streams can contain variables. In addition, calls can alternate between being producer and consumer, sharing the incremental generation of a data structure.

2) The guards of clauses are not restricted to calls to test-only primitives, they can contain calls to program-defined predicates. This introduces a potential problem: a guard can now prematurely bind a variable in the call. Guards that are guaranteed not to bind variables in the call were called safe guards. It is algorithmically possible to confirm, using the modes, that all guards are safe (see Ref. 17); otherwise the programmer guarantees that they are safe.

3) The programmer can constrain the search for a candidate clause. The default in Parlog is that all clauses for a predicate \(r\) are tested for candidate status in parallel given some call to \(r\). By using ; as a separator between clauses, the programmer can delay the testing of the clauses that follow the ; until all preceding clauses have been tested and found to be non-candidates. This control device is useful for writing programs with a default clause.

Program 4 is a Parlog program exploiting most of these extensions. It is the implementation of an ordered tree object as a process. The process incrementally consumes a stream of messages generated by some other process/object, for example

\[
\text{insert}(1, \text{pears}), \text{insert}(2, \text{apples}), \\
\text{on}(1, V_1), \text{insert}(2, \text{bananas}), \\
\text{on}(2, V_2), \text{on}(5, V_3), \ldots
\]

It will reply to the on messages by binding the variables in these messages. Thus it will generate the following bindings:

\[
V_1 = \text{pears}, V_2 = \text{bananas}, V_3 = \text{no_value}
\]

as replies. Logically, tree_obj(M) should be read as ‘\(M\) is a legal sequence of messages to a labelled ordered tree object that starts with state the empty tree’. aux_tree_obj(M, T) should be read as ‘\(M\) is a legal sequence of messages to a labelled ordered tree object that starts with state the tree \(T\)’.

---

Program 4

```prolog
mode tree_obj(Messages?).
tree_obj(M) <- aux_tree_obj(M, e).

mode aux_tree_obj(Messages?, OrdTree?).
aux_tree_obj([], T). 
aux_tree_obj([insert(K, V)|T] <-
  tree_insert(K, V, T, NewT),
aux_tree_obj(M, NewT).
aux_tree_obj([on(K, V)|M], T) <-
  on_tree(K, T, Val) :
    V := Val,
aux_tree_obj(M, T); 
aux_tree_obj([on(K, V)|M], T) <-
  V := no_value,
aux_tree_obj(M, T).

mode tree_insert(Key?, Value?, Tree?, NewTree^). 

% insert into empty tree e

% insert new key and value

% K already on tree

% test if K is on T with non-flat guard
% bind V of message to assoc. value
% note the ';' 
% default clause for Key not on T

% K less than key at root
% insert in left subtree

% K greater than key at root
% insert in right subtree

mode on_tree(Key?, Tree?, Value^).
on_tree(K, t(L, n(K, V), R), V).
on_tree(K, t(L, n(K1, V1), R), V) <-
  K < K1 : on_tree(K, L, V);
on_tree(K, t(L, n(K1, V1), R), V) <-
  K > K1 : on_tree(K, R, V).
```

486 THE COMPUTER JOURNAL, VOL. 33, NO. 6, 1990
In Parlog the separator between calls denotes parallel conjunction. In the second aux_tree_obj clause the call tree_insert(K,V,T,NewT) will incrementally generate a tree structure that will be communicated to the parallel recursive call aux_tree_obj(M,NewT). This is an example of non-stream inter-process communication. The parallelism between the tree_insert and aux_tree_obj calls allows the aux_tree_obj process to accept and start handling the next message before the update associated with the previous insert message has terminated. aux_tree_obj consumes the new tree as it is incrementally generated by tree_insert.

The third aux_tree_obj clause has a non-flat guard: a call to on_tree, a relation which is defined by a program. The call on_tree(K,T,Val) generates a binding for the local variable Val if K is on T, otherwise it fails. The retrieved value, if there is one, is assigned to variable V of the incoming on(K,V) message by the single assignment V := Val in the body. Note that we cannot simply put V instead of Val in the guard call, for this would make the guard unsafe: we would be generating a binding for a variable given in the call argument inside the guard. In this case, the assignment in the unsafe guard would be harmless, for the guard would only assign to V if and when it succeeds, and in this case the computation would always commit to that clause. However, in general an unsafe guard may assign to the call variable and then fail, or it may assign and some other clause becomes the selected candidate clause. That is why we must avoid unsafe guards.

The first clause for on_tree generates a binding for a call variable, but a call variable that will be given in an output argument position. In fact, this first clause is logically and behaviourally equivalent to the clause

\[
\text{on_tree}(K,t(L,n(K,V)),R), V_1) \leftarrow \text{true} : V_1 := V.
\]

in which the assignment to the output argument of the call is done explicitly in the body of the clause.

Parlog as described in Ref. 17 uses unification instead of single assignment to generate output. That is, clause 3 of aux_tree_obj would be expressed as

\[
\text{aux_tree_obj}([\text{on}(K,V)|M], T) \leftarrow
\text{on_tree}(K,T,\text{Val}) ;
V = \text{Val}, \quad \% \text{unify } V \text{ with } \text{Val}
\text{aux_tree_obj}(M,T) ;
\]

This means that the process can accept messages of the form on(1, pears) as well as on(1, V1). However, in this case it is of dubious benefit since a received message on(1, apples) where apples is not associated with the key 1, or where 1 has no associated value, will cause the tree_obj process to fail.

Program 5 uses output unification to some benefit. This implements the 'same leaves' test for binary trees with leaves but no root labels. The use of unification rather than assignment allows either leaves call to put a leaf on the leaf profile. There is arbitrary two-way communication. Because it can fail, the program should only be called inside a guard. The program constructs the leaf profile using difference lists.

When we need to distinguish between the different versions of Parlog, we shall use Parlog(A) to refer to Parlog of Ref. 7 with output via assignment and Parlog(U) to refer to Parlog of Ref. 17 with output via unification.

The output unification of Parlog(U) is non-atomic. It will bind variables as and when they are paired with other terms. It does not wait until the entire unification has been determined to succeed before binding any variable, which is atomic unification. This means that an output unification may bind one or more variables and then fail. This does not matter, because in that case either the entire program has failed, or the call that generated the failed output unification occurred inside a guard. In the latter case, no binding generated will have been communicated outside the guard (the safety condition) and all the bindings will be discarded because the guard has failed.

There are two main implementations of Parlog(U) in current use: one based on Prolog with emulated parallelism, and a C-implemented emulator of a parallel Parlog abstract machine, which executes in parallel on shared-memory multiprocessor machines such as Sequent Symmetry. The Prolog-based implementation has good debugging facilities, for example one can set a spy point on a variable and have its incrementally generated binding displayed in a separate window. It is used mainly to teach the language. The multiprocessor C implementation exhibits near-linear speedups on a wide range of problems. We plan to convert this into a message-passing implementation suitable for non-shared memory architectures using ideas presented in Refs 15 and 23. However, we might restrict the implementation to Parlog(A) in order to avoid the need for distributed unification. Further information about both current implementations of Parlog can be obtained from the author.

3.2 GHC

GHC is very similar to Parlog(U). The main differences are the following.

1. It has no sequential conjunction.
2. There are no explicit mode declarations. Every non-variable term in the head of a clause must be used only for suspendable input matching. In Prolog terms, there is an implicit mode declaration that gives every argument a ? mode.
3. A clause guard must automatically suspend if it tries to bind a variable in the call.
(4) Output must be done by explicit unification calls in the body of the clause.
(5) There is no sequential clause search connective such as ;. Instead, at least in the KLI extension,4 there is a special guard primitive otherwise which only succeeds when all preceding clause guards have failed.

The GHC version of tree_insert is shown in Program 6. (GHC uses :- as in Prolog, instead of \(<\) and | instead of : to separate the guard from the body.)

**Program 6**

```prolog
\[\begin{align*}
\text{tree_insert}(K,V,e,T) :- \\
T &= t(e, n(K,V), e). \\
\text{tree_insert}(K,V,t, (L,n(K,V),1),R),T) :- \\
T &= t(L,n(K,V),R). \\
\text{tree_insert}(K,V,t, (L,n(K,V),1),R),T) :- \\
K &< K1 |
T &= t(NewL,n(K1,V1),R), \\
\text{tree_insert}(K,V,L,NewL). \\
\text{tree_insert}(K,V,t, (L,n(K,V),1),R),T) :- \\
K &> K1 |
T &= t(L,n(K1,V1),NewR), \\
\end{align*}\]
```

The lack of sequential conjunction in GHC can be compensated for, using control tokens. It requires more programming effort, but one can rewrite any Parlog program using sequential conjunction into a GHC program which uses only parallel conjunction. At the top level, instead of a conjunction such as

\[r(X,Y) \& p(Y,Z)\]

we use

\[r(X,Y), p(C,Y,Z)\]

where \(r\) and \(p\) have one extra argument. \(p\) is easily reprogrammed to suspend until \(C\) is bound, to, say, the atom done:

\[\text{p(done,Y,Z)} :- p(Y,Z).\]

The reprogramming of \(r\) is more complex. We must rewrite it so that it binds its third argument \(C\) if done when and only when all the processes it invokes terminate. The trick (due to Takeuchi) is to use a chain of variables linking all the processes that \(r\) invokes such that done will be passed from one end of the chain to the other when and only when all the processes successfully terminate.

Suppose \(r\) has a clause of the form

\[r(X,Y) \leftarrow r1(X,Z), r2(Z,Y).\]

We rewrite this as

\[r(X,Y,C) :- r1(X,Z,done,L), r2(Z,Y,L,C).\]

\(r1\) and \(r2\) are rewritten so that they unify their two extra arguments when they terminate, resulting in \(C\) being bound to done. If either forks, new variables are put into the chain and each new program that is invoked is made to unify its two extra arguments on termination. So, if \(r1\) has a Parlog program

\[\text{mode r1(?,-). \ r1([],[]). \ r1([U|T],[U1|T1]) \leftarrow q(U,U1), r1(T,T1).}\]

this becomes the GHC program

\[r1([],0,L1,L2) :- \text{unify_and_close}(0, [], L1, L2). \]
\[r1([U|T],0,L1,L4) :- \text{unify_and_close}(0, [U1|T1], L1, L2), \]
\[q(U,U1,L2,L3), \]
\[r1(T,T1,L3,L4). \]

where \text{unify_and_close} is defined as

\[\text{unify_and_close}(T1,T2,L1,L2) :- \]
\[T1 = T2, \]
\[\text{test_and_close}(T1,T2,L1,L2). \]
\[\text{test_and_close}(T,T1,L1,L2) :- \]
\[L1 = L2. \]

The test_and_close clause waits until \(T1\) and \(T2\) have been made identical by the unification call before unifying \(L1\) and \(L2\). The unifying of the link variables on termination either removes one variable from the chain or assigns done to the right variable of the pair. Ultimately, if all processes successfully terminate, done will have moved along the chain and \(C\) will be bound to done.

The major difference between GHC and Parlog is that GHC automatically suspends guards that try to bind call variables, whereas Parlog requires that all guards be checked at compile time for safety, or be guaranteed to be safe by the programmer. Checking during evaluation is an elegant concept but is quite complex to implement. In fact, nearly all implementations of GHC implement only the flat variant, FGHC. For flat programs, as with Parlog, a simple restriction that guards comprise calls to test-only, automatically suspending primitives guarantees safety and removes the need for a runtime check.

Every program in FGHC can be trivially transcribed into a Flat Parlog(U) program. By using control tokens and short circuits instead of & every Flat Parlog program can be reprogrammed into FGHC.

In fact, Flat Parlog augmented with a suitable metacall, call(\(\text{Conj}\), \(\text{Result}\), \(\text{Control}\)), which signals success and failure of \(\text{Conj}\) by output bindings to \(\text{Result}\), and which prematurely terminates its evaluation if \(\text{Control}\) is bound to stop by some other process, can be used to implement safe Parlog with full guards. The details are given in Ref. 17. This technique is used in the C implementation of Parlog for procedures with parallel non-flat guards. FGHC cannot be used to implement full GHC because of the requirement for the runtime binding check, absent from FGHC.

FGHC has a C-based implementation for shared-memory multiprocessor Unix machines similar to that for Parlog. The two are compared in Ref. 32. A major difference is that the Parlog implementation supports sequential execution using reference counts on processes that record the number of active subprocesses. There is also a multi-machine implementation of GHC/KLI using specially micro-coded PSI processors.29

### 3.3 Strand

In both Parlog and GHC the output is done as a body call. In their flat variants, the use of unification rather than assignment for output is thus of little benefit. One cannot guard against a failed output unification by enclosing the call in a guard. All output unifications must succeed or the entire computation will fail. (If one could
test whether the output unification would succeed as part of the guard, and only commit to using the clause and doing the output unification if it would succeed, that would be a different matter. This is what one can do in FCP, the flat version of CP. But it needs atomic unification, which is quite complex to implement. More of this later, when we discuss CP.

In practice, in Flat Parlog or FGHC one usually programs in such a way that output arguments of a call are always unbound variables and output unification is always just single assignment, as in RL and Parlog(A).

Strand\textsuperscript{16} is a restriction of Flat Parlog(A). It has no sequential conjunction, and it restricts identity testing of input arguments to ground terms. That is, in the Parlog clause

\[
\text{mode ord_merge(?, ?^).}
\]
\[
\text{ord_merge([U|S1],[U|S2],[U,U|S])} \leftarrow
\]
\[
\text{ord_merge(S1,S2,S).}
\]

the implicit identity test that the head elements of the two input lists are identical will succeed if there are any identical terms, including two terms that are just the same variable. In Strand, this test will only succeed if the heads are identical ground terms.

The use of output assignment and the restriction of identity testing to ground terms simplifies a message-passing implementation. (Strand is a commercial product of AI Ltd, England.) The implementations are mostly C-based emulators of an abstract machine. There are implementations for shared-memory multiprocessors and message-passing architectures. The shared-memory implementations offer a similar performance to that of the C-based Parlog, because the Strand restrictions are only of benefit for a message-passing implementation.

3.4 Concurrent Prolog

Concurrent Prolog (CP)\textsuperscript{27} differs from the other languages in three important respects.

1. The data flow through shared variables is specified by a \texttt{?} annotation on the occurrence of the variable in the consumer calls, not by an input match restriction on the programs they invoke. A variable annotated \texttt{?} in a call cannot be bound by the program it invokes. This is similar to the \texttt{?} annotation of IC-Prolog but with a subtle difference. A call with an annotated variable \texttt{X} in IC-Prolog cannot be selected for reduction until \texttt{X} is bound. In CP, this is the case only if each clause for the call tries to match \texttt{X} with a non-variable term. If some clause matches it with a variable \texttt{U}, \texttt{U} will be bound to \texttt{X}, effectively transferring the read-only annotation to \texttt{U} in the body of the clause. In addition, variables in the heads of CP clauses can be given \texttt{?} annotations. This allows output to be protected from update by consumer processes, even if the program calls to these processes do not have \texttt{?}-annotated variables.

2. There is no separation of the head arguments of a clause into input and output arguments. For a call \texttt{r(X, Y)}, \texttt{X} is input, for a call \texttt{r(X, ?Y)}, \texttt{Y} is input. In each case the other argument is output providing it has not been previously bound to some \texttt{?}-annotated variable.

3. In a CP program non-flat guards do not need to be safe, and they are not suspended if they try to generate bindings for call variables. The guard computation of a clause is allowed to generate bindings for call variables, but all these generated bindings, and any bindings that need to be made to the call variables by the head unification, are delayed until the computation commits to the clause. The computation will only commit if all the bindings can be made. They are then all made as a single atomic multiple assignment as part of the commitment to the clause. This is a generalisation of atomic unification, for it extends the atomic assignment beyond the bindings generated by the call/head unification to those generated by the guard.

The CP program for \texttt{tree_insert} is shown in Program 8; its call in the \texttt{aux_tree_obj} CP program must be expressed as

\[
\text{tree_insert(K?, V?, T?, NewT).}
\]

Interestingly, for this program, and for many of the programs written in CP, the weaker ! control annotation of IC-Prolog gives equivalent behaviour.

\textbf{Program 8}

\[
\text{tree_insert(K, V, e, t(e,n(K,V),e)).}
\]
\[
\text{tree_insert(K, V, t(L,n(K,V1),R),}
\]
\[
\text{t(L,n(K,V),R)).}
\]
\[
\text{tree_insert(K, V, t(L,n(K1,V1),R),}
\]
\[
\text{t(NewL,n(K1,V1),R))} \leftarrow
\]
\[
\text{K < K1 :}
\]
\[
\text{tree_insert(K?, V?, L?, NewL).}
\]
\[
\text{tree_insert(K, V, t(L,n(K1,V1),R),}
\]
\[
\text{t(L,n(K1,V1),NewR))} \leftarrow
\]
\[
\text{K > K1 :}
\]
\[
\text{tree_insert(K?, V?, R?, NewR).}
\]

The semantics of unification with \texttt{?}-annotated variables is not simple, and needed to be revised to remove ambiguities in the original definition. (For example, what happens when two \texttt{?}-annotated variables are matched.) In addition, the need to hold back bindings made by non-flat guard computations means that an implementation of full CP requires multiple bindings for variables to be stored. In fact, one can very easily define in CP an or-parallel meta-interpreter for Prolog using non-flat guards to store multiple bindings, see Ref. 28.

The last complexity has meant that nearly all implementations are of FCP (Flat CP). Flat guards need not be evaluated in parallel, they can be tried in turn until a candidate clause is found, provided the next clause is tried when the preceding clause suspends as well as fails. Thus multiple bindings for call variables do not need to be stored. However, local copies of bindings that need to be made to call variables as a result of the call/head unification need to be stored as each clause is tried, and the call variables that will be bound need to be locked to prevent some other process binding them before the guard computation terminates. If the guard suspends, or fails, locked variables need to be undone. If the guard succeeds, all the locked call variables are assigned the locally stored values. An implementation also has to be careful to avoid indefinite suspensions when two processes want to bind overlapping sets of variables. One of them must be prepared to release binding permission if each has obtained binding permission for some but not all the variables.\textsuperscript{31}

Atomic unification with read-only variables is not simple to implement, but it is a powerful feature. One can
write programs in FCP that one cannot transcribe to the other languages. Ref. 28 has several examples. Whether or not this expressive power is worth the implementation overhead is another matter.

One can write programs in which processes compete to bind a shared variable to different values. By testing whether the binding can be made before attempting the binding, one can avoid the failure of processes that are not first in their attempt to bind the variable.

Program 9 is a simple example. It is a program that searches a loop-graph from some start node s to find a node N, a term of the form n(V1, V2, V3), that has some particular value V as its first argument. It forks to pursue all paths from each node in parallel. The graph is represented by a program for the relation next_pages(Node, List_of_neighbours). The program is called with find_node(S, V, N), where N is a variable to be bound to a node reachable from S containing value V.

Program 9

```
fnd_node(S, V, N) :- wait(N) | true.
% N found by another process
fnd_node(n(V, V2, V3), V, n(V, V2, V3)).
% this process has found N
fnd_node(S, V, N) :- otherwise |
next_pages(S, L),
% find all neighbours of S
fnd_from_all(L, V, N).
% search from each in parallel
fnd_from_all([], V, N).

\text{fnd_from_all}([S|L], V, N) :-
fnd_node(S, V, N),
\text{fnd_from_all}(L, V, N).
```

The wait(N) guard succeeds if N is bound to a non-variable. It suspends if N is unbound. (All the concurrent languages have equivalent test primitives.) The first clause terminates each forked search as soon as some search path finds a required node. The atomic test unification of the second clause for fnd_node is needed because whilst the first two arguments of call and clause are being unified some other process could bind N to a term that does not unify with n(V, V2, V3).

The Parlog, GHC, and Strand versions have a second clause that is equivalent to

```
fnd_node(n(V1, V2, V3), V, N) \leftarrow
V == V1 : N := n(V, V2, V3).
```

and the computation will commit to the clause before trying to bind N to the found node. Between commitment and assignment some other find_node process could bind N to a different term causing the entire find_node computation to fail. (There could be another node in the graph having V as first argument, which per chance is found at exactly the same time.)

In this case, as in many other cases where atomic unification is used, we can make do with a much simpler device of atomic test assignment, an idea proposed in Ref. 2. This is a minor variation on an idea independently proposed by Ueda (personal communication) in the original design of GHC, but later abandoned. Ref. 2 proposes a two-part guard for Flat Parlog(A), which we shall write in the form G: A: . G is a normal guard, A is a single assignment \( V := T \) to a variable V. If G and the input matches succeed, as part of commitment the assignment to V is attempted. But if V is already bound, the combined guard simply fails, allowing another clause to be selected as candidate clause. Using this concept we can rewrite the second clause for fnd_node as

```
\text{fnd_node}(n(V1, V2, V3), V, N) \leftarrow
V == V1 : N := n(V, V2, V3) : true.
```

This gives the desired behaviour.

It is the author's opinion that Flat Parlog(A), with one atomic test assignment allowed in each clause (as well as any number of normal body-call assignments), will adequately cover the required applications of committed-choice concurrent logic programming. A single atomic test assignment is very much easier to implement than full atomic unification.

The main implementation of FCP is a message-passing implementation\(^{31}\) for non-shared-memory multiprocessors.

3.5 Concurrent constraint programming

Maher\(^{44}\) presents an abstract model, called ALPS, which generalises concurrent logic programming to concurrent constraint logic programming. Clauses have the form

\[ H \leftarrow \text{Ask} | \text{Tell}, B, \]

where Ask and Tell are conjunctions of calls to a set of reserved constraint predicates. Computations now generate a set S of mutually consistent calls to constraint predicates. When a clause is used to reduce a call, the Tell calls of the clause are added to the current partially constructed S. If they are inconsistent with S, the computation fails.

The Tell calls generalise the output unification calls of Parlog and GHC, which we can consider are added to a partially constructed set of = calls when a Parlog or GHC clause is used. (In practice all the = calls are reduced to variable bindings, but logically these are equivalent to the = calls from which they are constructed.)

As in the concurrent languages, ALPS calls to program-defined relations are evaluated concurrently, and clauses are tested for candidate status in parallel. Suppose S is the current partially constructed set of consistent constraint calls, the accumulated Tell calls of clauses used for previously reduced calls. A clause is a candidate for a new call C if, for all assignments of values to variables in S, C that make S true, there is some assignment of values to variables of H and Ask that makes C = H, Ask true. In this case, C = H, Ask is said to be a valid guard for the call C in context S. The validity condition ensures that the guard cannot put additional constraints on variables in S or C. This is the logical formulation of the Parlog/GHC 'no assignment to call variables' requirement for guards.

A call can be reduced using any candidate clause, with commitment to the clause. In addition, ALPS allows a call to be reduced using a clause which does not qualify as a candidate clause provided it is the only clause with a guard that is satisfied. The guard is satisfied if the conjunction C = H, Ask, S is true for some assignment of values to its variables. The satisfiability condition allows the guard C = H, Ask to put additional constraints on variables in S and C. That is, the 'read only' restriction
of guard computations can be relaxed if there is only one clause that can be used – the deterministic and-parallelism idea again.

Ref. 25 extends these ideas to allow two phase guards which are a generalisation of two phase guards for Flat Parlog that we discussed in Section 3.4. Clauses are of the form:

\[ \text{H} := \text{Ask} : \text{Tell} \mid \text{B} \]

The head/call unification and Ask component must satisfy the ALPS validity condition. However, after this has been determined to hold, there is still only commitment to use the clause if the Tell constraints can be determined to be consistent with the current set S. If they are, they are automatically added to S. This is the constraint logic programming generalisation of atomic unification tested in the guard.

4. EXTENSIONS FOR OBJECT-ORIENTED PROGRAMMING

A'UM, Parlog++, Polka and Vulcan are concurrent object-oriented programming languages built on top of concurrent logic languages in which messages are logical terms. A'UM is built on top of KL1, Vulcan on top of FCP, Parlog++ and Polka on top of Parlog. All are implemented as compilers to the underlying language. Parlog++ and Polka differ from the other two in allowing access to the underlying language; in fact this is an important feature of these languages. We shall illustrate Parlog++.

Program 10 is Program 4 written as a Parlog++ program.

Program 10

\[
\text{parlog++} \quad \text{tree_obj}
\]

invisibility state \( T <= e \) clauses

last => true.
insert(K,V) => \( \text{tree_insert}(K,V,T,\text{NewT}), T \text{ becomes } \text{NewT.} \)
on(K,V) => \( \text{on_tree}(K,T,\text{Val}) : V := \text{Val;} \)
on(K,V) => \( V := \text{no_value.} \)

code
{ordinary Parlog programs for 
\text{tree_insert, on_tree}}
end

Things to note.

(1) There is no explicit recursion and no explicit mention of an input stream of messages. Every Parlog++ definition is for a relation that has at least one input stream and which is recursively defined.

(2) The clauses rules correspond to the clauses given in Program 4 for aux_tree_obj. The auxiliary relation is implicitly declared by the invisible state \( T \) declaration. This declares \( T \) an extra argument of \( \text{tree_obj} \) that should not appear in the top level call (i.e. which is invisible to other objects). \( T <= e \) says that this extra argument is initialised to \( e \). This declaration is equivalent to, and creates the \( \text{tree_obj}(M) \leftarrow \text{aux_tree_obj}(M,e) \) clause of Program 4.

(3) Only the pattern for the message term is given in at the head of each message-processing rule. The right-hand side of the rule indicates what processing should follow the receipt of the message. Where there is an update of the invisible \( T \) argument this must be given as a becomes condition, as in the second clause.

(4) The processing rules can have guards and be separated by ; to give a default rule.

(5) The predicates used in the code section are local to the object; they can only be called by the message-processing rules of the object or by other programs in the code section.

(6) The action to be taken when the input stream is closed is given by a rule for the reserved \( \text{last} \) message.

Parlog++ is a flat object-oriented system in which messages must be explicitly sent to other objects, as with actors. In Parlog++ it is done by having explicit output streams which are linked with input streams in the initial call to the object, which is a normal Parlog call. For example, \( \text{other_obj}(\ldots, M), \text{tree_obj}(M) \) allows \( \text{other_obj} \) to talk to a new invocation of \( \text{tree_obj} \).

Polka is an extension of Parlog++ that supports object hierarchies and hence implicit communication between objects as well as explicit communication. Messages that cannot be handled by an object are automatically sent up the object hierarchy, and finally to a default error-handling object. Polka also allows explicit sending up the hierarchy, the super destination for a message, and sending a message back down the hierarchy, the self destination. It has many other features which combine to give a powerful, concurrent object-oriented extension of Parlog.

5. EXTENSIONS FOR CONSTRAINED SEARCH

We have already introduced in Sections 2 and 2.1 the main control concepts of Andorra and Pandora. We shall exemplify their use using Pandora. This is an extension of Parlog which allows calls to ‘don’t know’ non-deterministic Prolog-style relation definitions. These definitions are expressed as sequences of rules of the form

\[ \text{H} := \text{G} : \text{B} \]

where \( \text{G} \) is a flat guard. There is no mode declaration and the Prolog := operator is used instead of \( \leftarrow \).

The ‘don’t know’ (DK) relations defined by such rules must be disjoint from the ‘don’t care’ committed-choice (CC) relations, defined by Parlog programs. However, the bodies of both the Prolog and the Parlog clauses can have calls to both types of relations.

The computation begins with a normal and-parallel Parlog style evaluation. In a parallel conjunction, all calls are evaluated concurrently, including the calls to the DK relations. CC relation calls suspend as in Parlog, when there is an input suspended clause and as yet no candidate clause for the call. DK relation calls suspend unless it can be determined, without binding variables in
the call, that there is only one clause with a head that will unify with the call and which will have a true guard. If there is only one such clause, the call is unified with the head (with no restrictions on the binding of call variables) and reduced to the body of the clause. Using these reduction rules, the computation may succeed, fail or deadlock. If it deadlocks, and there are suspended DK calls, the computation forks into alternative evaluation paths. One of the DK calls C is selected, using a user-definable meta program, or a default rule if none is supplied. (This is a major difference between Pandora and Andorra. The latter always selects the leftmost such call using the Prolog computation rule.) For each clause that unifies with C and which has a true guard, a new or-branch of the computation is started by reducing the call using that clause. Current prototype implementations of Pandora try the alternative evaluation paths using backtracking. Andorra uses or-parallel search. The use of one of the clauses for C usually allows several suspended calls to reduce, allowing a new phase of and-parallel evaluation. Failure of a call now just causes failure of some branch of the evaluation, as in Prolog.

Program 11 is Program 1 rewritten in Pandora (supplied by Reem Bahgat).

Program 11

mode eight_queens(\^).
eight_queens(L) ←
perm([1, 2, 3, 4, 5, 6, 7, 8], L), safe(L).
mode safe(?).
safe([]).
safe([Q|R]) ← no_take(Q, R), safe(R).
mode perm(?, ^).
perm([], []).
perm([U|L], [V|PermL]) ←
delete(V, [U|L], List),
perm(List, PermL).
delete(U, [U|L], L).
delete(V, [U|L], [U|DelL]) :-
delete(V, L, DelL).
{+ suitable Parlog definition of no_take for mode no_take(?, ^)}

Note that delete is the only DK relation and the definition of perm is the only extended Prolog definition - the only one that calls a DK relation.

Given a call eight_queens(L), it is immediately reduced to the two and-parallel calls
perm([1, 2, 3, 4, 5, 6, 7, 8], L), safe(L)
The safe call immediately suspends, but the perm call can be reduced to
delete(V, [1, 2, 3, 4, 5, 6, 7, 8], List),
perm(List, PermL)
both of which now suspend. The delete call suspends because both clauses of its DK definition unify with the call.

The reduction of the perm([1, ..., 8], L) call binds Lo to [V|PermL]. This allows the suspended safe call to be reduced to
no_take(V, PermL), safe(PermL),
both of which suspend.

We now have deadlock, broken by reducing the only DK call, delete(V, [1, 2, 3, 4, 5, 6, 7, 8], List), using one of its clauses. The computation continues, with phases of and-parallel evaluation of the safe computation coinciding with suspended evaluation of perm. On deadlock of the safe computation, the `lazy' perm evaluation uses one of the clauses for delete to generate a new queen placing, as in the IC-Prolog program. But the non-deterministic generation occurs when we know that all the useful checks that safe can make have been made, not as in IC-Prolog when one of the no_take calls demands the next queen position. Failure of a branch leads to backtracking to the most recent deadlocked state for that branch for which there is an untried delete clause. This is exactly as in IC-Prolog.

Ref. 4 gives other constrained search applications of Pandora, and Ref. 37 gives similar applications of Andorra, while Ref. 18 describes and illustrates another variant, called Andorra Prolog. This is close to Pandora in that it also allows calls to committed choice programs.

Acknowledgements

This paper was written whilst the author was visiting the Department of Computer Science, University of Queensland, which provided a very congenial and supportive environment. It also benefited considerably from discussions with Reem Bahgat and Andrew Davison and from comments by Steve Gregory.

REFERENCES


492 THE COMPUTER JOURNAL, VOL. 33, NO. 6, 1990