The Knowledge Engineering Review, Vol. 4: 4, 1989, 305–332

A Comparative Exploration of Concurrent Logic Languages

G. A. RINGWOOD
Department of Computing, Imperial College, LONDON SW7 2BZ,UK

Abstract

The execution model of Prolog, the first popular language based on Horn Clauses, was designed for efficient evaluation on von Neumann architectures. An alternative process model of execution, better suited for parallel evaluation and reactive programming, has given rise to a new class of languages based on Horn Clause logic, concurrent logic languages. There appears to be a profusion of languages which claim to fall into this class and it is difficult for an initiate to appreciate why each is the way it is. One notable member of this class, FGHC, forms the cornerstone of the Japanese 5th Generation Initiative. Fortunately, the seemingly exponential growth in these languages is only an illusion. A finite number of synchronization mechanisms arise from attempting (or sometimes not attempting) to control two principle synchronization difficulties: the premature binding problem and the binding conflict problem. Suitable combinations of these synchronization mechanisms reproduce the languages of this family. A background knowledge of Prolog is assumed and some familiarity with the difficulties encountered in concurrency would be advantageous.

1 Introduction

The Japanese Fifth Generation Initiative can be recognized as yet another incarnation of the ancient dream of producing intelligent machines (Ringwood, 1988b). The large gap between applications and hardware advocates a stratified methodology to the design of such symbolic processing machines. Top-down and bottom-up approaches to design are plausible and in principle both should produce the same machine. This is not the case in practice; compromizes made at different stages in the design with the different methodologies result in disparate architectures. The Japanese initiative is based on the middle-out philosophy: working both upwards and downwards from some middle ground should provide a better synergy between hardware and applications.

Much of bottom-up design has been encapsulated in computational models and much of top-down design has been encapsulated in symbolic knowledge representation systems. Programming languages embody both these concepts and so form a suitable starting point for the middle-out strategy. There are a number of features of Prolog which make it attractive as the middle layer (Ringwood, 1988b). The pattern directed procedure invocation mechanism of Prolog leads to some elegant programs. It has a logic based semantics which makes it easy to understand programs and their output. It has been claimed that Prolog programs are closer to specifications than almost any other programming language. In addition, the language has a great affinity for metainterpretation and partial evaluation which make it highly attractive for designing the flexible control systems required in AI applications. Languages that can handle programs as data make it natural to bootstrap new languages or extended control features by programming interpreters. Such metainterpretation makes it possible to solve complex problems at a higher level of abstraction. Metainterpreters, naturally enough, run slower than direct execution so that flexibility is gained at the expense of efficiency. (Is this not always the way?) To some extent, the fashionable transformation technique of partial evaluation, for which Prolog is eminently synergetic, mitigates

this disadvantage. Partial evaluation takes a program and a partial specification of the data and combines them to produce a more efficient but more specialized program. In this way partial evaluation can be used to combine metainterpreters and object programs.

Using metainterpretation and partial evaluation, user oriented specification languages can be implemented in Prolog with a minimal amount of effort. For these reasons among others (Ringwood, 1988b), the Japanese chose Prolog as the starting point for their middle-out strategy. However, Prolog is not at all well adapted for evaluation on parallel hardware: exhaustive search has been much criticized and vital programming tasks, reactive systems (such as environments and operating systems), are difficult to express cleanly in the language. In particular the depth-first, backtracking search of Prolog is designed for efficient implementation on von Neumann architectures and cannot easily provide the dynamic resource allocation which is a major concern of reactive systems.

Deciding that pattern matching, metainterpretation and partial evaluation are indispensable features of Prolog but exhaustive search is expendable allows the possibility of exploring alternative computational models for Horn Clause logic. Two types of computation can be distinguished, transformational and reactive (Harel and Pnueli, 1985). Transformational systems begin with the input of data, transform the input, output the result of the transformation and terminate. On the other hand, reactive systems continuously interact with the environment. Their reason for being is not the output they produce on termination but, rather, their ongoing interaction with their environment; sometimes reactive programs are effectively perpetual (they do not terminate even when the machine is switched off, for example the clock). The earliest programs, algorithms, were of the transformational type but an increasing number of programs today are of the reactive type. Systems programs such as environments with multiple windowing systems, operating systems and event driven realtime systems are examples of reactive programs. Reactive systems consist of conceptually parallel processes that communicate with each other and together respond to events that occur indeterminately in realtime. For example, the resource management processes which make up an operating system cooperate to provide a virtual machine to a user.

Closer examination of the Prolog computational model reveals the possibility of a concurrent computational model. For Prolog the exploitation of parallelism is complicated by the nondirectionality of the modes of variables, and backtracking (e.g. Ringwood, 1988a). The computational model of Prolog is stack based because this is most suitable for implementation on a von Neumann machine. Depth-first search and chronological backtracking is a natural partner of a stack based implementation. Essentially, in a resolution step the selected literal is the top of the goal stack and the newly introduced literals from the body of a unifying input clause are pushed on top of the stack. This gives rise to the well known, hierarchical, subroutine computational model of Prolog in analogy with block structural languages, such as Pascal. An alternative model more suitable for medium grain parallelism can be realized by means of a goal queue. The selected literal is the head of the queue of literals; new literals from an input parent are added to the end of the queue. This gives rise to a process interpretation (or possibly object oriented interpretation) of goal literals and this observation, that goals can be treated as processes (objects), is the essence of concurrent logic languages. For two descriptions of how these languages may be derived from Prolog see Ringwood (1987a, 1988a).

There have been a large number of languages using the process model of Horn clauses and the family is still growing. With a few exceptions they are the output of three competing research teams: ICOT (Institute of New Generation Computer Technology), Japan; Imperial College, UK; and the Weizmann Institute, Israel. These languages have sometimes, rather mistakenly, been dubbed committed choice languages because they allow the programmer to deliberately prune the search space. While they do embody committed choice, real Prolog, with the cut (as opposed to pure Prolog without the cut), can also be described as a committed-choice language (as for that matter can the parallel P-Prolog (Yang and Aiso, 1986)). These other committed-choice languages are not the concern of this review. The vital ingredient of concurrent logic languages is the process based computational model. Fortunately, the seemingly exponential growth in the number of these

languages is only an illusion. There is a basic theme around which a number of options have developed in response to two synchronization difficulties: the premature binding problem and the binding conflict problem. It is the purpose of this paper to enumerate the principle synchronization mechanisms and emphasize the similarity between the languages. To this end some liberties have been taken with the original syntax (and some of the language designers might claim with the operational semantics).

The family of concurrent logic languages is introduced by describing the simplest member of the family, Guarded Definite Clauses, denoted GDC (Ringwood, 1987a). This archetype serves to expose the two principle synchronization problems which compromise the operational behaviour. It also serves to illustrate that Prolog's affinity for metainterpretation is preserved in concurrent logic languages but unlike Prolog, where search is usually implicit, search in concurrent logic languages is more usually explicitly programmed. The two synchronization problems can be circumvented by programming or alternatively treated by the addition of runtime or compiletime synchronization mechanisms to the operational semantics of the language. This is illustrated with an extension, AGDC, of GDC. In the penultimate section, the whole spectrum of synchronization mechanisms is uncovered by a process of incremental language modification starting from GDC. The small set of problems used to introduce GDC is used to illustrate and compare the different languages.

This paper should be considered as a sequel to a previous paper by the author (Ringwood, 1988b) explaining in greater detail a rationale behind Japanese Fifth Generation. Unlike the previous paper, a background knowledge of Prolog is essential and some familiarity with concurrency should prove advantageous.

2 The simplest member of the family: GDC

Following (Ringwood, 1987a) a representative concurrent logic programming language, GDC, is introduced using a well known Prolog example. The following are the three familiar clauses for partition used in the Prolog Quicksort program.

```
partition([ ], Pivot, [ ], [ ]) ← !.
partition([H | Ts], Pivot, [H | Ls], Gs) ← H = < Pivot, !,
   partition (Ts, Pivot, Ls, Gs).
partition([H | Ts], Pivot, Ls, [H | Gs]) ← Pivot = < H, !,
   partition(Ts, Pivot, Ls, Gs).
```

In this example, it is to be noted that the clauses are almost determinate; that is, they are almost mutually exclusive. (They can of course be forced to be determinate by replacing the test, Pivot = < H, in the last clause by strict inequality, but this would destroy an opportunity to make an important point.) The choice between the second and third clauses is only determined by the comparison tests before the cut in the bodies of the clauses. The cuts are metalevel directives to the interpreter to take advantage of the mutual exclusion and prevent backtracking; the cut prunes the stack to the most recent previous choice point.

Exhaustive search in the form of chronological backtracking can be avoided if the clause choice is determined not only by goal clause-head unification but by the combination of the clause head and the comparison test; a complex inference step of a nature similar to paramodulation or typed inference. The replacement of the clause head by a conditional is made explicit in the archetypal concurrent logic language, GDC. The GDC equivalent of the Prolog relation for **partition** is

```
partition([ ], Pivot, Ls, Gs) ← true ←
   Ls = [ ], Gs = [ ].
partition([H | Ts], Pivot, Ls, Gs) ← H = < Pivot ←
   Ls = [H | L1s], partition(Ts, Pivot, L1s, Gs).
partition([H | Ts], Pivot, Ls, Gs) ← Pivot = < H ←
   Gs = [H | G1s], partition(Ts, Pivot, Ls, G1s).
```

The head of each Prolog clause has been substituted by a conditional called the *guard.* (Implication is assumed to be left associative so as to avoid bracketing.) The **true** proposition denotes an empty guard condition and is inserted to illustrate the mandatory presence of a guard for every clause. The relegation of some unifications to the bodies of clauses is explained a little later. (This notation makes the clauses logically equivalent to the previous Prolog ones, when of course the cut is ignored.) The guard conditionals are used to restrict the choice of clauses which can be used to reduce a goal literal. Only those clauses for which the conditional is satisfied by the call arguments are candidates to reduce a call.

As noted, the second and third guarded clauses of the partition relation are not quite mutually exclusive. If the head of the list which is to be partitioned is equal to the Pivot then both the second and third clauses are candidates to reduce the goal. In GDC, one of these is arbitrarily chosen, a process known as *committal.* In this way the language provides indeterminacy. In nondeterminate systems all branches at a choice point in the computation are fully explored. This in general gives rise to a number of possible conclusions and of course the attendant combinatorial search explosion. With indeterminacy, only one branch at a choice point is pursued; this will give at most one possible solution with the possibility of incompleteness. Prolog is nondeterminate whereas GDC is indeterminate. Indeterminacy is thought to be desirable for reactive systems: Dijkstra (1976) argues that programmers should not be required to specify details that are inessential to solving a given problem. Not only is it wasteful of programmers time, it over specifies the required functionality of the program. When the above example is used as part of a Quicksort program, if the head of the list to be partitioned is equal to the *Pivot* it is irrelevant whether the second or third clause is chosen.

In the Prolog relation for **partition** the decision of which clause to use to reduce a goal is made purely on the basis of call-head unification and backtracking is used to roll back a choice which later proves to be inappropriate. In GDC of the candidate clauses, those for which the guard is satisfied, only one is chosen so there is no recanting. In this way the second implication symbol in a clause is comparable with the Prolog cut, but in GDC its effect is symmetric, unlike Prolog where it is dependent on clause order. This is a declarative improvement over Prolog, but this advantage is gained at the expense of loss of the control achieved by the ordering of clauses and backtracking. Because of indeterminism an inappropriate decision based on a guard conditional may be doomed to failure. This is the first intimation of a synchronization problem.

Because there is no backtracking on the clause committed, it seems foolhardy to perform any binding of variables which appear in a goal before the appropriate choice of clause to commit to has been made. This synchronization problem between the irrevocable binding of call variables and goal reduction will is called *premature binding.* With GDC, the strategy of ensuring that processes make suitably considered choices of alternative actions before binding variables is to provide the programmer with a synchronization mechanism to replace the control synchronization of clause ordering and backtracking which has been lost from Prolog. The idea is to delay variable binding and process reduction until enough information is available so that an informed choice of which clause to commit to can be made. So in the example given, the first clause will only be considered as a candidate for committal if its first argument is bound to the constant [ ] at the time of call. The second and third clauses are only candidates for committal if the first input argument has been bound to an instance of the pattern [H | Ts]. This feature can be compared with Robinson's (1965) introduction of the most general unifier in Resolution theorem proving which postpones choosing values for quantified variables until dictated by the facts at hand (as explained in Ringwood, 1988b). More generally, the reduction of a goal process ← **a** by a relation suspends until the goal pattern matches the head of some clause and the guard condition of that clause is satisfied by the input pattern.

This can be illustrated syntactically by explicitly including the head pattern matching in the guard constraints

partition(Xs, Pivot, Ls, Gs) ← Xs < = [ ] ←
  Ls = [ ], Gs = [ ].

partition(Xs, Pivot, Ls, Gs) ← Xs < = [H | Ts], H = < Pivot
← L2s = [H | L1s], **partition**(Ts, Pivot, L1s, Gs).
partition(Xs, Pivot, Ls, Gs) ← Xs < = [H | Ts], Pivot = < H
← Gs = [H | G1s], **partition**(Ts, Pivot, Ls, G1s).

This is a perfectly legitimate GDC program and is a source to source transformation which is part of the compilation process of GDC. The primitive < = is used to signify the pattern matching of the input arguments; this syntax further explains why the head of a clause is considered as part of the guard.

For Prolog, both input and output is performed by unification but is deeply intertwined with goal reduction. In GDC, to try to resolve the premature binding problem input and reduction are separated from output; Prolog's clause-head unification invocation mechanism is replaced by a condition synchronization invocation mechanism, the guard conditional. The purpose of the guard is for clause selection and parameter passing only. Output, instantiation of goal variables, is sequenced after committal by relegating it to the body of a clause where it is made explicit by an equality predicate, =. The equality predicate is a privileged primitive which can effect binding of call variables by the unification of its two arguments. As will be seen in the next section, GDC's sequencing of constraints before output does not completely resolve the binding conflict problem.

The bodies of clauses of a relation in a GDC program represent subsequent alternative process behaviors, *continuations*. The body of a clause is not evaluated until the choice of clause is committed. Goals in the body of a clause are considered as spawned processes. Roughly speaking a "," in the "body" of a clause operationally means "fork" a new process. Again, these ideas can be represented syntactically with a *homogeneous syntax*:

partition(Xs, Pivot, Ls, Gs) ←
Xs < = [ ] ← Ls = [ ], Gs = [ ]
+ Xs < = [H | Ts], H = < Pivot ← L2s = [H | L1s], **partition**(Ts, Pivot, L1s, Gs)
+ Xs < = [H | Ts], Pivot = < H ← Gs = [H | G1s], **partition**(Ts, Pivot, Ls, G1s).

This is a further stage in the compilation of GDC. The + symbol is used to separate alternative guarded process continuations producing a syntax not unlike CSP (Hoare, 1985). The similarity with CSP is not surprising since GDC was, indirectly, inspired by the development of CSP. With a message passing analogy, suspension while waiting for variables to become instantiated can be understood as blocking-receive. Output by unification primitives in the body of a clause can be understood as nonblocking-send. With nonblocking-send, sometimes known as asynchronous communication, a process sends a message and continues processing without waiting for the message to be received. (With blocking-send, a process sending a message suspends until it has been accepted by the recipient.)

The compilation process can be taken a stage further by noting that the test Xs < = [H | Ts] occurs in both clauses. Redundant tests can be eliminated by composing the guard tests into a decision tree:

partition(Xs, Pivot, Ls, Gs) ←
Xs < = [ ] ← Ls = [ ], Gs = [ ]
+ Xs < = [H | Ts],
(H = < Pivot ← Ls = [H | L1s], **partition**(Ts, Pivot, L1s, Gs)
+ Pivot = < H ← Gs = [H | G1s], **partition**(Ts, Pivot, Ls, G1s)).

Indeed, the two constraints H = < Pivot and Pivot = < H can be coalesced at the expense of indeterminism.

3 Dynamically Reconfiguring Pipelined Operation

Delaying variable binding until after committal gives a dynamically reconfiguring pipelined dataflow synchronization operation to recursive GDC programs (as opposed to the textual order control flow synchronization of Prolog). This is illustrated by a goal queue:

← **genRandom**(343, ListNums),
partition(ListNums, 0, Ls, Gs),
consume(Ls),
consume(Gs).

The goal can be represented diagrammatically shown in Figure 1.

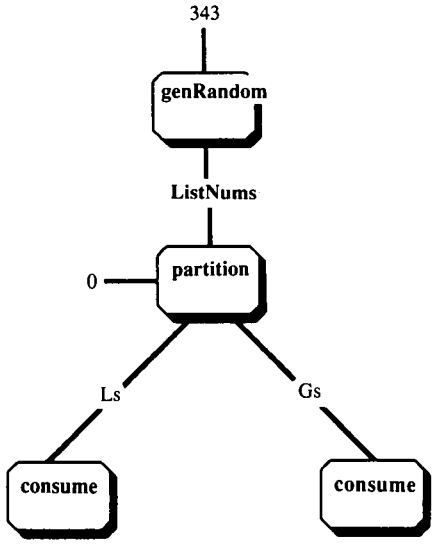

Figure 1 The goal process structure

Here, **genRandom** is a process which generates a list of random integers; the number 343 is the seed of the pseudo-random number generator. **Partition** splits the list into nonpositive and nonnegative integers. Note that the indeterminacy in **partition** means that 0 may go onto either sublist. The two lists are then consumed by two **consume** processes. After one reduction the diagram representing the goal becomes as shown in Figure 2. And after a further reduction we have Figure 3. After several more reductions the picture may be as in Figure 4.

At this stage the conditions of a number of clauses are satisfied simultaneously by the surviving goal literals and they may be committed (fired) in parallel (subject to the availability of processors). The important point about condition synchronization is that the two lists Ls and Gs are constructed and

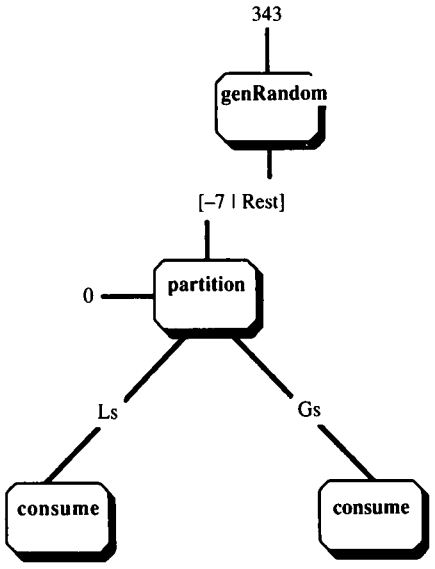

Figure 2 The Goal After One Reduction

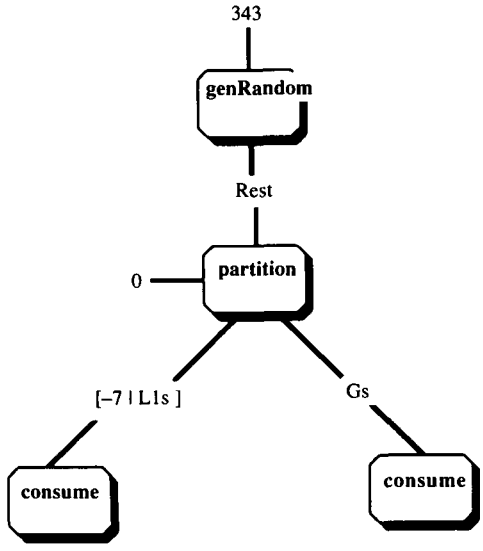

Figure 3 The Goal After Two Reductions

consumed incrementally and concurrently; any number of processes can be active at any time depending on the availability of data and processors. This is in contrast to a corresponding Prolog program which, operationally, would construct the two lists fully and then consume the first and then the second. In GDC the data flows from producer to consumer along dynamically created pipelines: the list structure automatically provides a means of streaming. (More on the graphical representation of GDC is presented in Ringwood (1987b).)

The process reading of a logic program with condition synchronization allows a form of programming not possible in Prolog: that of perpetual processes which are desirable for reactive systems. A Prolog relation for partition without its first clause would be unthinkable, except as a bug, since if this were omitted there would be no base clause to terminate the recursion. Because in GDC processes have the potential to be fairly reduced the behavior of partition without the first clause describes a nonterminating process. **Gen Random** can produce an infinite stream of integers which are partitioned incrementally by partition and consumed incrementally and concurrently by the two consume processes. The pipelines allowed by condition synchronization can be created dynamically in GDC as described in Ringwood (1988a).

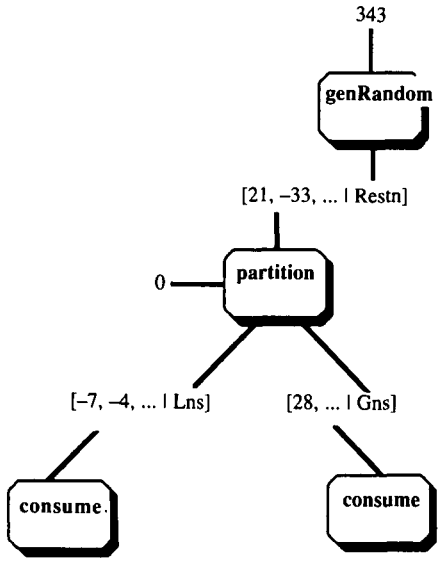

Figure 4 After several more reductions

4 Metainterpretation, Indertiminacy and Completeness

The process computational model of a logic program does not destroy its pattern matching clause invocation mechanism or its affinity for metaprogramming. The simplest form of metainterpreter, the *vanilla interpreter*, is one that emulates the underlying computational model:

```
reduce([ ]) ← true ←
    true.
reduce([literal(Goal) | RestQueue]) ← true ←
    clause(literal(Goal), Body),
    reduce(Body),
    reduce(RestQueue),
```

where a clause of the form **h ← g1, ... gn ← b1, ... bm.** is represented in the metainterpreter by

```
clause (literal(H), T) ← g1, ... gn ←
    T = [literal(b1), ... literal(bm)].
```

Using enhanced metainterpreters (flavours) one can implement a wide spectrum of programming environments and operating system functions such as inspecting and affecting the state of the computation, and detecting distributed termination and deadlock in a simple and uniform way. The dynamic pipeline parallelism and indeterminacy is particularly suited to reactive programming (e.g. Ringwood, 1988a).

The completeness sacrificed by replacing call-head unification in Prolog by condition synchronization manifests itself in the restricted mode of use of program clauses. For example the GDC relation for concatenate

```
concatenate([ ], List, Total) ← true ←
    Total = List.
concatenate([Item | List1], List2, Total) ← true ←
    Total = [Item | Rest], concatenate(List1, List2, Rest)
```

can only "fire" when the first argument of the call is a list. Unlike Prolog, the relation as it stands cannot be used to split a list; that is, with the third argument known but the first and second unknown. With a redefinition, concatenate can be used to indeterminately split a list

```
split(List1, List2, [ ]) ← true ←
    List1 = [ ], List2 = [ ].
split(List1, List2, [Item | Total]) ← true ←
    List1 = [Item | Rest], split(Rest, List2, Total).
split(List1, List2,[Item | Total]) ← true ←
    List1 = [ ], List2 = [Item | Total].
```

We now have a situation where two logically equivalent representations of the same formulae behave totally differently in operation. (But this is not unfamiliar to Prolog programmers because of the control flow synchronization, the ever present bottomless pits in depth-first search and the sometimes inevitable use of cut.) The patterns in the head of the clause in GDC are thus being used for metalevel control, restricting the mode of use of relations; in this way (directed relations), GDC is somewhat closer to a functional language than is Prolog.

While a specific mode of use is implied by individual clauses it is not true of relations as a whole. The ability to have multimode relations is useful as the following arithmetic relation for **plus** illustrates. It has the three expected modes of use, one mode for each clause.

```
plus(X, Y, Z) ← integer(X), integer(Y) ←
    Z is X + Y.
plus(X, Y, Z) ← integer(X), integer(Z) ←
    Y is Z − X.
```

```
plus (X, Y, Z) ← integer(Y), integer(Z) ←
    X is Z − Y.
```

The primitive **integer** is similar to its Edinburgh Prolog counterpart except that the clause is not a candidate for commitment until its argument, provided by the caller, is instantiated. The primitive **is** is the same as its Edinburgh counterpart: along with the syntactic identity predicate, "=", it is privileged in being able to perform output (binding of non-local variables).

Despite the committed choice, the much criticized exhaustive search of Prolog (Ringwood, 1988b) has not been totally eradicated from GDC. A program which will determine if a given key is present at some node of a binary tree is as follows:

```
onTree(Key, tree(Left, Key1, _), Found) ←
    Key =/= Key1, onTree(Key, Left, Found) ←
    true.
onTree(Key, tree(_, Key, _), Found) ← true ←
    Found = found.
onTree(Key, tree(_, Key1, Right), Found) ←
    Key =/= Key1, onTree(Key, Right, Found) ←
    true.
```

This example illustrates that the premature binding problem is more deep rooted than has, hitherto, been suggested.

5 The Premature Binding Problem Revisited

In GDC there is little restriction on the kind of literals that may appear in a guard (no output primitives which involve clause head variables are allowed), so a guard process may recursively spawn other AND-parallel processes at will. As the new processes may invoke new guards, this can lead to a system of arbitrarily nested guards as will be seen on closer examination of the behaviour of the **onTree** example. Output primitives which involve variables which occur in the clause head are deliberately excluded from the guard because they can cause premature binding of variables before the clause is committed and this in turn can cause unwarranted premature failure of the query if some other clause is chosen. As will be explained, arbitrary guard conditions can produce failure for the same reason.

Condition synchronization in GDC is intended to control the eagerness of unification (Ringwood, 1988a), but it does not prevent the possibility of eager nested guards giving rise to bindings; call variables could be prematurely bound by some guards before the top-level clause is committed and this can lead to failure of the goal when a solution is possible. Suppose the initial query (hypothesis or denial) ← Q (where the unary implication symbol denotes logical negation)

$$\leftarrow \mathbf{a}_1, \mathbf{a}_2, \ldots \mathbf{a}_j, \ldots \mathbf{q}_n. \qquad (\leftarrow Q)$$

is given and, where for the sake of exposition, the arguments of literals have been suppressed. Note that the initial query should not be considered as a list of atoms, no ordering is implied, but rather it is a set. Let β^1 be the most general unifier of some $\mathbf{a}_j$ in the query and the head $\mathbf{h}$ of some clause

$$\mathbf{h} \leftarrow \mathbf{g}_1, \mathbf{g}_2, \ldots \leftarrow \mathbf{b}_1, \mathbf{b}_2, \ldots \mathbf{b}_m. \qquad (C)$$

in the program that is, $\mathbf{a}_j\beta^1 = \mathbf{h}\beta^1$ (syntactic equality modulo the renaming of variables). The substitution is an input pattern match if $\mathbf{a}_j = \mathbf{h}\beta^1$, where the substitution is only applied to $\mathbf{h}$. If β^1 is an input match and if the guard condition $(\leftarrow \mathbf{g}_1, \mathbf{g}_2, \ldots)\beta$ is successful with some substitution of variables β which is an extension of β^1 (i.e. $\beta = \beta^1\beta^2$, juxtaposition denoting composition of substitutions) then the clause is a candidate for committal. If this clause is the one selected, the call $\mathbf{a}_j$

is replaced by the body $\mathbf{b}_1, \mathbf{b}_2, \ldots \mathbf{b}_m$ so that the resolvement of the initial goal and the input clause reduces to

$$\leftarrow (\mathbf{a}_1, \mathbf{a}_2, \ldots (\mathbf{b}_1, \mathbf{b}_2, \ldots \mathbf{b}_m), \ldots \mathbf{a}_n)\beta.$$

(Reduction sometimes seems to be an inappropriate terminology for an action which may increase the complexity.) If any of the bindings entailed by β are due to recursive guards of clause C they could well be communicated to other $\mathbf{a}_i$ ($i \neq j$), which share variables with $\mathbf{a}_j$, before committal to clause C.

$$\leftarrow (\mathbf{a}_1, \mathbf{a}_2, \ldots \mathbf{a}_n)\beta \qquad (\leftarrow Q')$$

If by some chance the clause C is not committed but some other is, the communicated bindings will, in general, not relate to the chosen clause. It is then likely that the query will fail because the variable bindings are more specific than is warranted. There is no need to worry about the substitution β^1 since this is limited to pattern matching and only affects the local variables of the clause; we are left with the bindings caused by the guard conditions:

$$\leftarrow (\mathbf{a}_1, \mathbf{a}_2, \ldots \mathbf{a}_j, \ldots \mathbf{a}_n)\beta^2 \qquad (\leftarrow Q'')$$

There will be no problem if the bindings generated by the guard β^2 do not cause bindings of the variables in the call, the global variables

$$\mathbf{a}_i\beta = \mathbf{a}_i\beta^2 = \mathbf{a}_i.$$

The possible inequality of $\leftarrow Q'$, $\leftarrow Q''$ and $\leftarrow Q$ is a formalization of *the premature binding problem.* The equality of $\leftarrow Q'$ and $\leftarrow Q''$ is guaranteed by pattern matching and the equality of $\leftarrow Q''$ and $\leftarrow Q$ can be thought of as the natural extension of pattern matching for guards with nonempty conditions. This gives rise to the concept of *guard safety*: a guard is said to be safe if its evaluation does not incur any substition of nonlocal variables. (The formalism presented above can be extended to give semantics to perpetual processes in terms of conditional answers (Ringwood, 1987a). With the conditional answer semantics the language is most suited to an Open-World interpretation rather than the Closed-World assumption allied with Prolog.)

Safe guards can only test the input parameters since they produce no output bindings, for example, the membership relation defined by

```
member(X, [X | Y]) ← true ←
    true.
member(X, [Y | Ys]) ← X =\= Y ←
    member(X, Ys).
```

is clearly safe as can be easily recognized syntactically because there are no equality predicates in the body of any of its clauses or the bodies of any of the clauses of its subgoals. (Note that the recursive call to **member** in the second clause could equally well be part of the guard with the same declarative and operational effect. This gives another way of recognizing a safe guard: the body goals can *safely* be transferred to the guard.)

By comparison the **OnTree** relation defined previously

```
onTree(Key, tree(Left, Key1, _), Found) ←
    Key =/= Key1, onTree(Key, Left, Found) ←
    true.
onTree(Key, tree(_, Key , _), Found) ← true ←
    Found = found.
onTree (Key, tree(_, Key1, Right), Found) ←
    Key=/=Key1, onTree(Key, Right, Found) ←
    true.
```

is unsafe, as the variable *Found* in a top level call can be bound to *found* by a recursively invoked guard call before committal of the initial invocation. In this example, this does not cause a problem

because there is no conflict as to what the variable *Found* can be bound to; it can only be bound to the constant *found* and nothing else; furthermore, it will only be bound by a recursive guard call if the *Key* is on the tree so the conclusion will be sound. This example illustrates that the property of safety is not vital for soundness of the conclusion but because of indeterminism the unrecognized presence of this problem can lead to unnecessary failure when greater care with the programming can circumvent the problem (Ringwood, 1987a).

A simple way to make the **onTree** program safe is by making local copies (copies in the guard) of all endangered variables.

```
onTree(Key, tree(Left, Key1, _), Found) ←
   Key =/= Key1, onTree(Key, Left, FoundL) ←
   FoundL = Found.
onTree(Key, tree(_, Key , _), Found ← true ←
   Found = found.
onTree(Key, tree(_, Key1, Right), Found) ←
   Key=/=Key1, onTree(Key, Right, FoundR) ←
   FoundR = Found.
```

The local copy is only unified with the global parent after committal.

In the above computational model the guard describes a complex inference step of a nature similar to paramodulation or typed inference. Regarding the guard as a self contained unit prompts the following restriction of the language which can guarantee safety. With unrestricted guards a GDC program gives rise to a hierarchical computation. The computation is organised as an AND/OR process tree, the depth of the OR-tree corresponding to exhaustive search in the nesting of guards. The flat restriction of the language, FGDC (Flat GDC), only allows primitive (and consequently nonrecursive) predicates in the guard which only test the input parameters and perform no output. The primitives include syntactic equality-inequality testing, comparisons, and type checking none of which can perform any output. In the flat restriction of the language there are no recursive guard calls, the computation degenerates to an AND-tree and in this way flat programs are assured of being safe. Hence the terminology *flat* (note, however, the computation is still a tree, but only an AND tree).

The restriction to a flat subset of the language is not debilitating but requires a shift of programming style further away from Prolog. The FGDC programmer cannot fall back on the exhaustive search of implicit failure driven loops or OR-parallelism; search has to be programmed. However, many programs one writes are naturally flat; the vanilla meta interpreter for example remains unchanged. Those programs which are not naturally flat and require search can be rewritten using suitably chosen recursive data structures. A flat version of the tree search program in FGDC is as follows:

```
onTree(Key, tree(Left, Key1, Right), Found) ← Key =/= Key1 ←
   onTree(Key, Left, FoundL),
   onEither(FoundL, FoundR, Found),
   onTree(Key, Right, FoundR).
onTree (Key, tree(_, Key, _), Found) ← true ←
   Found = found.
onTree(_, emptyTree, Found) ← true ←
   Found = notFound.

onEither(found, _, Found) ← true ←
   Found = found.
onEither(_, found, Found) ← true ←
   Found = found.
onEither(notFound, notFound, Found) ← true ←
   Found = notFound.
```

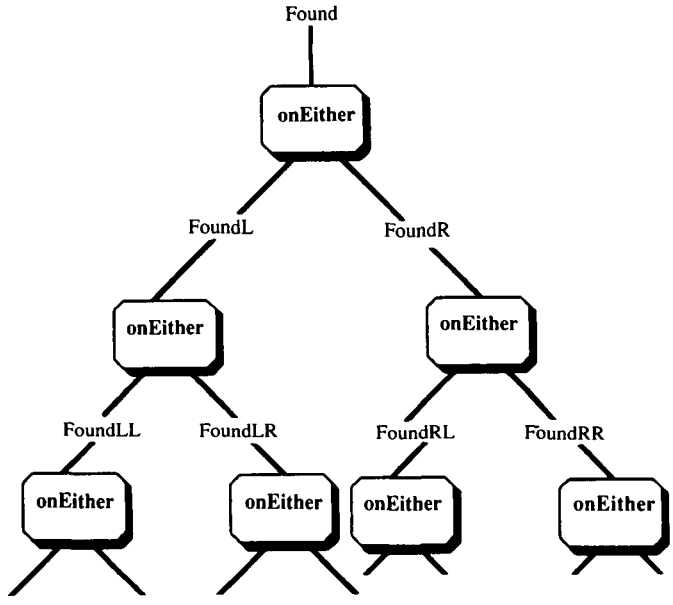

Figure 5 The AND Process Tree of **onEither**

A hand trace of a call to the process **onTree** reveals that the data tree in the call is mimicked by a dynamically constructed tree of **onEither** processes and *found* or *notFound* eventually percolates to the top of the tree (see Figure 5).

This example is indicative of the nature of programming in concurrent logic languages as opposed to Prolog. Whereas stacks (nests) of subgoals (subordinated procedures) are generated in Prolog, trees of data sharing processes dominate the evaluation of concurrent logic languages. But this is all to the good, since the original desire was to produce a new computational model for logic programming with a high potential for parallel evaluation.

The flat restriction of GDC has a very simple computational model where the state of a computation is represented by a process pool which has no hierarchical dependency (guards). Processes in the pool are then reducible if they satisfy the guard condition of some clause or they may suspend waiting for some variables to be bound by other processes. A process can terminate by reduction with a unit clause (a clause with no body). In the flat variant of the language the primitive guard conditions can be recognized as constraints on the use of the clause for reducing the call.

The flat version appears to be more of a metalevel description of the search problem than the non-flat version as it doesn't depend on the failure of recursive guards to effect the search. This program embodies the desire of converting OR-parallel search to AND parallelism (Ringwood, 1988b). However, a computationally explosive OR-search has been replaced by an equally explosive AND-search and so does not overcome the strongly raised objections to exhaustive search. But, because there is now cooperation among processes there is an opportunity for more intelligent searching (Huntbach, 1988). To conclude, the premature binding problem does not cause unsoundness in the theorem proving sense, as has been claimed by those with only a superficial understanding of the problem. It only has the danger of unnecessarily overspecifying (instantiating) goal variables leading to premature failure when a solution is possible. Because a program is unsafe it does not mean that premature binding will necessarily be a problem as illustrated by the unsafe **onTree** program; this example illustrates how it is possible for a program to be unsafe and yet not lead to unwarranted failure or an unsound conclusion.

6 The Binding Conflict Problem

The premature binding problem is one of synchronization between the binding of variables and committal to a particular clause. There is another synchronization problem which relates to

cooperative processing. There may be some conflict between the intentions of two concurrent goals because of inadequate coordination. This is illustrated by the next seemingly innocuous example.

p(X) ← **true** ←
  X = a.
p(X) ← **true** ←
  X = b.

If each primitive component of the body of a clause is, for the sake of argument, assumed to be an atomic action (this may be too much to ask of unification) then simulating parallelism by interleaving, a possible evaluation scenario for what seems a rather perverse initial query

← **p**(X), **p**(X).

is:

← **p**(X), X = a.
← X = b, X = a.
← a = b.

Thus, even though there are two solutions to the initial query, X = a and X = b, the goal fails: one conjunct has made one choice for the solution and the other conjunct another incompatible choice. That is, the multiple producers of the variable X are in conflict and this leads to failure of the goal when an amicable solution is achievable. Like the premature binding problem this inconvenience does not cause any unsoundness in a conclusion; it just increases the possibility of a solution not being found when there is one. In a mild attack, it may cause overspecification (instantiation) of the variable binding and at worst it will cause failure as in the above example. Rather than a synchronization problem this is more properly a problem of coordination.

The problem here is that one process has irrevocably committed to one solution before obtaining agreement with its cooperating partners (any other processes which share the same variable and can make assignments to it); this is a problem of multiple producers of bindings for the same variable. In the present example the problem can simply be circumvented by programming it away: renaming one conjunct in the initial goal ← **p**(X), **c**(X) and redesignating it a consumer.

p(X) ← **true** ←
  X = a.
p(X) ← **true** ←
  X = b.
c(a) ← **true** ←
  true.
c(b) ← **true** ←
  true.

In this way there is only one producer; generally, there may be as many consumers as one likes. This example is trivial to solve by this device but, still, it is indicative of more complex examples where such a simple solution is not possible, such as the **onTree** example above. In such situations another programming technique indeterminate merging (e.g. Burt and Ringwood, 1988) can be adopted.

← **p**(Y), **p**(Z), **either**(Y, Z, X).

either(Y, Z, X) ← **nonvar**(Y) ← X = Y.
either(Y, Z, X) ← **nonvar**(Z) ← X = Z.

This is the method used in programming the **onTree** example in FGDC above. The technique can be understood as eliminating multiple producers by introducing consumer producer pairs, the arbitration predicate, **either**. Another technique, *synchronization variables*, can also be employed (e.g. Burt and Ringwood, 1988). A valid criticism of these *programming techniques* is that they divert the

declarative ideal and obscure the intention of the program: synchronization variables have little or no declarative meaning. This is a general complaint: effective computation by nature tends to impose directionality on variables leading to algorithmic prescriptions. When effective computation is not of the essence one can afford not to assign modes to variables.

An alternative to programming the coordination using such programming techniques is to modify the language and introduce another synchronization mechanism. Clearly we need to introduce a test in the guard to see if the proposed unification is acceptable to all cooperating partners.

$$\mathbf{p}(X) \leftarrow X = a \leftarrow X = a.$$
$$\mathbf{p}(X) \leftarrow X = b \leftarrow X = b.$$

The clauses are almost as they were before but with the addition of unifiability conditions pairing output primitives in the body. The equality primitive in the guard is only used to test for unifiability and may not cause any variable bindings. Only those clauses which can perform the output are candidates for committal. Testing for unifiability before committing is not enough, as consideration of the above scenario will confirm; the passive and active pair of unifiability tests in the guard and the unification output in the body must be inseparable, an atomic test, commit and unify. The concept of atomic test and unify is much more complex than the familiar atomic test and set introduced into more conventional concurrent languages for much the same reason.

To clarify matters, an alternative approach to atomicity is as follows: for a clause to be a candidate for committal the guard must not only ascertain that the bindings prescribed by the unfiability primitive are possible but also obtain exclusive access to those variables which would be bound if the clause were selected and unification was to be performed. A clause will not be a candidate for committal until the exclusive access rights have been obtained. Access to variables obtained by the equality predicate in the guard endows exclusive binding privilege to the corresponding equality predicate to the body of the clause. In this way no output binding is committed to if it cannot be achieved. Exclusive access to a large number of variables involved in a unification, some of which will only be determined at runtime, is computationally expensive to implement; this is particularly true of a distributed implementation. So, when there is no possibility of confict of bindings this form of guarded output is not to be encouraged. Rather than being the only form of output, the above atomic unification could be considered as an additional output mechanism, to be used rarely. Eventual unification, equality primitives in the body of a clause without a counterpart in the guard, will be treated as before, no variable access privilege is required before committal. The extension of GDC just described will be called Atomic Guarded Definite Clauses (AGDC) so that it can be referred to later. In the analogy of message passing, the additional mechanism can be understood as a blocking-send. Thus, AGDC offers a blocking-send in addition to the blocking-receive and nonblocking-send of GDC.

The problematic scenario depicted above cannot occur in AGDC. There are only two scenarios:

$\leftarrow \mathbf{p}(X), \mathbf{p}(X).$		$\leftarrow \mathbf{p}(X), \mathbf{p}(X).$
$\leftarrow \mathbf{p}(X?), X = a.$	or	$\leftarrow \mathbf{p}(X?), X = b.$
$\leftarrow \mathbf{p}(a).$		$\leftarrow \mathbf{p}(b).$
$\leftarrow a = a.$		$\leftarrow b = b.$
$\leftarrow$		$\leftarrow$

That exclusive access to a variable has been obtained is indicated by a question mark suffix on all occurrences of the variable other than the privileged equality predicate. (The question mark is not a syntactic feature of the language but is used in evaluation scenarios as a syntactic device to illustrate exclusive write access.) Variables with a question mark suffix can be thought of as read-only occurrences; only unannotated occurrences can be bound, the exclusive access ensuring that there is only one such write-enabled occurrence in a query.

The syntax of the equality literal pairs in the guard and body of a clause seems to suggest that they

are independent which, as described, is not the case. A syntactic improvement can be made by coalescing the body occurrence of the equality predicate with the guard occurrence

p(X) ← **true**; X = a ← **true**.
p(X) ← **true**; X = b ← **true**.

with the intention that unification takes place atomically at committal (or equivalently after committal when exclusive binding rights have been obtained). The semicolon is used to sequence pure test conditions from test and output conditions; the additional **true** goals indicate that there are no pure test conditions in these particular clauses. Although not entirely satisfactory, this syntax does to some extent convey the operational behaviour that test unification only takes place after pure input tests and output binding takes place only after it has been established that atomic unification will succeed. With this syntax, guard conditions now come in two forms, blocking-receive, just those that were allowed in GDC for test purposes, and blocking-send, the privileged output primitives involving clause head variables previously only allowed in the body of GDC clauses.

Guarded output as contemplated in AGDC is notoriously difficult to implement. The concurrent programming language CSP (Hoare, 1985), for example, uses guarded output but its practical realization, Occam (Inmos, 1984), does not. Atomic binding of variables or, what amounts to the same thing, obtaining exclusive write access to variables, requires coordination amongst all processes which share those variables. The problem is further complicated since in general the variable inculcated by the unification may only be known at runtime. The difficulties of obtaining exclusive simultaneous access to resources is well illustrated by the Dining Philosphers problem (e.g. Ringwood, 1988a). Replacing guarded unification by atomic assignment (specified variable binding) as suggested by Burt and Ringwood (1988) is a much less expensive alternative because it can be compiled whereas unification cannot be fully compiled. With atomic assignment the previous example becomes:

p(a) ← **true**; **true** ← **true**.
p(X) ← **true**; X := a ← **true**.
p(b) ← **true**; **true** ← **true**.
p(X) ← **true**; X := b ← **true**.

For a clause to be a candidate for committal any variable on the left hand side of any assignment statement, for example X := a, in the guard must be unbound at the time of call and exclusive access must be obtained on this variable before the clause is a candidate for committal. The variable is bound as part of the committal process. The possibility that a variable may be bound at the time of call can be catered for by other clauses which specify the anticipated bindings. Note that in homogeneous syntax the two clauses

p(X) ←
  X < = a; **true** ← **true**
  + **true**; X := a ← **true**,

can be regarded as a compilation of

p(X) ← **true**; X = a ← **true**.

The fewer the number of variables specified in the guarded atomic assignments the less expensive it is to achieve coordination (exclusive access to all of them simultaneously) so ideally at most one atomic assignment should appear in the guard. In this case the assignment primitive in the guard is equivalent to an atomic var-test-and-assign primitive more familiar as atomic test and set introduced into concurrent programming languages which allow multiple assignment.

Note that AGDC does not cure the premature binding problem but the flat restriction, abbreviated FAGDC deals with both problems.

7 The Other Concurrent Logic Languages

The computational models of logic programming languages can be classified as subroutine (stack) based or process (queue) based. These two forms give rise to Prolog (and variations such as coroutining and OR-parallel Prologs) and the concurrent logic languages, of which GDC and AGDC are representative. In addition to the process computational model, the distinguishing features of concurrent logic languages are guarded-inference indeterminacy (as opposed to Prolog's backtraking nondeterminacy) and dataflow (more properly condition) synchronization (as opposed to Prolog's controlflow synchronization). The two examples of concurrent logic languages introduced above, GDC and AGDC, are intended to be indicative and there have been many variations. What follows attempts to distinguish the main trends without going into unilluminating detail or side issues. The original syntaxes of the different languages will not be adhered to; rather a common syntax is adopted to better appreciate the similarities. The form of exposition is not the historical one, as indicated by the inheritance tree in Ringwood (1988a), but, rather, is chosen for pedagogical reasons. Alternative ways in which the two problems of premature binding and binding conflict can be handled serve to distinguish the different languages of the concurrent logic programming family. These differences, generally, manifest themselves as compiletime or runtime specified synchronization mechanisms. With GDC the preference is for compiletime specification of synchronization and user programmed circumvention.

The languages differ by the synchronization mechanisms they adopt to mitigate the two principle synchronization difficulties. The first, the premature binding problem is a synchronization between binding goal variables and clause commitment. The second, the binding conflict problem is the synchronization between concurrent goals competing to bind shared variables. Two classes of concurrent logic languages can be distinguished: in the largest class, the synchronization is mainly concerned with mitigating premature binding; this class includes GDC and the Parlog family (Clark and Gregory, 1986; Ringwood, 1988a) and the GHC family (Ueda, 1986; Fuchi and Furukawa, 1986). In the other class the synchronization mechanism is principally concerned to mitigate binding conflict; this is the Concurrent Prolog family of languages (Shapiro, 1986). Somewhat bridging the gap between the two classes is the CP family (Saraswat, 1987a). (The name CP for Saraswat's family was an unfortunate choice because these are the initial letters of Concurrent Prolog. This confusion is further compounded because the Flat subset of Concurrent Prolog is abbreviated FCP. For this reason the flat restriction of CP will be referred to as Flat CP.)

Guarded Horn Clauses

Guarded Horn Clauses GHC is very close to GDC is using no explicit syntactic symbols to specify the synchronization, apart of course from term patterns in the head of a clause and guard condition literals. In GDC the reduction of a process by a relation is delayed until the process pattern matches the head of some clause for which the input data satisfies the guard constraints. GHC does not talk about pattern-matching, but Prolong-like unification of terms. However, in GHC two forms of unification are prevented from occurring at runtime:

1 Unification invoked directly or indirectly in the guard of a clause C by a goal $\leftarrow \mathbf{a}$ (unification of **a** with the head of C and any unification invoked by reducing the guard conditions of C) cannot instantiate variables in the call $\leftarrow \mathbf{a}$.
2 Unification invoked directly or indirectly in the body of a clause cannot instantiate variables in the guard of a clause C or the goal **a** until C has been selected for commitment.

A goal reduction, in the Prolog sense, that is possible only by causing such instantiations is suspended.

A notable difference between GHC and GDC is that body goals can be invoked before clause committal as is suggested by the second suspension rule. Suppose for the moment that this is not the case and that like GDC the body of a GHC clause is not invoked until it is committed. The first

suspension rule of GHC is a complete solution to the premature binding problem, while the GDC mechanism gives only a partial control over the problem. The GHC mechanism is a runtime safety check on programs, unsafe bindings are operationally prevented from occurring, whereas in GDC there is little impediment to running unsafe programs. But as the unsafe **onTree** example illustrates, it is fallacious to believe that just because the guards of a program are unsafe they will produce an unsound conclusion. No unsound solution is ever produced by an unsafe guard (Ringwood, 1987a). The danger is that premature binding will lead to an overspecified solution or, in the worst case, unnecessary failure to draw a conclusion. The runtime safety check of GHC gives a greater tendency for programs to deadlock; this is again illustrated by the unsafe **onTree** program which will deadlock in GHC yet GDC will return a sound conclusion.

The greater tendency of GHC programs to deadlock is further compounded by a prfusion of suspended processes because, unlike GDC, body goals can be spawned before a clause has been committed. An example of this is the counting program:

```
p(0) ← true ←
   true.
p(N) ← N > 0 ←
   N1 is N − 1, p(N1).
```

With a call ← **p**(0) in GHC, the second clause can spawn a body goal **p**(N1) before performing the guard test which in turn can spawn a body goal and so on, even though the first clause is the only candidate. Clearly, there can be much ineffectual computation of body goals in clauses which are not chosen for committal and a complex process control structure is needed to kill off unwanted, initiated and, more often than not, suspended body goals.

Parlog84 and Parlog86

Another way of tackling the premature binding problem, emerges in Parlog84 (Clark and Gregory, 1986). In this predecessor of Parlog86 (Ringwood, 1988a) (the two year discrepancy between names and publication dates indicates the lethargy of the editorial process, KER excluded) and GDC a mode declaration for each relation specifies whether an argument position is used for input or output. For example, in Parlog84 the partition relation becomes:

```
mode partition(List?, Pivot?, Lesser!, Greater!)
partition([ ], Pivot, [ ], [ ]) ← true ← true.
partition([H | Ts], Pivot, [H | L1s], Gs) ← H = < Pivot ←
   partition(Ts, Pivot, L1s, Gs).
partition([H | Ts], Pivot, Ls, [H | G1s]) ← Pivot = < H ←
   partition(Ts, Pivot, Ls, G1s).
```

Input argument positions, denoted by a question mark in the mode declaration, are used for pattern matching in the same way as in GDC. Output argument positions, denoted by an exclamation mark in the mode declaration, are used to denote unification to call variables which takes place only after a clause is committed. (In fact, in Parlog84 the equality predicate was not a primitive only eventual assignment primitive ":=" was available.) Mode declarations have the disadvantage that one has to refer back to the mode to determine what takes place before and after committal but more importantly mode declarations are not a suitable syntax for metainterpretation. Imagine a metainterpreter for Parlog84 corresponding to the one given above for GDC. The ability to write simple metainterpreters is sacrificed by such mode declarations. The mode declarations also suggest that a relation can only be used in one mode but this is not the case as the Parlog84 definition of the **plus** relation shows:

```
mode plus(?,?,?)
plus( X,Y, Z) ← integer(X), integer(Y) ←
   Z is X + Y.
```

```
plus(X, Y, Z) ← integer(X), integer(Z) ←
   Y is Z − X.
plus(X, Y, Z) ← integer(Y), integer(Z) ←
   X is Z − Y.
```

This completely belies the mode declarations and illustrates how irrational they really are. Mode declarations are a residue of predecessor languages, Parlog83 and the Relational languages (Ringwood, 1988a) in which the modal use of clauses was more strictly controlled.

The Parlog84 program for the above **onTree** relation looks as one might expect

```
mode onTree(Key?, Tree?, Found!).
onTree(Key, tree(Left, Key1, _), Found) ←
   Key =/= Key1, onTree(Key, Left, Found) ←
   true.
onTree(Key, tree(_, Key , _), found) ← true ←
   true.
onTree(Key, tree(_, Key1, Right), Found) ←
   Key=/=Key1, onTree(Key, Right, Found) ←
   true.
```

but variable copying is implicit in the output mode declaration and the direct equivalent is the safe GDC **onTree** relation:

```
onTree(Key, tree(Left, Key1, _), Found) ←
   Key =/= Key1, onTree(Key, Left, FoundL) ←
   FoundL = Found.
onTree(Key, tree(_, Key , _), Found) ← true ←
   Found = found.
onTree(Key, tree(_, Key1, Right), Found) ←
   Key=/=Key1, onTree(Key, Right, FoundR) ←
   FoundR = Found.
```

By the duplication of potentially assignable variables the guards of alternative clauses maintain separate local environments for variables, only unifying with global variables on commitment. Because the variable copying is implicit in the mode declarations it is much more difficult to recognize safe programs by inspection with Parlog84 than GDC.

In addition to the above Parlog84 differs from GDC by allowing sequential control flow operators, sequential conjunction and sequential clause trial. Parlog86 (Ringwood, 1988a) differs from GDC by exactly these sequential operators, and so is transitional between Parlog84 and GDC. With hindsight, sequential operators are something of a mistake because mixing condition synchronization and control flow synchronization tends to destroy the symmetry and semantics of the languages and leads to unnecessary complications for implementation and increased problems of deadlock. These additional control synchronization mechanisms could be imported into any of the concurrent logic language if they were ever considered necessary or desirable. (Concurrent Prolog has a somewhat restricted clause of sequencing or default operator called *otherwise* which is, perhaps, aesthetically preferable to Parlog84's sequential clause separator.)

CP

In Parlog84 the variables which are expected to be bound by a call (those in the output call argument positions) are indicated at compiletime by the mode declarations and the compiletime generated environment ensures safety. However, there is nothing to prevent other variables (those in input argument positions of the call) being bound prematurely when clauses are used in modes which are not accurately reflected by the mode declaration, as for example the **plus** relation. The CP family of languages takes the idea of variable copying a stage further and makes it a runtime rather

than compiletime feature; all variables in the goal are duplicated. Recall that Parlog84 only makes copies of those variables specified to be used for output at compiletime. Some call variables, will only be known at runtime; an example, is a goal ← **p**(f(X)) with a clause head **p**(Y) ←. It is variables such as X that CP additionally makes copies of. There is obviously some overhead in choosing to do this at runtime as opposed to compiletime.With, multiple runtime environments there is no premature binding problem and so there is no need to sequence all input before output in unification as is done in GDC and GHC. In CP, goal and clause held unification is the default clause invocation mechanism, as in Prolog.

CP is like GHC in that it chooses to prevent premature binding by a runtime mechanism: runtime multiple clause environments in the case of CP, runtime suspension rules in the case of GHC. In a rough approximation, the GHC synchronization rule can be interpreted as follows: if you need a new environment, suspend. This suggests a lazy way in which multiple environments can be constructed for CP: the GHC unification suspension rules become rules for forming a new environment for call variables which are threatened with being prematurely bound in unification.

In CP the local environment bindings are unified with global originals as part of the committal process. If a clause were committed to and then its local bindings made public as separate actions there is the possibility of a binding conflict problem as described for GDC. The CP family of languages make compatibility of the local bindings with the global environment part of the clause candidacy test so that the clause is not committed if the local bindings cannot unify with the environment. The unification of local and global environments is treated as part of the atomic action of commital. This is analogous to the guarded atomic unification introduced with AGDC.

The CP family of languages in fact contains nondeterminate, all solution, members but these obscure the present discussion so attention will only be paid to the (at most) single solution subfamily. In this subfamily, in which, like GDC, each clause has a commit operator, the only synchronization mechanism so far described is guarded atomic unification which prevents the binding conflict problem. However, when only one clause is committed, unification as a mechanism for clause invocation gives only limited control over which clause is chosen. The pattern matching in GDC programs and the suspensions rules of GHC gives greater restriction on the choice of which clause can be committed: in both these languages the goal must present the pattern in the head of the clause before it can be considered for committal. In order to better direct the solution, CP takes the dataflow synchronization of argument positions, as specified in the mode declaration in Parlog84, to a lower level of detail. Annotations denoting pattern matching, are made directly on subterms in the head of a clause. The CP relation corresponding to **partition** is

partition([ ]?, Pivot, [ ], [ ]) ← **true** ←
  true.
partition([H | Ts]?, Pivot, [H | L1s], Gs) ← H = < Pivot ←
  partition(Ts, Pivot, L1s, Gs).
partition([H | Ts]?, Pivot, Ls, [H | G1s]) ← Pivot = < H ←
  partition(Ts, Pivot, Ls, G1s).

A question mark specifies that pattern matching is to take place at the corresponding position in the call argument; that is, the clause is not a candidate until the corresponding position in the call argument term is nonvariable. (The actual syntax of CP uses a down arrow or sometimes exclamation mark for the annotation, but a question mark is used here in order to strive for consistency.) The annotation, thus, only applies at the level of function symbol (or constant) and so is a finer level of dataflow synchronization than is specified by the argument pattern in GDC and GHC. Unlike Parlog84, this form of specification encourages the use of different modes for different clauses within the same CP relation without compromising its mode declarations.

Like AGDC, CP has two synchronization mechanisms, atomic unification and pattern matching on annotated terms. However, unlike AGDC the pattern matching has little to do with the premature binding problem. Unlike GHC and GDC in CP unannotated terms (and subterms) take part in full unification (bidirectional pattern matching), not just (unidirectional) pattern matching.

This does not cause any premature binding problems because, as described, all call variables are duplicated in a local environment and the binding caused by head unification only affects the local copies, not the originals. The purpose of the pattern matching synchronization in CP, is then not designed to control the premature binding problem or the binding conflict problem but rather to allow the programmer to better direct which choice will be made at a branch point; that is, to restrict the indeterminism.

The mixing of pattern matching and unification in clause invocation in CP causes an as yet untold of synchronization problem. The unification of a goal ← **p**(X, X) with a clause head **p**(a?, a) suspends if the first argument is pattern matched before the second is unified. However, it does not suspend if the second argument is unified before the first is pattern matched; there is a self feeding of coreferenced variables. This difficulty is a symptom of the fact that the granularity of the specification of synchronization afforded by term annotations is incongruously finer than the granularity of goal reduction. In GDC, pattern matching is sequenced before any unification, a total separation of input testing and output, so this problem does not arise. Another way of looking at the problem is that, in GDC, the granularity of synchronization provided by the pattern matching of the call and the head arguments of a clause is of the same order as the granularity of the process structure as the separate unification primitives in the body of a clause, used for output, suggest.The finer level of synchronization provided in CP is below the granularity of the process structure. The apparently finer level of synchronization afforded by the term annotation of CP can always be achieved in terms of call argument pattern matching.

Concurrent Prolog

The condition synchronization of GDC which limits the premature binding problem serves a double role in that it can, with some programming, be used to limit the binding conflict problem as the producer-consumer example above shows:

← **p**(X), **c**(X).
p(a) ← **true** ← **true**.
p(b) ← **true** ← **true**.
c(X) ← **true** ← X = a.
c(X) ← **true** ← X = b.

but where for perversity **p** is the consumer and **c** the producer. In simple terms some processes are designated as producers and some as consumers of variable bindings; consumers have to wait until the producers have produced a binding before the consumer can continue; that is, asynchronous communication. So, ideally data flows from producers to consumers or, in other words, sources to sinks. However, in GDC, CHC or CP there is no way that clauses can be defined that can make one goal a producer and one goal a consumer in the query.

← **p**(X), **p**(X).

since there is nothing to distinguish the two goals, they refer to the same relation and synchronization is specified at compiletime by the guard of a clause. The atomic unification described for AGDC was illustrated with a scenario in which read-only occurrences of variables were annotated with question marks. Rather than annotating terms in the head of clause like CP Concurrent Prolog makes this variable annotation a syntactic synchronization feature of the language: dataflow synchronization can be specified in Concurrent Prolog by "?" annotations on variables at the language level.

← **p**(X?), **p**(X).

(Unlike the evaluation scenario for AGDC afforded above, however, Concurrent Prolog does not insist that there is only one unannotated occurrence of a variable.)

The Concurrent Prolog goal corresponding to the example which introduced GDC in a previous section

```
← genRandom(343, ListNums),
  partition(0, ListNums?, Ls, Gs),
  consume(Ls?),
  consume(Gs?).
```

clearly indicates the intended data flow. Data flows from unannotated variables to annotated variables, input and output channels. A variable with a read-only (question mark) annotation cannot be instantiated to a nonvariable, any attempt to do so causes the suspension of the process involved. Only a process which has access to the corresponding write enabled variable (no read-only annotation) can instantiate it. The equivalent Concurrent Prolog relation for partition is

```
partition([ ], Pivot, [ ], [ ]) ← true ← true.
partition([H | Ts], Pivot, [H | L1s], Gs) ← H = < Pivot ←
  partition(Ts?, Pivot, L1s, Gs).
partition([H | Ts], Pivot, Ls, [H | G1s]) ← Pivot = < H ←
  partition(Ts?, Pivot, Ls, G1s).
```

A read-only annotated variable in Concurrent Prolog goal roughly corresponds to an input annotated term in the head of a clause in CP; that is, there is a sort of duality here. As with CP, this is a finer grain of dataflow control than GDC or GHC as the annotations can be embedded in variables inside terms.

Like CP, Concurrent Prolog uses runtime multiple clause environments to prevent the premature binding problem and as with CP and GDC guarded output is employed to eliminate the binding conflict problem. The read-only annotation synchronization of Concurrent Prolog like the term annotation of CP is not essential to mitigate the binding conflict or premature binding problems; these are covered by the atomic unification and multiple runtime environments. Like the term annotations of CP, the read-only variable synchronization mechanism is only used to allow the programmer to exert some influence control over the indeterminism.

There is no need to make local copies of read-only variables in a goal, they can be write enabled copies since bindings caused by unification being made only to local copies will be safe. Recall, these are generally only known at runtime. The dataflow synchronization imposed by the read-only variables could then take place before committal when testing if local variable copies can be unified with their global counterpart; only the global counterpart is read-only. Publication of local bindings which propose instantiating global read-only variables would suspend, while alternative candidate clauses which had no such impediment could still commit. In this way synchronization would take place purely at commital in contrast to the other languages where dataflow synchronization takes place at the guard invocation stage. This would seem to be the appropriate place to perform synchronization to give a perfect duality between Concurrent Prolog and CP. However, in some reports of CP Concurrent Prolog, dataflow synchronization to take place at guard invocation like the other languages. Call-head unification attempts to bind read-only variables cause suspension. (This means that, read only variables in the goal need not be duplicated in the local environment.)

In early definitions of Concurrent Prolog it was not clear that atomic unification was being advocated. Furthermore, there was no restriction on the variables which could receive read-only annotations: the annotation could appear on variables in clause heads, guard conditions and clause bodies. The operation of the read-only variable was described by replacing the usual Herbrand unification algorithm by a more general algorithm. These early definitions of read-only unification were order dependent (Saraswat, 1986), a problem from which, as described above, CP itself suffers. Intuitively, the read-only variable cannot be instantiated, it can only take on an instance by virtue of an occurrance of a write enabled counterpart (an occurrence of the same variable without a read-only annotation), in the same scope, becoming instantiated. A process that attempts to instantiate a

T1 \ T2	clause head var	goal var	goal RO var	constant	compound term
clause head var	T2 := ref(T1)				
goal var	T2 := ref(T1)	T2 := T1			
goal RO var	T2 := RO(T1)	suspend on T1	suspend on T1		
constant	T2 := ref(T1)	T2 := T1	suspend on T2	equal constant	
compound term	T2 := ref(T1)	T2 := T1	suspend on T2		equal func unify args

Figure 6 Read-only unification

read-only variable suspends until its write enabled counterpart becomes instantiated. This embodies the implication that a coreferrent goal could feed off itself; for example, the unification of **p**(X, X?) with **p**(a, a); this is the counterpart of the example for CP. After criticism of this ilk the read-only variable in Concurrent Prolog was restricted to guard conditions and clause bodies (it was not allowed in clause heads) bringing out the duality with CP.

Restricting read-only variables to goals of itself does not eliminate the synchronization problem caused by the mixing of unification and read-only variable suspensions since **p**(X, X?) could be the goal and **p**(a, a) the clause head. A reportedly order independent unification algorithm (Taylor, 1988) is said to be encapsulated in Figure 6. (I personally can't see how Figure 6 of itself overcomes the difficulty. Shapiro (1986) has given a description of how this problem can be resolved but this is a global solution involving all the variables which take place in the goal clause-head unification. The table above is of a local nature describing a single step in an extended Herbrand unification algorithm.) A consequence of this extended unification algorithm is that the read-only property of variables can be inherited; write enabled variables (unannotated) can be assigned to be references to read-only occurrences of variables (the matrix element [goal-RO-var, clause-head-var] in Figure 6). As a result, when goals in the initial query become reduced the intended flow of data specified by the original annotations becomes obscured. Since dataflow synchronization is embedded in the variable at runtime, it becomes difficult to predict when and where synchronization will occur without tracing the program. This feature, of the read-only variable, having a dynamic aspect, isolates the Concurrent Prolog family from the others and leads to a less pleasing operational semantics. For GDC, where and under what condition synchronization will take place is specified at compiletime by the guards of clauses.

8 Conclusion

The structure revealed of the different languages can be summarized by the following three dimensional table shown in Figure 7. Figure 7 is a great oversimplification of the issues: for example, the fact that GHC has eager body calls but for GDC body calls are only invoked after committal is not represented in the diagram.

There are two practical ways by which an outsider may discriminate between the subfamilies: one is efficiency and ease of implementation; the other is the powers of expression and the ease with which novices and, indeed, experienced programmers can understand the operational behavior of programs. These are not disconnected but rather go hand-in-hand. The declarative simplicity of Prolog is undoubtedly one of the reasons for its many adherents. But the operational behaviour of

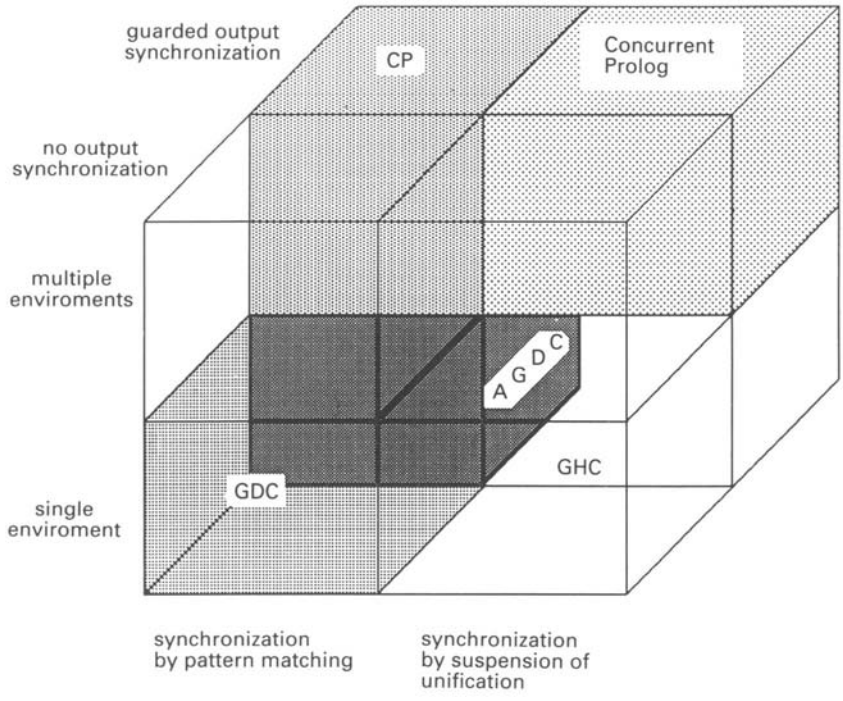

Figure 7 The classification of concurrent logic language

Prolog, with implicit nested loops has been criticized as presenting some difficulty for novices (e.g. Ringwood, 1988b). In this respect the Flat concurrent logic programming languages, with no hierarchical guards, could be said to be preferable to Prolog.

There have been exaggerated claims (e.g. Shapiro, 1988) that FCP is more expressive than FGHC or FGDC and that some programming problems can only be solved in FCP. From the point of view of the class of functions or relations the languages can compute there is nothing to distinguish the languages: they are all Turing complete. It is, however, true that the read-only variable of Concurrent Prolog has strange operational properties which produce behaviour which the other languages would find difficulty imitating (Shapiro, 1988). In this sense Concurrent Prolog is more expressive than the other languages but this is the same sense in which assembler is more expressive than high level languages. A high level language should do its best to discourage bad (nondeclarative) programming practice. Rather than presented as oddities in FCP these tricks are elevated to the status of virtues.

Shapiro (1988) appears, now, to recognise the defects of the read-only variable as a synchronization mechanism and gives his own criticism. Consider the Concurrent Prolog program corresponding to the first GDC relation for **concatenate**.

concatenate([ ], List, Total) ← **true** ←
  Total = List.
concatenate([Item | List1], List2, [Item | Rest]) ← **true** ←
  concatenate(List1?, List2, Rest).

An attractive feature of this program is that, ignoring the read-only annotation, it is identical to its Prolog counterpart. However, the intention of the program, as with its GDC cousin, presented earlier, is that the relation for **concatenate** should be invoked by a goal with its first argument bound to a list. This is ensured for recursive calls by the "?" decoration in the body of the second clause. But, it is not guaranteed by the initial invocation which may come from an unrelated query. If the initial call is ← **concatenate**(List1, List2, Rest) rather than the intended ← **concatenate**(List1?, List2, Rest) the first (or second) clause can be chosen erroneously because without annotations full unification

takes place. As Shapiro now admits placing the responsibility on the caller is not modular and makes proofs of the operational properties of programs difficult. Furthermore, full unification and local environments add runtime overhead while pattern matching can be completely compiled.

Occam's principle has been invoked by Shapiro (1987) to claim that FCP is the simplest language in the concurrent logic programming family. The invincible Doctor, William of Occam, revived the tenets of nominalism in the fourteenth century and Occam's Razor is often cited as a guiding principle on which to base decisions.

> **Occam's Razor** [William of Occam, early 14th c]: A scientific and philosophical principle that entities should not be multiplied unnecessarily.

This principle is interpreted as requiring that the simplest of competing theories be preferred to the more complex or that explanations of unknown phenomena be sought in terms of known quantities. One look at the unification table, Figure 6, for the read-only variable should convince anybody that FCP is not the simplest member of the family. Programs with the read-only synchronization are not modular and only a detailed study of program behaviour can determine when synchronization takes place. It is to be noted that in a recent review Shapiro (1988) chooses to introduce the concept of concurrent logic languages via FCP(1). Shapiro's description of FCP(1) appears to be exactly the same as FGDC. Presumably Shapiro now believes that the read-only variable is too difficult for novices.

Occam's principle can be applied to the number of synchronization mechanisms a language employs. The sequential, control flow synchronization mechanisms advocated by Parlog84 are additional to the condition synchronization mechanism. The principle would claim that a language with only one form of synchronization mechanism is preferable. A multitude of synchronization devices can hide many opportunities for deadlock.

Both CP and Concurrent Prolog have two synchronization mechanisms (as indeed does AGDC): one user specified the other user transparent. If CP is considered without its clause head term annotations or Concurrent Prolog without its read-only variable then they are the same language. This common sublanguage does not suffer from the premature binding problem, because of multiple environments, or the binding conflict problem, because of atomic commit unification. The single synchronization mechanism of atomic commit unification is, however, computationally very expensive. The read-only annotation of Concurrent Prolog can be regarded as an additional programmer specified synchronization device which takes some of the load off atomic commit unification. The additional programmer supplied synchronization mechanism of clause head annotations with CP is to be preferred to read-only variables because it is modular. With the read-only variable it is difficult to predict where and when synchronization will take place and furthermore it is open to abuse with programming tricks. Such programming tricks compromise the (perhaps unachievable) ideal of declarative programming. Shapiro (1988) now admits that the use of the read-only variable results in programs which are more prone to bugs and are less efficient than those which use clause guards to specify the synchronization. Despite acrimonious disputes over intellectual property rights, it appears that the language GDC and Concurrent Prolog are threatening to converge. Shapiro (1988), however, is clearly reluctant to abandon the read-only variable since this was the principle contribution of Concurrent Prolog.

GHC is to be preferred over CP if only for the reason that it has one synchronization mechanism, suspension of unification when binding of goal variables is necessary. It is debatable whether the run time suspension test of GHC is less expensive to implement than the multiple environments of CP; certainly the runtime safety check makes less space demands. (The runtime generated environment of CP is obviously more expensive than the compiletime generated environment of Parlog84.) The granularity of the synchronization provided by term annotations of CP is more complex than the complete separation of input and output provided by GHC and leads to further synchronization problems. The synchronization provided by clause head argument patterns appears to be sufficient level of detail for there to be a close correspondence between GHC programs and corresponding CP programs, as illustrated by the examples.

Saraswat has come to admit there are problems with CP (Saraswat, 1989). Following Maher (1987) he presents a constraint based interpretation of concurrent logic programming languages in which there are no term annotations and pattern matching is sequenced before unification. In this interpretation the **partition** relation becomes

partition(Xs, Pivot, Ls, Gs) ←
  Xs < = [ ]; Ls = [ ], Gs = [ ] ← **true**
+ Xs < = [H | Ts], H = < Pivot; Ls = [H | L1s] ← **partition**(Ts, Pivot, L1s, Gs)
+ Xs < = [H | Ts], Pivot = < H; Gs = [H | G1s] ← **partition**(Ts, Pivot, Ls, G1s).

exactly the same as the homogeneous syntax representation of GDC. (Saraswat uses [ ] in place of + for alternative process behaviours but this would overload its use as the nil list constant.) Saraswat calls the pure input test conditions *Ask constraints* and the guarded output conditions *Tell constraints* following Brachman and Levesque (1984). This redesignated language is dubbed CC by Saraswat for concurrent constraint programming. Constraints are partial finite specifications of (possibly infinite) sets of values for variables, which, as in logic, represent fixed but possibly unknown objects in some domain of discourse. Unification can be recognised as an algorithm for solving constraints on the Herbrand term algebra (Lassez et al., 1988). Restricting the domain on which the constraints are expressed to the Herbrand base gives essentially AGDC (with atomic unification). However, the constraint interpretation of concurrent logic languages opens up the possibility of using domains other than the Herbrand base; for example, taking a domain of arrays could perhaps add something to logic programming which it has been criticized for lacking.

GDC is to be preferred over GHC because pattern matching is easier to understand than the suspension rules of GHC and the runtime safety check of GHC is expensive to implement whereas pattern matching is simple to compile. Violation of safety in Parlog86 is not necessarily fatal as had been previously exaggerated. (Note that FGHC can probably be as easily compiled as FGDC.) For GDC a safe program is not mandatory and in fact unsafe programs can behave perfectly well as illustrated by the unsafe versikon of the **onTree** example. Although there is a danger of unnecessary failure with unsafe Parlog86 programs, pattern matching is to be preferred to the runtime safety test of GHC because there is less tendency to deadlock, and for efficiency pattern matching can be compiled whereas the safety test can only be provided with compiletime assistance. A compiletime safety check is possible but in practice one was never needed because safety is usually transparent as in the **member** relation. Allowing body goals to spawn before committal as GHC does is just the sort of eager speculative computation which synchronization generally tries to discourage. The eagerness of body calls in GHC should, perhaps, be regarded as a mental aberration.

Atomic unification in AGDC, CP or Concurrent Prolog cannot be advocated on the grounds of simplicity or ease of implementation. Its advantage is that the program is more declarative. Besides the other drawbacks another difficulty is that it is one of two synchronization mechanisms operating in these languages. This complicates runtime behaviour giving enormous scope for deadlock, starvation and the other evils of concurrency (e.g. Ringwood, 1988a). Furthermore, guarded output is notoriously difficult to implement. CSP (Hoare, 1985) has guarded output but its practical realization Occam (Inmos, 1984) does not. Atomic unification, or what is equivalent, obtaining exclusive access to variables, requires coordination amongst all processes that share variables which will be bound in the unification process.

As noted, the advantage of pattern matching over read-only variable unification is that it can be completely compiled. It would be equally advantageous to compile guarded unification but the variables which can be found by unification are only known at runtime. (Of course any part of unification which is known from the structure of two arguments of the equality predicate in the body can be compiled.) Replacing atomic unification by atomic assignment (specified variable binding) as suggested by Burt and Ringwood (1988) can be compiled and so is much less expensive than atomic unification. From the implementational point of view, the languages which choose to solve the two problems of premature binding and binding conflict at compiletime rather than runtime (as opposed

to those that leave the problem for the programmer to be aware of and circumvent) are both easier to understand and implement.

From an Occamist's viewpoint the flat restrictions of the languages are less complex than the languages with deep (hierarchical) guards. Flatness provides a very simple computational model in which the state of a computation is represented by a pool of processes which have a single environment and no hierarchical (subroutine) control dependencies. AND-parallelism is much easier to understand and implement than AND-OR parallelism.

FGDC has the virtue of being less complex in terms of commitment than GDC (because there are no hierarchical calls) and does remove the previously exaggerated danger of unsafe guards. Safety is achieved in the other languages by other mechanisms and is not affected by flatness. In GHC safety is achieved by runtime unification suspension and in CP and Concurrent Prolog by multiple environments. Concurrent Prolog withdrew to FCP, Flat Concurrent Prolog, because of difficulties in efficiently implementing multiple environments even on uniprocessor machines. It is clear from FGDC that in the languages CP and Concurrent Prolog multiple environments are not required to maintain safety. Multiple environments are still required in the flat subsets, Flat CP and FCP because output is generated by unification of the goal with the head of a clause; these languages don't in general use unfication primitives in the body to generate output as do GDC or GHC, though of course they could. ICOT (the Japanese Institute for New Generation Computer Technology) abandoned GHC in favor of FGHC as the kernel language of the Fifth Generation initiative because of the difficulty in efficiently implementing the runtime safety check with deeply nested guards. (It is interesting to note that Prolog interpreters produced by ICOT for FGHC actually appear to implement pattern matching, that is FGDC rather than a run time unification inhibitor.)

The view of flat concurrent logic programming as a constraint language over the Herbrand base, with the guard primitives as constraints presents an attractive interpretation. Generalizing the constraint domain beyond the Herbrand base presents exciting possibilities for future development. FAGDC appears to be the Herbrand constraint language now being advocated by Saraswat. However, both CC and FCP strongly advocate guarded output unification which is more complicated than guarded assignment as advocated by FAGDC. As FAGDC with assignment is an extension of FGDC and guarded output is deemed expensive, if thought necessary, it should be used with discretion. Rather eventual unification when it is known that binding conflict will not be a problem is to be preferred.

The constraint interpretation gives a pleasing conceptual computational model of agents cooperating to establish sets of constraints on the conclusion. The fundamental idea in constraint programming is that of producing finer and finer upper bounds to the values variables can take. At each step of the computation the blocking Ask constraints must be subsumed by the constraints accumulated so far before a clause can commit. The blocking Tell constraints can be added to the constraints accumulated so far. In this process, new Tell constraints and new variables may be introduced to the constraint pool, which further constrain the initial set of variables. This is really the ultimate expression of the idea embodied by the most general unifier (Robinson, 1965), which avoids assigning values to unknowns until they can, in many cases, be uniquely determined by constraints.

In the author's opinion FGDC is the simplest to understand and implement if you are content to leave the binding conflict problem to the programmer. It is possible to program around this problem using synchronization variables as shown by Burt and Ringwood (1988) at the expense of a less concise program. The guiding principle of FGDC is to compile everything that can be compiled so avoiding run time tests. If you are concerned with the binding conflict problem should be solved at the language operational level, and this will give greater simplicity of expression, then FAGDC with guarded output assignment (rather than guarded unification) is chosen by Occam's principle.

In the greater scheme of things the debate between the various concurrent logic languages is little more than a big endian little endian argument. In truth, the concurrent logic languages fall short of their declarative aspirations, as indeed does Prolog. But when compared with other languages which

are claimed to be suitable for concurrent programming, such as Occam, Ada and the concurrent glue, Linda (Carriero and Gelerntner, 1989), concurrent (constraint) logic languages are aesthetically superior. One can argue that a discussion of aesthetic issues is not compatible with the practice of computer science (as Carriero and Gelerntner (1989) do), and that such arguments are like whether the bits and bytes of a machine word should be numbered from the least or most significant end. Well, if aesthetic issues are not compatible with the practice of computer science then they should be.

Acknowledgements

The author is grateful to Alastair Burt and George Papdopoulos for helpful comments on an earlier version of this article and to Bob Kemp and Sasikumar, M. for proof reading.

References

Burt, A and Ringwood, GA, 1988. "The Binding Conflict Problem in Concurrent Logic Languages" submitted IJPP

Brachman, RJ and Levesque, HJ, 1984. "The Tractability of Subsumption in Frame-based Description Languages" In: Proc. National Conf on AI, pp 34–37

Carriero, N and Gelerntner, D, 1989. "Linda in Context" *CACM* **32** pp 444–458

Clark, KL and Gregory, S, 1986. "Parlog: Parallel Programming in Logic" *ACM TOPLAS* **8**(1) pp 1–49

Dikstra, EW, 1976. *A Discipline of Programming*, New York: Prentice Hall

Fuchi, K and Furukawa, K, 1986. "The Role of Logic Programming in the Fifth Generation Computer Project, *Proc. Third Int. Conf. on Logic Programming*, July 14–18, 1986, London: Springer-Verlag, Lecture Notes in Computer Science, pp 1–24.

Harel, A and Pnueli, A, 1985. "On the Development of Reactive Systems" In: Apt, KR, ed., *Logics and models of Concurrent Systems*, New York: Springer Verlag

Hirata, M, 1985. "Self-description of Oc and its Applications" (inJapanese). Proc 2nd National Conference of Japan Society of Software Science and Technology, pp 153–156

Hoare, CAR, 1985. *Communicating Sequential Processes*, New York: Prentice Hall

Huntbach M, 1988. "Parlog as a Language for Artificial Intelligence" TR, Dept Computing, Imperial College

Inmos Ltd, 1984. *Occam Programming Manual*, New York: Prentice Hall

Lassez, JL, Maher, MJ and Marriott, K, 1988. "Unification Revisited" In *Foundations of Deductive Databases and Logic Programming*, Palo Alto, Morgan Kaufmann

Maher, MJ, 1987. "Logic Semantics for a Class of Committed Choice Programs" In 4th Int. Conf. on Logic Programming, Cambridge Mass, MIT Press

Ringwood, GA, 1987a. "Pattern-Directed, Markovian, Linear Guarded, Definite Clause Resolution" submitted *JLP* but rejected after the unreasonably long time of 21 months.

Ringwood, GA, 1987b. "Predicates and Pixels" to be published New Generation computing **7**, 1989

Ringwood, GA, 1988a. "Parlog86 and the Dining Logicians" *CACM* **31** pp 10–25

Ringwood, GA, 1988b. "Meta Logic Machines: A Retrospective Rationale for the Japanese Fifth Generation" *Knowledge Engineering Review* **3**, 303–320

Robinson, JA, 1965. "A Machine-oriented Logic Based on the Resolution Principle" *JACM* **12** pp 23–41

Saraswat, VA, 1986. "Problems with Concurrent Prolog" Tech Rep 86-100, Carnegie Mellon University

Saraswat, VA, 1987a. "The Concurrent Logic Programming Language CP: Definition and Operational Semantics" Proc. of SIGACT-SIGPLAN *Symposium on Principles of Programming Languages*, New York, ACM, pp. 49–63

Saraswat, VA, 1987b. "Merging Many Streams Efficiently: The importance of atomic commitment" *In Concurrent Prolog: Collected Papers*, MIT, vol. 1, pp. 421–445

Saraswat, VA, 1987c. "GHC: Operational Semantics, Problems and Relationship with CP" In *IEEE Int. Symp. on Logic Programming*, San Francisco, pp. 347–358.

Saraswat, VA, 1989. "Concurrent Constraint Programming Languages" PhD thesis, Carnegie-Mellon University

Shapiro, EY, 1986. "Concurrent Prolog: a Progress Report" *IEEE Computer*, pp 44–58

Shapiro, EY (ed.), 1987. *Concurrent Prolog: Collected Papers*, 2 vols, Cambridge Mass, MIT Press

Shapiro, EY, 1988. "A Survey of Concurrent Logic Programming Languages", Weizmann Inst., Israel, in preparation

Taylor S, 1988. PhD Thesis, Weizmann Inst., Israel
Ueda, K, 1986. "Guarded Horn Clauses" DEng thesis, University of Tokyo
Yang, R and Aiso, H, 1986. "P-Prolog, a Parallel Logic Language Based on Exclusive Relation" In Proc. of 3rd Int Logic Prog. Conf. (London, July) New York: Springer-Verlag, pp 255–269
Warren, DHD, 1987. "OR-parallel Execution Models of Prolog" In *Tapsoft* 87, *LNCS* **250**, New York: Springer-Verlag