CMU-CS-86-100

Problems with Concurrent Prolog

Vijay A. Saraswat
Carnegie-Mellon University

May 1985.
(Revised January 1986)

Abstract. In this paper I argue that pure Horn logic does not provide the correct conceptual framework for concurrent programming. In order to express any kind of useful concurrency some extra-logical apparatus is necessary. The semantics and proof systems for such languages must necessarily reflect these control features, thus diluting the essential simplicity of Horn logic programming.

In this context I examine **Concurrent Prolog** as a concurrent and as a logic programming language, highlighting various semantic and operational difficulties. My thesis is that **Concurrent Prolog** is best thought of as a set of control features designed to select some of the many possible execution paths in an inherently non-deterministic language. It is perhaps not a coherent set of control and data-features for the ideal concurrent programming language. It is not a Horn logic programming language because it does not distinguish between derivations and refutations, because of its commitment to don't care indeterminism. As a result, soundness of the axioms does not guarantee a natural notion of partial correctness and the failure-as-negation rule is unsound. Because there is no don't know determinism, all search has to be programmed, making it a much more procedural rather than declarative language.

Moreover, we show that its proposed '?' (read-only) annotation is under-defined and there does not seem to be any consistent, reasonable way to extend its definition. We propose and justify alternate synchronisation and commitment annotations.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

CONTENTS

TABLES

1. Introduction

The notion of (Horn) logic programming is predicated on the so-called procedural interpretation of Horn logic. It is essentially the discovery that sets of *definite clauses* can be thought of as defining (in a mutually recursive fashion) inductive sets with a finite basis (corresponding to the set of 'facts').

Definite clauses are rules of the form

$$a_1 \wedge \ldots \wedge a_n \Rightarrow a \quad n \geq 0$$

where the a_i, a are atomic formulas in some first-order language without equality, any free variables in which are assumed to be universally quantified over the whole clause and $a_1, \ldots a_n$ is called the *body* of the clause and a its head. Given a set of ground atoms, such a rule says that *if* there is any instance of the rule such that (the instances of) a_i are in that set, *then* (the instance of) a too is in that set. The smallest such set is then taken to be the set inductively defined by the program. It is also the initial Herbrand model of the theory so that one has the nice property that some (ground) atom a is in this set (denoted SS_P) iff $\Sigma \models a$, where Σ is the set of all the rules.

1.1. Logic programming

The programming language interpretation of Horn logic is then the realisation that in order to determine whether an atom (also called a *goal*) $a \in SS_P$, one can look for a rule which has an instantiation which matches a and then determine, recursively, if the (instantiated) atoms in the body are in SS_P. This simple technique determines, operationally, a *tree* of possibilities with nodes labelled by goals: label the root with the initial goals, select one of them and let the node have as many sons as there are clauses whose heads will match the goal and then let the new nodes be labelled with the bodies of those clauses, with the match applied to them. (If there were more than one goal in the initial query, copy the rest of them, and also apply the match to them). The process terminates when there are no more goals left to be matched and the 'answer' is the composition of all the matches on the path from the root to this leaf.

This tree is also called an *SLD-refutation tree* for the given query, and, while other operational representations are possible, SLD-refutation is more or less a canonical way of representing the execution of a goal. This is because it is sound and complete: some instance of a goal is in SS_P iff the goal is the root of a successful SLD-tree. Moreover, if *one* SLD-tree is successful, *all* are [1].

Each predicate can thus be thought of as a recursive procedure, defined non-deterministically (because there can be more than one rule whose head could match an atom and because there can be more than one atoms in the body of a clause) and mutually recursively. The only 'primitive' in this language is procedure call via matching, everything else is progammed in terms of it. (Note that typically most general unifiers are used for matching, though this is not essential.)

1.2. Logic programming languages

From a programming language point of view, such a frame-work offers *choice-nondeterminism* in a natural fashion. But because of the widespread perception that

such non-determinacy is good for specification, but can lead to inefficient algorithms, it is thought necessary to provide some control mechanisms which would guide the search for a proof. A coherent set of such 'control features' constitutes a Horn logic programming language, and the first of these was **Prolog**. **Prolog** is a sequential and deterministic approximation to the non-deterministic model: it searches for a proof by following a depth first path through the search tree, backtracking when it encounters a failure, that is, when there is no clause whose head will match the selected goal. The essential control feature that **Prolog** introduced was *sequentiality* both in deciding which rule one should try next (the *OR-decision*) and in deciding which of a set of goals to try first (the *AND-decision*). While it can be shown (e.g. [1]) that the AND-decision is in a sense academic because if there is a proof given some choice, then there is a proof given *any* choice, the choice is still important because it determines the size of the search space.

1.3. Concurrent logic programming languages

Concurrent searches of the proof space are also possible. At the very extreme, one can have *disjoint* parallelism: to find the proof of a set of goals, find the proof of all of them independently, form their cross-product and select those which agree on the values they give to shared variables. For a given goal, to find a proof, try in parallel all the clauses which match it and return all the proofs that are found on each branch. Such a computation scheme is, however, combinatorially explosive, hence not very tractable. One would like the bindings for shared variables produced in the proof of one goal to be made immediately available to the other goals because this could reduce their branching factor: the set of clauses whose heads match an instantiated version of an atom is contained in the set which matches the atom. This introduces the notion of *advisory* communication: advisory because the receiving goal is free to ignore the communication until it has computed its own set of bindings for its variables. Advisory communication does not *guarantee* improvement in performance.

A plausible next step would be to introduce *authoritative* communication: the receiving goal *cannot* proceed until it receives bindings on (some) shared variables. In a concurrent programming language with authoritative communication, a programmer can force the execution of a highly indeterministic goal to suspend until enough information is obtained to focus the search for a solution.

It would seem therefore that to make any effective use of concurrency to speed up the search process, one would need to introduce some form of control. Ideally we would like this control mechanism not to cut off any successful paths bu[+] only failure and infinite paths. That, of course, is impossible: there cannot be any such effective and general scheme. Any general purpose control scheme, e.g. like **Prolog**'s would, in general, have to be incomplete to be useful, i.e. not exhaustive. The problem then becomes one of defining the *semantics* of these control structures. Briefly, given a query and a program with control structures, one would like to know which SLD-derivations of the pure program are allowed by the control structures. (To start with, one assumes that the control annotations would be *sound*, i.e they would not sanction any computation sequence which does not reflect an SLD-derivation. But see also Section 2.6 on fair computations.)

1.4. Concurrent Prolog

In the following we will examine one such proposed set of control structures, the language **Concurrent Prolog**, originally defined in [17]. The control structures presented therein essentially create a non-deterministic data-flow like language, where channels may be passed as values. In order for the lanuage to be a *useful* concurrent language, a number of control 'features' are needed, which take the language farther and farther away from pure Horn logic, and they cannot be given any kind of logical semantics. Reasoning about such programs is going to be concerned much more with the semantics of these control structures rather than with the logic of the underlying axioms, and hence it is doubtful what is gained by sticking to a Horn logic frame-work. While such control annotations seem to provide a useful operational frame-work, they do not seem to provide any new insights into the nature of concurrent computation.

Interest in the language **Concurrent Prolog** was generated because it has been claimed, in a number of papers, that this language is suitable for expressing a variety of interesting programming paradigms ([18], [19], [20]).

1.5. This paper

The rest of the paper is organised as follows. First, we assume familiarity with [17] and a general familiarity with the concepts of logic programming. In Section 2 we examine the informal semantics of the don't care commit operation presented in [17]. In Section 2.6 we introduce the notion of *OR-fairness* and exhibit a simple countably non-deterministic **Concurrent Prolog** program. In Section 3 we discuss various problems with the read-only annotation, whose semantics we show to be under-defined and unneccesarily complex. In Section 4 we propose an alternate annotation, the input-only annotation '↓', which is essentially a wait primitive and give a formal definition. In Section 5 we point out that, contrary to claims in the literature, the '|' commit may *not* be read declaratively as a conjunction because of its don't care nature: a query may terminate in failure even though there is a successful refutation which satisfies the constraints imposed by the control annotations. In Section 6, we develop two alternate commit operators ('\' and '&') and justify the soundness of the negation-as-failure rule for them. In Section 7, we use one of these, the parallel *don't know* commit to write a simple meta-interpreter for the language **CP**[↓, |, &], i.e. the *AND*- and *OR*-parallel Horn logic programming language with the extra-logical annotation '↓' and the '|' and '&' commitment operations.

This paper may be seen as laying the ground work for [14] where we develop a formal operational model for **CP**[↓, |, &] using the eager forms of the don't care and don't know commits and the input-only annotation.

2. Concurrent Prolog

In the following we discuss the informal semantics of **Concurrent Prolog** as defined in [17]. It must be kept in mind that this semantics is an *informal*, English description of the operational semantics of **Concurrent Prolog**, and, in places, ambiguous and imprecise. When forced to make a decision between conflicting interpretations, we try to choose an

interpretation which is 'in the spirit of things' and consistent with the other decisions that we have taken.

Concurrent Prolog adds two control structures to the Horn logic framework: the '?', or *read-only* annotation, and the '|' or *don't care commit.*

A **Concurrent Prolog** program is a set of guarded clauses. Each guarded clause is of the form:

$$a \leftarrow g_1, \ldots g_k \mid b_1, \ldots b_m \ (k, m \geq 0)$$

where the a, g_i, b_i are *atomic formulaes* or atoms for short. $g_1, \ldots g_k$ is the *guard* for the clause and $b_1 \ldots b_m$ the *body.* We will adopt the convention that any sequence of atoms $a_1 \ldots a_n : n \geq 0$ stands for true when $n = 0$.

The atoms are defined as usual for any first-order language with constant, function and predicate symbols. In addition we assume that the language has a pre-defined unary function symbol '?' which will be written in postfix notation.

The presence of '?' in a term is supposed to indicate constraints on the terms that it can unify with. In the following we quote from [17], page 11:

> The unification of terms containing read-only variables is an extension to normal unification. The unification of a read-only term $X?$ with a term Y is defined as follows. If Y is a non-variable, then the unification succeeds only if X is non-variable, and X and Y are recursively unifiable. If Y is a variable then the unification of $X?$ and Y succeeds, and the result is a read-only variable. The symmetric algorithm applies to X and $Y?$.

In [17], the author also gives a **Prolog** progam implementing this unification, which we will call ?-unification. This program is given in Table [1]. We will discuss the problems with this definition in Section 3. At this point, it is sufficient to note that the *intent* of the '?' definition seems to be that if $X?$ attempts to unify with an instantiated term, then this attempt will suspend until X is instantiated. It is also important to note that the *intent* of this definition is that most general unifiers are being computed. In pure Horn logic programming, it is convenient, *but not necessary* to assume most general unifiers. Indeed the semantics of the inductively defined set that we have given in Section 1 assumes a *ground unifier*, rather than a most general unifier. It follows that we cannot attempt to capture the semantics of a **Concurrent Prolog** program by any transformation over sets of ground atoms.

The rest of the description of **Concurrent Prolog** in [17] is given in terms of 'A sketch of a distributed **Concurrent Prolog** interpreter'. Thus, unfortunately, the semantics of **Concurrent Prolog** is given in terms of this *implementation*, which is written in a sequential language and employs one specific kind of scheduler. It is not immediately clear which of the features of the interpreter are essential, (i.e. part of **Concurrent Prolog** *semantics*) and which are just implementation details assumed for this particular implementation.

2.1. Spawning guard systems

A guarded clause is supposed to function analogously to a guarded-command. Execution of a program begins with the presentation of goals $\leftarrow q_1, \ldots q_p$. Each of these q_i will

be called an *AND-sibling* of the other. Each goal tries to reduce itself to other goals by unifying against the head of a clause.

A (variant of a) clause $a \leftarrow \bar{G}|\bar{B}.$ is a *candidate* for reduction for goal $q_i, 1 \leq i \leq n$ if $\theta = \text{mgu}_?(a, q_i)$ exists. The sequence of goals $\theta(\bar{G})$ [1] is then invoked. This is done, *in parallel*, for all the candidate clauses for a goal q_i. Each of these *guard systems* for a given goal will be called OR-siblings of each other. Note that θ is *not* applied to the other q goals at this time.

Each of these guard systems contains goals, which, in turn, may invoke other guard systems, and hence a whole hierarchy of goals can be built up, depending on the control annotations in the atoms involved. Communication of variable bindings between these sub-systems is governed by the *commitment mechanism.*

2.2. The environment for a guard system

In [17], pp. 12, the author writes

> The communication between these systems is governed by the commitment process. Subsystems invoked by a process A [i.e. goal A] have access only to variables that occur in A. As long as a process A does not commit to a reducing clause, these [guard] systems can access *only read-only variables in* A, [2] and all bindings they compute to variables in A which are not read-only are recorded on privately stored copies of these variables, which is not accessible outside of that subsystem.

Taken literally such a definition of the enviornment of a guard system presents problems.

1. What is a read-only *variable*? Uptil now, only read-only *instances* of variables have been defined, namely those decorated with a '?' annotation. Is X a read-only variable in $p(X, X?)$?
2. '?' occurrences now seem to have yet another meaning. If a binding $\{X \mapsto a, Y \mapsto b\}$ is being created by the environment of a goal $p(X, Y?)$, then it would seem that to the guard systems for $p(X, Y?)$ *only* the binding $\{Y \mapsto b\}$ is available because X, not being '?'-annotated in the original call, is inaccessible to subsystems spawned by the goal until it commits.
3. What happens to variables (e.g. V) that occur in a goal g and are unbound at call-time, but subsequently become instantiated to 'W?'? Will subsequent bindings for W be communicated to the guard systems?

2.2.1. The literal interpretation

In the following (without any justification !) let us assume that the intended interpretation of '*X is a read-only variable in a goal a*' is that '*X has* some *occurrence in a which is '?' annotated*'.

---

[1] The environment in which the guard system is called is dicsussed in the next section.

[2] *Italics* in quoted text indicate my emphasis. [Text in '[]', such as this, is my text.] Otherwise all the quotations are literally correct.

A strict interpretation of the quoted text would then require the following interpretation, which we will call *two-phase* commit [3]:

- When unifying a goal against a clause-head, maintain a separate 'global' environment for this goal which contains bindings for variables textually occurring in the goal at the time of invocation.
- When the environment changes the bindings of some of these variables by instantiating them, update the global bindings to reflect this. Then determine if the variable(s) changed were read-only *when the goal was initially invoked.* Communicate the bindings of all such variables to the guard systems for this goal.

2.2.2. Difficulties with it

Such an interpretation seems hard to justify. It limits drastically the information that can be exchanged between the environment of a goal and its guard systems, and is really contrary in spirit to the idea that as much information as possible should be shared so as to avoid redundant searches for *incorrect* solutions.

Moreover, under this interpretation, one cannot write some simple programs. For example, one cannot write a general program which first inputs any two lists of terms and then, all at once, produces some indeterminate merge of these two lists.

Example 1: Consider the following program as a candidate:

```
merge([A|X], Y, [A|Z])← merge(X?,Y,Z)| true.
merge(X, [A|Y], [A|Z])← merge(X,Y?,Z) | true.
merge([], Y, Y).
merge(X,[],X).
```

This will not work, e.g. with the call `?-merge(D?, E?, F), produce(D), produce(E).` where `produce/1` may produce its values one at a time. Assume that when the `merge` goal is invoked, `D` and `E` are unbound. Then the *first* time `D` (say) gets a value e.g. $\{D \mapsto [1|G]\}$, this value will be communicated to the guard systems for `merge`, *but subsequent values for* `G` will not! Hence the system will remain 'dead-locked'. ⊗

If it is indeed thought desirable that the programmer should have control over which variables in the goal allow subsequent bindings from the environment to be transmitted into the guard system then it is advisable to have some *other* extra-logical annotation to express this. Overloading the '?' with different meanings makes programming very obscure and difficult.

2.2.3. Possible justification for flat systems

If we do not allow such complex guard systems as in Example 1, then one scenario in which this restriction seems meaningful is if *all* those variables in a goal which are not '?'-protected will have their bindings created by the execution of this goal. Then, since no other goal (i.e. the environment) could be creating bindings for these non-read-only variables, the question of propogating these values to the guard systems is academic.

---

[3] because information transfer between the environment and a guard system takes place principally in two phases: at *spawn* time and at *commit* time

But this is a very strict interpretation in which every variable occurring in some AND-conjunctive system can have exactly one 'producer'-occurrence: in all the others it must be explicitly read-protected.

2.2.4. But we would like multiple producers!

However, as pointed out in [17], one would, in general like to allow *multiple* generators for the bindings of a given variable. In such cases, it makes sense to propogate the bindings for non-read-only-annotated variables to the guard systems so as to help them choose the value of the variable which has now been declared as the 'final' one, from other possibilities which are now guaranteed to end in failure. Here is the canonical example.

Example 2: Consider the program:

```
p(a).
p(b).
?-p(X), p(X).
```

Both the `p(X)` goals will spawn off two guard-systems, with each terminating successfully in the bindings $\{X \mapsto a\}$ and $\{X \mapsto b\}$ respectively. One of these will commit for each `p(X)` goal. Suppose the first one commits for the first goal. Now the binding $\{X \mapsto a\}$ is published by the first goal. But since X is not read-only annotated in its only occurrence in the second `p(X)` goal, the binding of X will not be visible to the two guard systems for this goal. The second one can then commit, leading to failure.

On the other hand, if the binding $\{X \mapsto a\}$ was communicated to the guards of the second goal, then the second guard-system would not be able to commit, because it has inconsistent bindings, thus giving the result $\{X \mapsto a\}$. Similarly the result $\{X \mapsto b\}$ is a possibility, but *failure* is not. $\otimes$

2.2.5. Conclusion

Therefore, making a distinction between variables that are read-only in a goal and those that are not is then a source of *spurious* failures, i.e. failures that are caused not by the logic of the program but the way search is carried out for a proof. We feel that the aim of any logic programming language should be to facilitate search for a proof and to keep such failures to a minimum.

Accordingly we propose another kind of commit, an *eager* commit. The aspect of this commit [4] relevant here is that when the environment creates bindings, these will be communicated to all guard systems without regard to whether they affect read-only variables or variables that were present when the goal was originally invoked etc. Under such an interpretation, information is shared as soon as it is available. However, as we shall see later such eager sharing leads to problematic interactions with the semantics of '?', which allow for dynamic creation of read-only variables.

[4] discussed in more detail in Section 2.3.2

2.3. The commitment mechanism

There must exist some clauses whose guards are empty, otherwise no guard system can ever terminate. Assume then that the guard system $\theta(\bar{G})$ has been solved successfully, i.e. has terminated with the answer substitution σ. For example, if $\theta(\bar{G})$ was empty (i.e. true) it may immediately terminate with the answer substitution θ. If more than one guard systems for a given goal have terminated successfully, exactly *one* of them is chosen to commit, by some mutual exclusion algorithm. Quoting again from [17]:

> Upon commitment to a clause $A : -G|B$, the private copies of variables associated with this clause are unified against their public counterparts, and if the unification succeeds the body system B of the chosen clause replaces A... After gaining such a permission [to commit], the unifier attempts to unify the local copies of its variables against the corresponding global copies. If successful, then the commitment completes successfully... When committing, the unifier is not required to perform the unification of the public and private copies of variable as an 'atomic action'. The only requirement is that the unification be 'correct', in the sense that it should not modify already instantiated variables, which can be achieved in a shared memory model with a test-and-set primitive.

2.3.1. Two-phase commit

Under two-phase commit, three things can happen when a successfully terminated guard system is selected to commit: its bindings are either compatible with the global bindings, or incompatible, or lead to a suspension due to read-only violations.

This raises the question: why should a terminated guard system be chosen *before* its bindings are checked against the global bindings? If there is a *pre-commit* stage in which the bindings of the terminated guard system are checked against the global bindings, then one can prevent such guards from committing as would lead to an immediate spurious failure. Similarly, if on checking it is discovered that committing these bindings would lead to suspension, it certainly makes sense to allow other successfully terminated guards a chance to commit. Discussion of such a pre-commit phase is conspicuously lacking in [17]. If the intention in [17] was to have a two-phase commit, then a pre-commit seems essential. [5]

2.3.2. Eager commit

A pre-commit stage can be avoided under eager commit. In this scheme, when a guard system commits, its unification with the 'global bindings' is done *atomically*, and these bindings are conveyed 'instantly' to all the guard systems for the goal. (Note that a guard system may actually have an hierarchy of goals: the bindings are communicated through all levels of the hierarchy)

The reason for publishing bindings atomically and conveying them instantly should be apparent from the following example.

---

[5] and, incidentally, gives another justification to the name 'two-phase commit'

Example 3: Consider the same program as in Example 2. Each of the two p(X) goals spawn off two guards. Let, for both the first one compute the binding $\{X \mapsto b\}$ and for the second $\{X \mapsto a\}$. Now assume all four guard systems are ready to commit. Suppose for the first goal, the first guard system has been selected to commit. Then, *before* it attempts to publish its bindings, the second guard system for the second goal has been selected to commit. Failure will now ensue.

On the other hand, suppose publication is atomic. Now, when the first guard for the first goal commits, the new 'global' value of X is communicated to the guard systems of the second goal. Now the second guard system for this goal *cannot* commit because it has inconsistent bindings. Hence the system is guaranteed to succeed, with either the values $\{X \mapsto b\}$ or $\{X \mapsto a\}$. $\otimes$

Now the notion of 'conveying' must be defined more carefully. It is clear that if the new committed binding is incompatible (i.e. does not unify with) the binding given by a goal in the guard system, then the goal should be considered *inconsistent* and not allowed to continue: the guard system it is a member of should fail. (This is the analogue of not allowing incompatible bindings to commit in the pre-commit phase of two-phase commit, except that here a goal is aborted as soon as it is discovered that it is not compatible with global bindings).

It should also be clear that in general the binding being committed could have some ?-terms in it: hence the unification done at commit time will have to be ?-unification. So there is a possibility that this unification may cause a read-only violation and some action akin to a suspension may have to be taken.

Example 4: Consider the program:

```
?-p(X), q(X).
p(Z?).
q(a):- s(b) | true.
s(b).
```

Consider the following sequence of events. The goal q(X) reduces, creates the binding $\{X \mapsto a\}$ and spawns the guard system s(b). Now p(X) reduces and commits, creating the binding $\{X \mapsto Z?\}$. Now, under eager commitment, the binding of X must be 'made available' to the guard for the goal q(X) and hence the unification of $Z?$ with a must be attempted. But such a unification should cause the suspension of *something*. $\otimes$

What has happened is that the goal q(X) reduced under the assumption that it was going to create a value for the unguarded variable X. But then the goal p(X) also created a value for X. If p(X) had bound X to an instantiated term, then this binding could have been checked against the bindings (here only one) created for X by the guard systems for q(X) and allowed only those guard systems to proceed which had a compatible value. But since p(X) produces the binding $\{X \mapsto Z?\}$ this means that some other goal (here none) is going to produce the value for Z in the future and the value of X must be whatever is produced then. Presumably, one should now freeze all the guard systems of q(X) which have *guessed* a value for X (and its descendants) because they were not supposed to. (In this case the guard system s(b) must be frozen because one of the bindings created by its parent goal q(X) when unifying agains the head of this clause is now realised to have incurred a read-only violation.)

As should also be clear from this discussion, such a situation could arise only because read-only annotated variables can be *created* at run-time under the present semantics of ?-unification. In our opinion the above example clearly demonstrates the undesirability of such behaviour with respect to an eager commit. Later (in Section 4) we will present another extra-logical synchronisation annotation '↓' for which the bindings being committed will always refer to pure terms, so that at commit time only pure unification needs to be done, and there is no possibility of freezing active guard systems.

2.3.3. Eager commit needs no 'global' bindings

If the above rules are followed, it is also easy to see that, in fact, no 'global bindings' are needed. Instead, a (local) environment can be associated with each goal, so that looking up the value of a variable means checking its value in this environment. Hence lookup is fast, but, of course, commit is slower because it has to traverse the entire processor sub-trees of AND-siblings. A formal operational model using these ideas is presented in [14].

2.3.4. Is eager commit realistic?

There might be some questions about how bindings can be published so as to be made available to all goals simultaneously, e.g. if the program is being executed in a distributed network. Our point is that it is upto the language to specify the desired behaviour, and upto an implementation to achieve it. Here we have chosen to stick as closely as possible to a Horn logic programming model and prefer its simplicity, even if it might incur some expense at implementation time. Of course, bindings do not have to be made available immediately, as long as the implementation guarantees that it would behave as if they were. For example, each goal might need to keep the other guard systems around and backtrack, if its proposed solution was found incompatible with some previously announced binding which hadn't yet made it to the goal.

2.3.5. Justification of restrictions in two-phase commit

The possibility of an active guard system being suspended because of the creation of a read-only annotated variable has not been discussed in [17] at all and is the only reason why we suspect that two-phase commit, unmotivated though it seems, might have been the intended interpretation there. By restricting the bindings propogated into the guard systems to be bindings for variables initially read-only annotated, it is ensured that no suspensions of active guard systems may arise.

On the other hand, if it is ascertained that any conjunctive system has at most one producer instance (i.e. non ?-annotated instance) of each variable, then as discussed earlier, it is not necessary to restrict the propogation of bindings to ?-annotated variables in order to ensure that no active guard systems are frozen.

2.4. Execution of bodies

When a guard system for a goal successfully commits its bindings, the *body* goals of the clause variant have to be executed in the new environment. As before, there must

exist some clauses whose body is empty (i.e. `true`), otherwise the execution of no body-system will terminate. When a body system does terminate, with answer-substitution σ, say, execution continues with the AND-siblings of the original goal, with σ reconciled with their environments. If there are no AND-siblings, then execution terminates with σ the answer-substitution.

2.5. Other 'features' of Concurrent Prolog

In the literature on **Concurrent Prolog**, one also finds the use of some other control primitves, which are listed here for the sake of completeness. A semantics for **Concurrent Prolog** should include a formal definition of these predicates and point out their relationships with one another. Most of these primtives are very operational in nature and, just as it is the case for the corresponding primitives in **Prolog**, it is difficult to rationalise their presence in a 'logic' programming language.

`var/1`: The goal `var(X)` succeeds immediately if `X` is a variable, and *fails* otherwise. While `var/1` is present also in **Prolog**, it is a source of semantic difficulty here in that, with *AND*-fairness, it allows countable non-determinism. See Section 2.6.

`wait/1`: The goal `wait(X)` suspends until `X` is instantiated, and then succeeds. This is essentially equivalent to the '↓' annotation we introduce later.

`otherwise/0`: The definition, given in [18] states that an `otherwise/0` goal can occur only as the only goal in a guard and the intended semantics is that this goal succeeds if and when all other guard systems for the parent goal fail.

Note that any attempt to give a 'logical' interpretation to otherwise must deal with the problem of the meaning of failure in **Concurrent Prolog**.

In [17], another predicate `diff/2` is also used, but its semantics is not clearly defined in an implementation-independent fashion.

It is also worth noting that the author of [17] has also introduced the notion of *stability* of **Concurrent Prolog** *implementations*. Our position is that since stability is not a property of the *language definition*, advantage cannot be taken of its properties when proving properties (e.g. bounded response time) of **Concurrent Prolog** programs. It is also not relevant to the current discussion which is about the *semantics* of **Concurrent Prolog**.

2.6. Fairness

An important point that is not mentioned in [17] is that of OR-fairness. It is implied that all the guard systems are spawned off in parallel and that no further action is taken until one of them is ready to commit. But this just specifies what *can* happen, not what *will*. The notion of OR-fairness says that all the guards will be allowed to advance if they can advance: no guard will be discriminated against. If in a system of guards some *can* terminate successfully, then at least one *will*. So if one guard system executes a non-terminating sequence of computations, and another one a finite successful sequence, then the second one wins, i.e. will be selected to commit, other thigs being equal. While it is not explicitly stated so in [17], one assumes that this is the desired behaviour.

If this is the intended interpretation (which is plausible, see comment on pp.48, [17].) then commitment is analogous to a local **amb** operator [8] in that of all guard execution sequences it chooses successfully terminating ones in preference to non-terminating ones or those that terminate in failure. We can make use of this local angelicism in writing the following uncountably nondeterministic program which is guaranteed to terminate, but whose output cannot be bounded *a priori* by any integer. Such a program cannot be written in pure Horn logic, which has only choice-nondeterminism, which, by König's lemma, is bounded.

Example 5: Consider:

```
p(s(X)) ← p(X) | true.
p(0) ← true | true.
```

Given the query ← p(X)., by OR-fairness, the only execution that is cut off is the one in which each p(X) goal chooses the first clause repeatedly on every incarnation. Hence the set of possible answers is $\{s^n(0)|n \geq 0\}$ and the query always terminates. ⊗

The similar issue of AND-fairness has been discussed in the context of Horn logic programming in [7]. Intuitively the idea here is that in a conjunctive system of goals, if one of them can make a transition (e.g. by committing or reducing against clauses), then it utimately will: that is, it will not be consistently ignored.

Lemma 1: Given AND-fairness, the var annotation '↑' is sufficient to write a countably non-deterministic Horn logic program, without assuming OR-fairness.

Proof 1: Here is the program:

```
p(X) ← p(X,Stop), stop(Stop).
p(s(X), Stop↑) ← p(X,Stop).
p(0, stop).
stop(stop).
```

Parenthetically, we note that the reason we need the '↑' annotation and cannot do with the '↓' annotation (to be introduced in Section 4) is that in these languages, a variable can become bound, but never unbound. The '↓' annotation prevents some action from taking place *until* a variable becomes bound and the '↑' *after* a variable becomes bound. Hence the '↑' is able to cut off a potentially infinite number of solution paths: the '↓' can only delay, perhaps indefinitely, it cannot prevent. Hence in the above example, we need the ability to stop, i.e. not examine any further branch, at some time in the future and return some result. The '↑' is tailor made for that.

3. The read-only (?) annotation

We now turn to a discussion of the semantics of the read-only '?' annotation. In the following, X and Y represent variables and $T, T_1, \ldots$ represent compound terms, i.e. terms of the form $f(X_1, \ldots, X_n)$ where $n \geq 0$ and the X_i are terms.

First, let us recall that in **Concurrent Prolog** processes are goals, and the input and output channels for the process are represented by the occurrences of variables in the atomic formula representing the goal. The only action that can be taken by a goal is to

```
unify(X,Y)←
      (var(X); var(Y)), !, X=Y.
unify(X?,Y) ←
      !, nonvar(X), unify(X,Y).
unify(X, Y?) ←
      !, nonvar(Y), unify(X,Y).
unify([X|Xs], [Y|Ys]) ←
      !, unify(X,Y), unify(Xs,Ys).
unify([],[]) ← !.
unify(X,Y) ←
      X=..[F|Xs], Y=..[F|Ys], unify(Xs,Ys).
```

Table 1: The **Prolog** program for ?-unification. (See Reference 8.)

unify against the head of a clause. This unification typically would lead to an instantiation of various variables in the goal. However it is undesirable that variables that are supposed to represent input channels should be instantiated. Also, a clause typically implements some kind of a (possibly multi-valued) function from its input arguments to its output arguments and hence it is unwise for process reduction to continue until values for the input arguments are available. Both of these objectives are sought to be achieved by the introduction of the 'read-only annotation' (?) which can decorate instances of a terms in goals and clause heads and whose semantics is specified in Table [1] and in Section 2.

One can attempt to justify the decision to suspend when unifying $X?$ against T by saying that it effectively allows a goal to do a *case* on the possible inputs that might be provided via the variable X. That is, the environment may instantiate X to any *one* of $T, T_1, T_2 \ldots$ and then only the guard systems for clauses which have the correct term will be invoked. But this is useful only in the special case in which $X?$ occurs in the *goal* and the T_i in clause-heads. Even for such occurrences of annotated variables, the following conditions seem necessary:

Condition 1. At runtime, in every goal, if even one occurrence of a variable is read-only, then all instances are read-only. (Hence that variable may legitimately be considered a read-only *variable* in the call). This corresponds to the normal restrictions in data-flow languages that an input stream to a process cannot also be an output stream.

Condition 2. In every clause-head if even one occurrence of a variable is read-only, all occurrences of that variable should be read only.

Condition 3. If a variable is read-only annotated at some occurrence in a clause-head, then ensure that all goals that would reduce via this clause have a variable at that occurrence.

Condition 2 is symmetric to Condition 1, but for ?-occurrences in goals, a restriction corresponding to Condition 3 is not required; this is a fundamental difference between the use of ? in goals and in clause-heads in **Concurrent Prolog**.

In the following we discuss why these conditions seem indicated and the problems that arise even if they are followed.

3.1. Unification is order-dependent

Typically to unify compound terms T_1 and T_2, one checks the functors are the same (i.e. they have the same name and the same no. of arguments) and then unifies the corresponding arguments in the two terms *in any order*. This no longer works. e.g. `unify(f(X?, X), f(a,a))` suspends if the first arguments are matched before the second but succeeds otherwise. The semantics hitherto presented does not consider this at all. The program in Table [1] assumes a left-to-right order of unification of arguments, but it is not clear if this is an artifact of the implementation or is the desired semantics. If it is the desired semantics, then it is yet one more point of departure from logic programming, and it is not clear whether enough benefits accrue from this commitment to justify the loss in semantic simplicity.

One other possible semantics could be that all the arguments are unified in *parallel*. In that case `unify(f(X?,X), f(a,a))` would succeed because unification of the first arguments would remain suspended while the unification of the second arguments succeeds with the binding $\{X \mapsto a\}$ and then causes the unification of the first arguments to resume and succeed, resulting in `unfiy(X?, X), f(a, a))` succeeding with the bindings $\{X \mapsto a\}$. This is the interpretation we prefer because it is consistent with the behaviour of the system `?-g(X?), h(X).` when the only clauses in the program are `g(a).` and `h(a).`

Lemma 2: Conditions 1 & 2 are necessary and sufficient syntactic restrictions, to ensure order-independence of ?-mgu. That is, if Conditions 1 & 2 are satisfied then two ?-annotated terms have a ?-mgu with left to right scan iff they have an ?-mgu with any scan.

3.2. Spurious suspensions

The view ?-as-input-designator is useful *only* for '?' occurrences in goals (i.e. in bodies of clauses) *not* in heads of clauses. This is because if $X?$ occurs in the head of a clause at argument i, and argument i in the goal is a constant, *suspending* till X gets a value is, in general, meaningless because the only way a binding for X can be generated is by executing some goal in the body of the clause. *But that cannot be done if unifying $X?$ against a compound term suspends.*

This deadlock can only be avoided if parallel unification is assumed and Condition 2 is *violated*, i.e. there is also an occurrence of X in the clause-head which may (possibly in the future) match an instantiated term.

3.3. Unification of X? with Y

This raises the issue of why unifying $X?$ with Y should be defined to yield the binding $\{Y \mapsto X?\}$. Originally the view of '?' was as an occurrence-specific annotation, but this definition causes *all* occurrences of Y to be annotated. This seems difficult to justify, considering the original motivation for '?' as being a synchronising mechanism. It leads to the following problems:

3.3.1. Obscure code

Because '?' annotations can be propogated at run-time, a static analysis of clauses for a

predicate is not enough to define its behaviour: its *meaning* depends upon the annotations of the arguments to the goal. This makes proofs of properties of programs difficult.

3.3.2. Freezing active goals

If a goal has an occurrence of a variable X in it, this X can be converted into a read-only copy of some other variable by another conjunctive goal which also has an occurrenceof X. One kind of problem that it can raise for eager commit has already been discussed in Example 4. It is easy to concoct more examples where freezing active goals can have very unexpected consequences.

Note that this problem was not caused *because* of the appearance of a ?-annotated term in the head of a clause. The same situation would have arisen in Example 4 if instead of the clause `p(Z?)` we had the clause `p(Y,Y)` and the call `p(Z?,X)` instead of `p(X)`.

3.3.3. Unifying X against X?

The **Prolog** program for `unify/2` will loop because **Prolog** does not have the occurrence check. On the other hand, according to the definition given above for the case $X? - Y$, X should now become a read-only version of X in *all its occurrences* (here in its occurrence in `p(X)`) so that there will be no more producers for X left and the system is sure to deadlock.

An alternative possibility is to let the unification of X with $X?$ succeed with no binding being generated.

Lemma 3: Conditions 1 & 2 are necessary and sufficient to ensure that no call of the form `unify(X?,X)`, for some variable `X` arises as a result of any call to `unify/2`.

3.3.4. '?' remains unintuitive

Even if we were to define that `unify(X?,X)` should succeed with no bindings being generated, some systems of goals may still exhibit rather unintuitive behaviour.

Example 6: Consider the program

```
X=X.
```

and the query `?- Y=a, Y=X?, Y=X`. Intuitively, this query should mean that `Y` has a producer (the first goal, which can produce the binding $\{Y \mapsto a\}$), has read only access to another variable (`X`) and also has 'full' (i.e. read- and write-access, to `X`. One would then imagine that this query should succeed with the bindings $\{ Y \mapsto a , X \mapsto a \}$. This will happen, however, *only* if either the first or the third goal executes first.

If `Y=a` executes first we next have the system `?- a=X?, a=X`. Now the goal `a=X?` is blocked because of read-only violations, but the goal `a=X` can succeed, leading to the system `?- a=a?.` which will succeed.

If `Y=X` succeeds first, we will have the system `?-X=a, X=X?.` in which both the goals can succeed, and in either order.

On the other hand, if the goal `Y=X?` were to succeed first, we would have the system `?- X?=a, X?=X.` in which the second goal can succeed, *but the first must remain deadlocked!* ⊗

A goal system such as `?- Y=a, Y=X?, Y=X.` may seem rather contrived as it stands, but such sub-systems can easily arise during the execution of complex goals. What is unintuitive about this example is that even though one may think that a variable `Y` is being given full access to another variable `X` (via a goal such as `X=Y`), in reality such access may be denied, for example if earlier the variable `Y` had been given read-only access to `X`. In **Concurrent Prolog** read and write accesses are not additive!

This is just one of the many pitfalls that a beginning **Concurrent Prolog** programmer has to beware of and it arises because there is no consistent, intuitive interpretation for the behaviour of '?'.

3.4. Unification of $X?$ and $Y?$

What happens in the case where $X?$ has to be unified against $Y??$ The informal description of ?-unification does not discuss this case. According to the `unify/2` program in Table [1], `unify(X?,Y?)` fails. This program was taken from an implementation in which such failure leads to busy-waiting: hence we should think of the failure as meaning that the attempt to unify $X?$ against $Y?$ should *suspend.* In fact this unification will remain suspended (i.e. the calls continue to fail) until both X and Y have been instantiated to a value, when these values will be recursively unified.

There is really no a priori justification for causing unification of $X? - Y?$ to suspend until both X and Y get values and then to recursively unify them. It is as justified as allowing the unification of $X?$ with $Y?$ to succeed with X unified with Y, without suspending. Of course, these two definitions lead to different behaviours [6].

Suffice to note here that suspending unification of $X? - Y?$ until both X and Y are unified would lead to deadlock in case one of these variables occurs in a clause head (and hence all its occurrences are also in the clause head). This partly justifies Condition 3.

On the other hand assuming the second definition of $X? - Y?$ unification, if a goal contains an occurrence of X? and is unified against a clause head which contains a Y? in that argument position, then the result is as if the process becomes a producer for the channel, even though it was earlier passed just a read-only reference at run-time. Thus if a process always wants write-access to a channel, it can do that by placing a $X?$ at the argument position in the clause-head where it expects the channel to be supplied. In this fashion,a process has to make fewer assumptions about the behaviour of its environment at run-time. But deadlock may also ensue. See Section 3.2.

3.4.1. Unification of $X?$ with $X?$

According to the first definition, this unification should suspend until X gets a binding, when it would be checked against itself! This seems even more difficult to justify. On the other hand, by the second definition, this would immediately succeed with no bindings being created.

[6] and have been discussed by the author in more detail in [12]

3.5. Conclusion: the '?' is difficult to understand

To sum up, then, I would argue that the intuitive interpretation of '?' as a 'read-only' designator makes sense in only some of the many possible ways in which it can be used. If a user is supposed to think of the '?' precisely as designating which occurrences of variables are 'read-only', then perhaps the language should then provide syntactic restrictions which encourage (or maybe even sanction only) such use. Conditions 1-3 are a step in that direction.

Moreover the definition of the '?' given in [17] is not even complete. There seem to be a variety of ways in which to complete the definition [7] and none seem to have a clean motivation. Rather these definitions lend themselves to singularly obscure use as evidenced by the series of programs by the author in [10] and [13] which show how to use each of three different versions of the read-only annotation (proposed in [11] and [12]) to provide a weak simulation of `var/1` and hence obtain programs for the bounded merge predicate.[8]

In the next section we outline two synchronisation primitives, the 'input-only' annotation '↓' and the 'output-only' `var/1` annotation '↑'. We show how we can use both these primitives to simulate ?-unification, but the '?' alone cannot be used to get the behaviour of either of these primtives because it is much too powerful.

4. The input-only (↓) annotation

In order to use the non-determinism inherent in Horn logic to model specify and implement concurrent algorithms, it is essential to have a *wait* facility, which forces a goal to suspend until it receives bindings for some of its variables from the environment. In the following we propose and justify such an annotation to control communication of bindings between conjunctive goals.

This annotation has been inspired by the discussion of problems with the read-only annotation in the last section. Annotations similar to this have previously been discussed in the literature on logic programming: indeed given the problem of communication between conjunctive goals, the design space of solutions is rather small. But, typically, the semantics of such constructs is not precisely defined and a plethora of such constructs is proposed. On the contraray, we propose '↓' as a *replacement* for the '?'-annotation: the new language, called **CP**$[\downarrow, |]$, has a much simpler semantics but all the original **Concurrent Prolog** programs hitherto published can be translated simply into it.

4.1. The '↓' annotation

First we will assume that unification of two compound terms T_1 and T_2 is carried out by checking if T_1 and T_2 have the same functor and arity and then unifying their corresponding arguments in *parallel*.

---

[7] e.g. see some of the proposals in [11] and [12] by Tony Kusalik, Udi Shapiro, Jacob Levy and the author

[8] Some of these programs need further the assumption of strict *AND*-fairness.

The '↓' annotation can only decorate instances of terms in the head of a clause. If a term (variable *or* constant *or* compound term) $T\downarrow$ occurs in the head of a clause, then unification of $T\downarrow$ with V, where V is a variable will suspend and remain suspended until V has been instantiated (to a constant or a possibly non-ground compound term), after which the unification of the terms T and V will be attempted.

Like the '?', the '↓' is not inherited by embedded terms, that is, it applies only to the term instance textually indicated in the program. However, if '↓' annotates a term t_1 inside a term t, then it must also annotate all sub-terms of t which contain t_1 (including t). In fact, we will define an embedded occurrence of a '↓' to be shorthand for just such a series of '↓' annotations in the term. (The atom at the head of a clause is always ↓-annotated.) This restriction is necessary to prevent occurrences of `unify(X, f(a↓))` which does not make sense because a sub-term cannot be required to be present unless the super-term is also required to be present. This design decision essentially precludes the creation of 'protected embedded channels'.

In so far as a term $V\downarrow$ will suspend when unifying against a variable, the meaning of '↓' is similar to the meaning of `wait/1`. (See Section 2.5.)

Here are some simple properties of this notation:

- `unify(Y↓, X↓)` can never occur.
- `unify(Y, Y↓)` can, and suspends till `Y` is instantiated.
- There is no 'inheritance' of '↓' via `unify(X↓,Y)` like there is for `unify(X?,Y)`.
- The '↓' annotation can never 'occur' in any goal at runtime.

With '↓' each *clause* decides what is to be *input* to it. In other words the '↓' annotation is used to restrict the goals for which a clause is applicable by specifying which terms in the goal need to be instantiated.

If all the clauses have the same pattern of input specifications, then the '↓' annotations could be removed in favor of a mode-specification for the predicate. Since nested '↓' annotations are allowed within a term, in general it is not possible to remove '↓' annotations in favour of mode declarations. (Of course, every program annotated with the Dec-10 Prolog '+' (input) or '?' (dont-know) annotations can be rewritten using '↓' annotations; hence '↓' is more 'general'.)

4.2. Examples exhibiting expressivesness of '↓'

Example 7: This is the canonical example of two operations on a variable that is treated like a channel.

- `send/3`. Equivalent mode `send(+,?,?)`

```
send(Message↓, [Message|Channel], Channel).
```

  Typically, a call to `send/3` would be '... `send(Message, Channel, NewChannel)` ...'. The `send/3` goal waits until the environment instantiates the variable `Message` and then 'sends' it down `Channel` by unifying it against a list whose first element is `Message` and the rest is a new list, the `NewChannel`.

- `receive/3` Equivalent mode is `receive(?,+,?)`.

```
receive(Message, [Message | NewChannel]↓, NewChannel).
```

Complementary to send/3, receive/3 occurs typically as a call '... receive(Message, Channel, NewChannel)...'. It succeeds only when Channel is instantiated by the environment to a list. The first element of the list is then taken to be the current Message, and the rest of the list is now NewChannel.⊗

Example 8: merge/3. Equivalent mode is merge(+,+,?).

```
merge([A|X]↓, Y, [A|Z]):- merge(X, Y, Z).
merge(X, [A|Y]↓, [A|Z]):- merge(X, Y, Z).
merge(nil↓, Y, Y).
merge(X, nil↓, X).⊗
```

Example 9: plus/3. No single equivalent mode declaration.

```
plus(X↓, Y↓, Z):- Z is X+Y.
plus(X↓, Y, Z↓):- Y is Z-X.
plus(X, Y↓, Z↓):- X is Z-Y.⊗
```

4.2.1. Multi-use predicates in CP[↓, |]

The following program expresses behaviour that would be difficult, if not impossible, to express in **Concurrent Prolog, GHC** ([23]) or **Parlog** ([2]) which are all concurrent logic programming languages whose extra-logical annotations emasculate the notion of a logical variable. It follows that in order to exploit stream *AND*-parallelism and *OR*-parallelism it is not *necessary* to do away witht the logical variable, which must be regarded as one of the original contributions of logic programming to programming language theory. In **CP**[↓, |, &] it is still possible to have 'multi-use' predicates as in **Prolog**.

Example 10: p/4. The partitioning program.

```
p([Next | Rest], Pivot, [Next | Less], More):-
Next ≤ Pivot | p(Rest, Pivot, Less, More).

p([Next | Rest], Pivot, Less, [Next | More]):-
Next ≥ Pivot | p(Rest, Pivot, Less, More).

p(nil↓, Pivot, nil, nil).
p(nil, Pivot, nil↓/1, nil).
p(nil, Pivot, nil, ni↓/1).


Next↓ ≤ Pivot↓ :- Next :≤ Pivot.
Next↓ ≥ Pivot↓ :- Next :≥ Pivot.
```

In this program :≤ /2 and :≥ /2 are built in arithmetic predicates.

The power of this program lies in the fact that it will work 'correctly' independent of whether it is consuming/producing any combinations of its arguments. Consider specifically the following queries, on all of which it works correctly:

```
0.  ?-p([1,2,3,4,5], 2, Leq, Geq).  ⟹
{ Leq ↦ [1,2], Geq ↦ [3,4,5] } or
```

```
{ Leq ↦ [1], Geq ↦ [2,3,4,5] }.

1.  ?-p([1,2,3,4,5], 2, [1,2], [3,4,5]).  ⟹
true.

2.  ?-p([1,4,2,3|X], 2, [One, Two,2], [4,3,5]).  ⟹
{ X ↦ [2,5], One ↦ 1, Two ↦ 2 } or
{ X ↦ [5,2], One ↦ 1, Two ↦ 2 }.

3.  ?-p(X, 2, [1,1,2], [3,5,6,2]).  ⟹
{ X ↦ any merge of [1,1,2] and [3,5,6,2] }.

4.  ?-p(X, 2, [1,102,2], [3,5,6,2]).  ⟹
fail.
```

One can think of a p(Big, Pivot, Leq, Geq) goal as essentially being a transducer which needs a value for Pivot before it can be activated, and then monitors its three streams (Big, Leq Geq) until it gets a value down any one of them. It then dispatches that value to the appropriate stream. If it receives a nil down any channel, it terminates. Other termination conditions (e.g. the clause p(X, Pivot, nil↓, X).) are also possible. ⊗

The reason the above example 'works' is because when a p/4 is unified against the head of the first two clauses essentially *pure* unification is being done. The '↓' annotations in the heads of the next three (base) clauses ensure that they are never used until and unless one of the three streams is closed. Pure unification may be done in **Concurrent Prolog** too by not using any '?'-annotations in the program, but then there would be no way in a **Concurrent Prolog** program to specify that a clause should not be considered until some extra bindings are made available.

As far as the languages **Parlog** and **GHC** are concerned, they cannot express such behaviour because of their insistence on using matching rather than unification.

Another advantage of '↓' over '?' is that '↓' never causes the *creation* of annotations at run-time. Hence the eager commit defined earlier does not cause active goals to be suspended, because, it can easily be seen, with '↓' once a goal is unblocked, it remains unblocked. We believe that '↓'-unification should also be easier to implement.

4.3. The '↑' annotation

One can show that with the various definitions of '?' discussed in the last section, it is possible to simulate var/1, assuming strict *AND*-fairness, and hence to get a countably non-deterministic program. [9] This is not possible with the ↓-annotation, which is therefore less powerful than '?'. In practice, however, the ↓-annotation avoids the contentious issues with '?' and presents a simple, clean and powerful primitive.

To get the entire functionality of ?', var/1 must also be used. Its usefullness can be enhanced if it is also treated like an annotation ('↑', say) which can occur only in the heads of clauses. Again, it annotates just the occurrence of the Term it is textually adjacent to.

[9] See [13] for a program that presents a bounded merge program using '?' and strict *AND*-fairness.

As before, if '↑' annotates a term t_1 inside a term t, then all subterms of t which contain t_1 (including t) are presumed to be '↓' decorated (so that `unify(A, f(V↑))` will suspend until `A` is instantiated to an `f/1` term). As usual this obviates the creation of 'protected embedded channels'.

The unification `Term↑-X` succeeds only if `X` is a variable and results in `X` being unified with `Term`. If `X` is not a variable, then `Term↑-X` *fails*, though this must be regarded as a control failure. Operationally this means that the given clause cannot be used at all for the given goal. [10] `Term↑-X` *never* suspends.

This definition of the semantics of '↑' is ambiguous because it does not state what should happen in the situation when it is possible to unify a term t_1 with a term t_2, which contains a subterm $t\uparrow$ such that the subterm of t_1 unifying with $t\uparrow$ may be a variable if unification is done in some specific order and may not be, if it is done in some other order. For example, with the above definition, the unification of `f(Y, Y)` with `f(a,a↑)` may succeed if unification is attempted from right to left, but will fail otherwise. In such situations, we will define the unification to have *failed*, thus preferring not to succeed when there is a possibility of failure.

Note that nested occurrences of '↑' do not make sense.

Example 11:
- `foo(foo↑)↑` is meaningless.
- `foo(foo↑)` will match any of the following terms: `X`, `foo(X)`.
- `foo(foo↑)↓` will match any of the following terms: `foo(X)`. ⊗

4.4. A formal definition

We present here a formal definition of the '↓'-annotation.

In this section, we will think of terms as trees, that is, as partial functions from the set of all possible paths (i.e. finite and infinite sequences of natural numbers) to a co-domain $C_\perp$, where C is the set of node-labels. If the function is not defined for a given argument, we will take its value to be $\perp$. Then terms are trees over the co-domain of function symbols and variables. [See [3], though only elementary definitions are used here.]

We can now define *annotations* to be simply trees over the domain $\{\texttt{true}\}_\perp$, with the interpretation that the (node specified by the) path l is annotated by p iff $p(l) = \texttt{true}$. However, we would also like to insist that if a term is annotated then all its super-terms are also annotated. Hence this definition:

Definition 1: An **↓-annotation** is a tree $p : N^\omega \rightarrow \{\texttt{true}\}_\perp$ such that

$$\forall l.p(l) = \texttt{true} \Rightarrow \forall l' \sqsubseteq l.p(l') = \texttt{true}$$

where $\sqsubseteq$ is the 'is-a-prefix-of' relation between sequences. ⊗

We will represent by $\perp$ the annotation which is undefined everywhere.

In order to say that a term t is annotated by p, we must ensure that the annotated node exists. Hence:

---

[10] Note, on the contrary, that '↓' unifications fail iff the corresponding normal unifications fail.

Definition 2: An annotation p is **applicable** to a term t iff $\forall l.p(l) = \texttt{true} \Rightarrow t(l) \neq \bot.\otimes$

We would now like to give the semantics of an annotated term. The annotations serve to restrict the set of terms with which the annotated term can unify. Consider a term s annotated by p and a term t. First we would like to express the notion of a most general unifier for s and t which ignores all the annotated nodes in s and the corresponding nodes in t, if they exist.

Definition 3: The p-restricted **mgu** of two terms s and t, denoted by $\mathbf{mgu}_r(p, s, t)$, where p is an annotation applicable to s, is the most general substitution θ such that

$$\forall l.\, p[l] \neq \texttt{true} \Rightarrow (\theta(s))[l] = (\theta(t))[l]$$

where by $\theta(s)$ we mean the term obtained by applying substitution θ to the term s. $\otimes$

Note 1:

$$\mathbf{mgu}(s, t) = \mathbf{mgu}_r(\bot, s, t)$$

According to the intuitive meaning of '↓', we must ensure that all annotated terms unify against non-variable terms. This means that when unifying a term s against t where s' an annotated sub-term of s is to be unified against t' the corresponding sub-term of t, we cannot proceed until and unless t' is instantiated. Therefore if t' is instantiated, we can ignore the annotation on s'. So a simple strategy for finding the most general unifier of the term s annotated with p and the term t is to start by unifying all the sub-terms of s which are not ↓-protected by p against the corresponding sub-terms of t, i.e. by computing $\mathbf{mgu}_r(p, s, t)$. If this leads to instantiating a variable in t which has a corresponding sub-term in s that is ↓-protected, then we can remove this annotation and start again. The process terminates when there are no more '↓' annotations left to remove. Hence:

Definition 4: The $\mathbf{mgu}_\downarrow$ of two terms s, t with p an annotation applicable to s, (notated by $\mathbf{mgu}_\downarrow(p, s, t)$) exists whenever there exists $n \geq 0$ such that $T^n(p) = \bot$ where T is a transformation on annotations given by

$$\forall l.T(p)(l) = \begin{cases} \texttt{true} & p(l) = \texttt{true} \wedge \mathbf{mgu}_r(p, s, t)(s[l]) \in V \\ \bot & \textit{otherwise} \end{cases}$$

When it exists, $\mathbf{mgu}_\downarrow(p, s, t) = \mathbf{mgu}_r(T^n(p), s, t) = \mathbf{mgu}_r(\bot, s, t) = \mathbf{mgu}(s, t).\otimes$

In [14] we give a different definition by adding, to a conventional transition system for computing most general unifiers, a single transition for ↓-decorated terms.

4.5. Simulating '?' with '↓' and '↑'

In this section we show that given '↓' and '↑' it is possible to simulate '?'-unification by presenting a CP[↓, ↑, |] program which does that (Table 2). A query `unify(X,Y)` where the terms `X` and `Y` could be '?'-annotated will suspend iff the '?'-unification of the terms `X` and `Y` will suspend. It will succeed iff '?'-unification succeeds, and will bind the variables in the two terms to whatever value '?'-unification would have bound them too. It fails iff '?'-unification fails.

```
/* Program for ?-unification. (cf Table 1)

The predicate =/2 represents normal (i.e. '?'-free) unification. We assume
   this unification respects the occurs check. In what follows it is used just
   to unify a variable against a term. It may be defined as if by Clause 0:
  0.*/      X=X.


/* In Clauses 1 and 1', we need to put X=Var in the guard and not in the
   body so as to avoid incorrect failure due to occurs-check for goals such as
   unify(?(X), X) for which Clause 3 should succeed. Note that Clauses
   1 and 1' work also to unify X? against Var. (We do not need to check
   that X is not of the form Term? because of properties of ?-unification.)
  1.*/      unify(X, Var↑):- X = Var | true.
/*1'.*/      unify(Var↑, X):- X = Var | true.


/* When unifying ?(X) against a functional term T, wait until X is in-
   stantiated before proceeding.
  2.*/      unify(?(X↓)↓, T↓):- unify(X, T).
  2'.*/      unify(T↓, ?(X↓)↓):- unify(T, X).


/* Unifying lists.
In Clause 3, we need to put unify(X,R) in the guard so as to maintain left
   to right order of evaluation. If the arguments to a functional term are to
   be unified in parallel, then unify(X,R) should be put in the body.
  3.*/      unify([X | Y]↓, [R | S]↓):- unify(X, R) | unify(Y, S).
/*4.*/      unify(nil↓, nil↓).


/* Unifying functional terms.
Use Prolog's =../2 to convert the term into a list. (Constants are 0-ary
   functions.)
  5.*/      unify(X↓, Y↓):-
                  X ≠ ?(P), X =..  Xlist, Y ≠ ?(Q), Y=..Ylist |
                  unify(Xlist, Ylist).


/* Unifying X with X? or X? with X? should succeed immediately:
  6.*/      unify(?((X↓ a)↑), ?((X↓ a)↑)).
/*7.*/      unify(?((X↓ a)↑), (X↓ a)↑).
/*7'.*/      unify((X↓ a)↑, ?((X↓ a)↑)).
```

Table 2: A **CP**[↓, ↑, |] program for '?'-unification.

This program is almost a direct translation of the **Prolog** program given in [17] which purported to be the semantics of '?'-unification. It differs from it in distinguishing between suspension and failure, in avoiding unnecessary sequential over-commitment and in handling the issues raised in Section 3.

Specifically, it preserves the left-to-right order of evaluation, allows `unify(X?,X)` (and its symmetric counterpart) to succeed, allows `unify(X?,X?)` to succeed and causes `unify(X?,Y?)` to suspend until the principal functors of `X` and `Y` are known and then to recursively unify `X` with `Y`. The other variants of '?'-unification discussed in [11] and [12] can similarly be programmed in $\mathbf{CP}[\downarrow, \uparrow, |]$.

Note that in this program we choose to write the term `X?` in the more conventional **Prolog** notation `?(X)` (strictly, `'?'(X)`).

We also use a '$\downarrow$ /2' annotation (Clauses $2, 3, 3'$). This annotation is closely related to the unary '$\downarrow$' annotation ('$\downarrow$ /1') we have hitherto been using and its semantics is fully specified in [15]. Suffice to note here that the effect of the annotations in the head of Clause 2 is to ensure that it will unify only against goals of the form `unify(?(Var), ?(Var))` where `Var` is a variable: goals of the form `unify(?(Var), ?(Var1)` will *suspend* until some other process instantiates `Var` to `Var1` (or vice versa). Because of the '$\uparrow$' annotations the clause cannot be used if `Var` is a functional term.

Similarly only goals of the form `unify(?(Var), Var)` can succeed against the head of Clause 3.

5. Failure in Concurrent Prolog

The semantics of **Concurrent Prolog** can be given by specifying which SLD-derivations are *admissible*, given a query and a program. Because of its don't care nondeterminism, committing can only be locally angelic, i.e it chooses values for its free variables such that its guard executes successfully, but its body may still fail for the chosen values. Moreover, if a goal a occurs in a system of goals a, b, c (say), the values committed by a successful guard for a clause for a may cause b (and hence the whole guard system) to fail, even though *there exists* value(s) for the variables common to a and b for which they both succeed.

Example 12: Consider the program:

```
p(a).
p(b).
q(a).
q(c).
r ← q(X), p(X) | true.
```

The query `?-r.` may fail. For example, the `q(X)` goal in the guard for `r` may commit with the binding $\{X \mapsto c\}$. The goal `p(c)` will then fail so that the guard system fails, and the query `?-r.` fails even though there *is* a value of `X` which would cause the query to succeed. $\otimes$

We contend that the reason $\mathbf{CP}[\downarrow, |]$ is not a logic programming language is because it does not distinguish failed admissible SLD-derivations from admissible SLD-refutations,

i.e. it is possible for a $\mathbf{CP}[\downarrow, |]$ query to terminate in *failure* even though there may be a successful refutation of the query, given the '$\downarrow$' control annotations. Thus $\mathbf{CP}[\downarrow, |]$ is not even *potentially complete*. For example, **Prolog**, which is potentially complete, will never *terminate* in failure until and unless there is no solution for the given query, given the sequential search process (which is also enough to guarantee that there is no solution given the pure clauses, i.e. no control structure assumed). This implies that the same query may have a succesful execution sequence, as well as a failed exceution sequence: in pure Horn logic, as in Prolog, these two sets (denoted by *SS* and *FF*) are necessarily disjoint.

This means:

- Validity of unannotated axioms is not sufficient for partial correctness: a given query will assuredly succeed (or loop) *only* if all finite admissible SLD-derivations are refutations, which is a very strong condition.
- There can be no notion of negation-as-failure even with respect to admissible deriyations. (i.e. at best we can hope that negation means that no admissible derivation for the query is a refutation... even that is not compatible with '|'.)
- Many Horn logic axiom definitions cannot be used in $\mathbf{CP}[\downarrow, |]$, in the sense that no version of these axioms, suitably annotated to form a legal $\mathbf{CP}[\downarrow, |]$ program, has a meaning that is compatible with their logical semantics.

  As an illustration, no version of the axioms:

  ```
  p(X,Y):- t(X,Y).
  p(X,Y):- p(X,Z), p(Z,Y).
  ```

  annotated with '$\downarrow$' and '|' annotations can be guaranteed to work correctly (i.e. compute the transitive closure of p/2) for an arbitrary (Horn) definition of p/2).

For example, consider the program:

```
/*1.*/      p(0,1).
/*2.*/      p(0,2).
/*3.*/      p(1,3).
/*4.*/      t(X↓, Y↓):- p(X,Y)| true.
/*5.*/      t(X↓, Y↓):- p(X,Z), t(Z,Y)|true.
```

The query `?-t(0,3).` may fail because in Clause 5, the goal `p(X,Z)` may commit to Clause 2 (`p(0,2).`).

Perhaps one can look for a logical characterisation of the *strong success set* (that is the set of all those ground atoms on which the given $\mathbf{CP}[|]$ will always succeed). For sure $SSS_P \subseteq SS_P$ and the degree to which it is smaller reflects the constraints on existential search because of the don't care commit. For example for the above program, SSS_P = {p(0,1), p(0,2), p(1,3), t(0,1), t(0,2), t(1,3)} whereas $SS_P = SSS_P \cup$ {t(0,3)}.

6. Alternate commit operations

There are two simple alternate interpretations for 'commit' which distinguish between

successful and unsuccesful admissible derivations.

6.1. The parallel don't-know commit or the '&' annotation

If '|' can be thought of as the 'don't care' commit, then '&' is the 'don't know' commit. It interpretes 'commit' as 'publish bindings'. It does *not* delete *OR*-siblings, but instead continues to follow them, allowing *multiple* commits of guard systems. Each commit is to a different copy of the rest of the environment. In effect whereas `a:-g | b.` extends *some* admissible refutation of g by an admissible refutation for b to return *one* (selected from possibly many) refutation for a, `a:-g & b.` extends *every* admissible refutation of g by an admissible refutation for b and returns *all* of them as refutations for a, thereby avoiding a local commitment to one refutation of a. Therefore, executing a query ends in failure only if *all* admissible derivations are finite and failing, just as for **Prolog**. In fact, we show in [14] that negation as failure is *sound* for $\mathbf{CP}[\downarrow, \&]$ so that for every program $SS \cap FF = \emptyset$.

A completely formal description of & is given in [14]. A partial correctness semantics is given in [16]. Here we simply note that the language $\mathbf{CP}[\&]$, that is the language in which every clause body contains an & commit operator, but no clause head is $\downarrow$-annotated, has the same operational interpretation (and hence abstract semantics) as the corresponding pure logic program. Hence it is possible to write every set of Horn axioms as $\mathbf{CP}[\downarrow, |, \&]$ programs, with the computed meaning being the same as the desired meaning.

6.2. The sequential don't-know commit or the '\' annotation

Interpretes 'commit' as 'publish bindings and *freeze OR*-siblings'. Here *one* admissible SLD-derivation is followed until it terminates. If it terminates in success, nothing is done. Failure induces backtracking. For partial correctness, the exact backtracking scheme used (chronological, dependency-directed) is not important, as long as it can be guaranteed that the system will not terminate in failure as long as even one admissible SLD-derivation path has not been pursued. That is, no *finite* SLD-derivation is admissible unless it is a refutation or else all admissible derivations are finite and failing. [11]

6.3. Interrelationships between various interpretations of commit

Lemma 4: Given a **CP** program and a query, the set of possible (successful) answers to a query is the same if the commit operator is interpreted as '|', '&' uniformly throughout the program. That is, the success sets for corresponding programs in the two languages are the same.

This is proved in [14]. It is easy to see that the intention of '&' is to cut down the set *FF*, and not the set *SS*. With the '\' commit operator the issue is more complex because some of the successful answers may be unreachable because of intervening infinitary paths (as in the case of **Prolog**).

The elementary relationships between interpreting the commit operator as '|', '&' and '\' uniformly in a program are as follows:

[11] Note that pragmatically, the backtracking strategy may be quite important and will determine the character of the language.

CP[↓,|] Execution of a query always terminates in success *iff* all admissible derivations are finite and refutations.

CP[↓,\] Execution terminates in failure *only if* all admissible derivations are finite and none is a refutation. If a query terminates in failure for a **CP**[↓,\], it will always terminate in failure for the corresponding **CP**[↓,|] program.

CP[↓,&] Execution of a query always terminates in success *iff* there is an admissible refutation. (This is true only if we assume OR-fairness and that execution stops as soon as one answer is found for the top-level query, or equivalently if all top level queries are of the form ?-$a_1, \ldots a_n$|true.)

6.4. Why have don't know commits?

As in **Prolog**, allowing implicit search allows the user to write powerful but inefficient programs. It is our contention that one of the novel aspects of Horn logic programming is this capability to specify an implicit search: it is this capability which allows simple logic specifications of complex operations. Thus, for example, much of the work done on constraint-based computation [21] can be carried out in an *AND*-parallel, *OR*-backtracking framework (i.e. in **CP**[↓,\]) or in an *AND*-parallel, *OR*-parallel framework (i.e. in **CP**[↓,&]). As noted in [18], **CP**[?,|] is a poor framework for such computations.

Example 13: Consider the following **CP**[↓,|,&] program

```
prod(X↓,Y↓,Z):- Z is X*Y | true.
prod(X↓, Y, Z↓):- X =/= 0 | Y is Z/X.
prod(X, Y↓, Z↓):- Y =/= 0 | X is Z/Y.
prod(X, Y, Z↓):- less(X,Z), less(Y, Z), Z := X*Y | true.
Z := X↓*Y↓:- Z is X*Y.
```

Assume that `less/2` is defined as a generator as if by the collection of clauses `less(i#,j#):- true & true.` for each value of i# and j# such that $i\# < j\#$ and that `is/2` is a primitive for evaluating arithmetic operations. This program can then 'solve' `?- prod(X,X,16).` to give `X=4` by generating and testing possible values for `X`. For a finer control on the generation process, a *sequential* OR may be used. ⊗

More examples of this kind are to be found in the author's thesis proposal ([15]).

6.5. The role of don't-care commit

In this language the '|' should be used just as the cut is used in **Prolog**: to signal determinate solutions or to select one of many possible answers when it is known that any one of them will suffice. There is definitely a place for the '|' in concurrent Horn logic programming languages; but we balk from investing it with crucial importance as has been done in other CLP languages. [12]

[12] In fact a commonly held belief in concurrent Horn logic programming circles was that 'committed choice non-determinism is the crucial feature that makes stream and-parallelism implementable'. Through the don't know commits we hope to have shown that it is not necessary to give up stream and-parallelism.

```
X = X:- X ≠ Y$.
unify(X↓, Y):- X =Y | true.

cp(G↓):-
      clause((Goal :- Guard | Body)),
      unify(Goal, G),
      execute-all(Guard) | execute-all(Body).

cp(G↓):-
      clause((Goal :- Guard & Body)),
      unify(Goal, G),
      execute-all(Guard) & execute-all(Body).
cp( true↓):- true | true.

execute-all(','(One,Rest)↓):-
      true | cp(One), execute-all(Rest).
execute-all(true↓):- true | true.
```

Table 3: A CP[↓, |, &] meta-interpreter.

7. A meta-interpreter in CP[↓, |, &]

Finally, to show the expressive power of the language CP[↓, |, &] we present here a simple meta-interpreter. Note that it is not possible to give as simple an interpreter for CP[↓, |] in CP[↓, |]. This interpreter can be extended to give an interpreter for CP[↓, ↑, |, &] in CP[↓, ↑, |, &].

7.1. The meta-interpreter

We define a predicate cp/1 which takes as input a goal and solves it. The user-program is added in as clauses of the form:

clause((Head:-Guard % Body)):- true & true. where:

- All instances of '↓' in Head are replaced by '$', which will be regarded as a unary post-fix function symbol,
- Guard and Body are sequences of goals of the form '$\{g_1, \ldots g_n\}$'.

In addition we also have the following axioms (which can be added automatically) for every functor f/n, ($n \geq 0$) in the user-program, the axioms:

```
f(X1,...Xn)↓ = f(Y1,...Yn)$:- X1 = Y1,...Y1=Yn | true.
f(X1,...Xn)$ = f(Y1,...Yn)↓:- X1 = Y1,...Y1=Yn | true.
```

(Note that we could give an alternative definition using =../2 as in Table 2.)

Then the interpreter is given by the program in Table 3.

The 'built-in' predicate $\neq /2$ used in the program has the following semantics: a query $X \neq Y$ suspends until the principle functors of X and Y are known. It then succeeds

iff the functors are not identical. Note that in the above program whenever a $\neq /2$ goal executes the principal functors of its two arguments are known.

7.1.1. Why there is no simple Concurrent Prolog meta-interpreter

The reason one cannot write as simple an interpreter for CP[↓, |] is that in CP[↓, |] one cannot add the user-program as a list of `clause/1` clauses to the interpreter as we have done because then a call to `clause((Head:-Guard|Body))` would succeed at most once, selecting some `clause/1` clause at random whereas we would like to select *all* clauses in parallel and execute their guards concurrently, i.e. we *don't know* which clause we want. Hence we are forced to represent the program explicitly as an argument to the interpreter, as a list of clauses and that makes any meta-interpreter very messy.

Note that in [9], Meirowsky gives just such a meta-interpreter for **Flat Concurrent Prolog**. This meta-interpreter makes use of a `clause/2` predicate which is assumed to be pre-defined such that a goal `clause(Goal, Clauses)` succeeds iff `Goal` is instantiated to a goal and returns in `Clauses` a *list* of all the candidate clauses in the user program for `Goal`.

This is precisely the point we made earlier: because there is no don't know determinism in **Concurrent Prolog**, all search has to be programmed, making the language much more procedural rather than declarative.

7.2. An interpreter for Concurrent Prolog?

Given that we can specify the semantics of '?'-unification in **CP**[↓, ↑, |] it may seem as if we can write a simple interpreter for **Concurrent Prolog** in **CP**[↓, ↑, |, &] along the lines of the program given in Section 7. We would try to do this by using the same program, but with the definition of `unify/2` given in Table 2 instead of the definition of `unify/2` given above. Such an attempt will not succeed because in **Concurrent Prolog**, in effect, '?'-unification occurs at *two* places. One place is the unification of a goal with the head of a clause, and this we can take care of by using the alternate definition of `unify/2`. The other place where unification happens in **Concurrent Prolog** is when an *OR*-parallel guard system commits bindings to the *AND*-siblings of the parent goal. In **Concurrent Prolog**, as discussed in Section 3.3.2, this can lead to some previously active goals and guard systems being frozen. This can never happen in any CP language, where, by definition, ordinary unification is done at commit time.

8. Acknowledgements

Many thanks to Larry Rudolph and Steve Brookes for helpful discussions. I am also thankful to Tony Kusalik, Ehud Shapiro, Jacob Levy and other correspondents of the Prolog Digest for responding to a discussion on some of the issues raised herein and to Steve Gregory for his comments on an earlier version of this paper.

8.1. Historical Note

This paper was first written up in May 1985 and circulated privately in the concurrent

logic programming community. Some of this material has earlier appeared in the **Prolog Digest**, and was a subject of much discussion. It turned out that some of these points had been made earlier by Tony Kusalik privately in discussions with Ehud Shapiro and the ICOT Group. Meanwhile K. Ueda from ICOT also had written up his critique of **Concurrent Prolog** and this came out in June 1985 as [22]. This present version, which is the first to be published, is essentially a revision of the paper circulated in May 1985 with some additional expositions, examples and elaborations.

REFERENCES

[1] Apt, K.R., van Emden, M.H., 'Contributions to the theory of logic programming', JACM, vol. 29, No.3, July 1982, pp 841-862.

[2] Clark, K.L., Gregory, S., 'PARLOG: parallel programming in logic', Res report DOC 84/4, Imperial College, (revised June 1985).

[3] Courcelle, B., 'Fundamental properties of infinite trees', Theoretical Computer Science, 25 (1983) 95-169.

[4] Hellerstein, L., Shapiro, E.Y. 'The MAXFLOW experience', International Symposium on Logic Programimng, Atlantic City, New Jersey, February, 1984.

[5] Jaffar, J., Lassez, J.-L., Maher, M.J. 'A theory of complete logic progams with equality', Technical Report 43, June, 1984, Department of Computer Science, Monash University.

[6] Jones, N.D., Mycroft, A., 'Stepwise development of operational and denotational semantics for Prolog', Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City.

[7] Lassez, J.-L., Maher, M.J. 'Closure and fairness in the semantics of programming logic', *Theoretical Computer Science* 29 (1984) 167-184.

[8] McCarthy, J. 'A basis for a mathematical theory of computation', in *Computer Programming and Formal Systems* ed. Braffort, P. and Hirschberg, D., North-Holland Amsterdam, 1963.

[9] Mierowsky, C. 'Design and Implementation of Flat Concurrent Prolog', CS84-21, Weizmann Institute of Science, December, 1984.

[10] **Prolog Digest** Volume 3, Issue 6, 28 Feb 1985.

[11] **Prolog Digest** Volume 3, Issue 7, 5 Mar 1985.

[12] **Prolog Digest** Volume 3, Issue 8, 11 Mar 1985.

[13] **Prolog Digest** Volume 3, Issue 9, 18 March 1985.

[14] Saraswat, V.A., 'An operational semantics for Concurrent Prolog', in preparation.

[15] Saraswat, V.A., 'Concurrent logic programming languages', Thesis proposal, Computer Science department, Carnegie-Mellon University, November 1985.

[16] Saraswat, V.A. 'Partial correctness semantics for **CP**$[\downarrow, |, \&]$', *Proceedings of the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, December 1985, Springer-Verlag LNCS 206.

[17] Shapiro, Ehud Y., 'A subset of Concurrent Prolog and its interpreter', CS83-06, Weizmann Institute technical report.

[18] Shapiro, E.Y., Takeuchi, A., 'Object oriented programming in Concurrent Prolog', *New Generation Computing*, 1 (1983) 25-48.

[19] Shapiro, E. Y., 'Systems programming in Concurrent Prolog', *POPL*, 1984.

[20] Shapiro, E. Y., 'Systolic programming: a paradigm of parallel processing', *Proceedings of the Fifth Generation Computer Systems Conference*, 1984.

[21] Steele, G.L., 'The definition and implementation of a programming language based on Constraints', PhD Thesis, EECS Department, M.I.T., August, 1980.

[22] Ueda, K. 'Concurrent Prolog re-examined', ICOT TR-102 Draft, June 1985.

[23] Ueda, K., 'Guarded Horn Clauses', ICOT Technical report TR-103, June 1985.