ICOT Technical Report: TR-102

TR-102

Concurrent Prolog Re-Examined

by
Kazunori Ueda
(NEC Corporation)

November, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

CONCURRENT PROLOG RE-EXAMINED

Kazunori Ueda

C&C Systems Research Laboratories, NEC Corporation
(until May 1985)
and
Institute for New Generation Computer Technology
(since June 1985)

First version: December 1984
Last revision: November 1985

ABSTRACT

The language rules of Concurrent Prolog in Shapiro's original paper is re-examined. The main point is that some sequentiality must be assumed for unification in order to reasonably define the semantics of unification and commitment. This means that some 'logical' transformation of a program clause may change its semantics. Another point is that there are semantical problems in the semantics of multiple environments and a commitment operation.

1. INTRODUCTION

ICOT (Institute for new generation COmputer Technology) is now designing Kernel Language Version 1 (KL1) [Furukawa et al. 84] for a parallel inference machine (PIM) which is to be developed in the intermediate stage of its Fifth Generation Computer Systems (FGCS) project.

As the starting point towards KL1, Shapiro's Concurrent Prolog [Shapiro 83] was chosen for investigation. Concurrent Prolog was chosen because its language rules were so concise and it looked expressive enough. Our experience then showed that Concurrent Prolog was fairly expressive. However, we have not examined whether it is really concise; it still has only very informal semantics except for the operational semantics in [Hirata 84]. A simple language rule expressed in a natural language may be formalized into a set of quite complex rules. Therefore, we must examine every subtle point of Concurrent Prolog and make necessary clarifications or modifications.

Our design goals for KL1 should include the following general requirements to be satisfied as the specification of a parallel programming language.

(a) The semantics should be well-defined and clear in a fully parallel execution model.
(b) The specification should not assume unnecessary sequentiality, because it is against (a) above and it discourages both programmers and implementors from exploiting parallelism.

KL1 is not a language only for communicating sequential inference machines but also for parallel inference machines. Therefore, the best way to develop KL1 is not to exploit parallelism from sequential Prolog but to find appropriate restrictions on the full parallelism inherent in a set of Horn clauses, assuming that KL1 should be a logic programming language.

However, we are not so accustomed to parallel programming or languages for parallel programming. Our short experience with programming in Concurrent Prolog programming has shown that parallel programming is not so

easy even for those experienced with sequential programming. It is hard to expect what is implied by the interaction of complex language rules. Therefore,

(c) The language rules must be kept simple.

Moreover, the language specification should satisfy the following general principles.

(d) There should be no language features that are not implementable or are very hard to implement. If such rules were to exist, the blame should be laid on the language specification.
(e) There should be as few rules as possible whose violation cannot be easily detected at compile time or at run time. These would promote erroneous programs to circulate.

In the following, the language Concurrent Prolog will be re-examined in the light of the above principles. We will regard [Shapiro 83] as the defining document of Concurrent Prolog, since it is the original and the most detailed text. The fundamental method of examining a language defined informally is to examine every defining sentence thoroughly. However, here we will not make comments sentence by sentence: Our purpose is not to criticize the sentences defining Concurrent Prolog, but to try to obtain a correct interpretation of this language. For this purpose, we have examined other documents on Concurrent Prolog by Shapiro (e.g., [Shapiro 84] and [Shapiro and Takeuchi 83]) also, but no significant difference from the original document or new information which might help us to reach the correct interpretation was found as for the materials discussed in this paper.

2. THE DEFINITION OF CONCURRENT PROLOG

This chapter introduces the syntax and the semantics of Concurrent Prolog as described in the original paper [Shapiro 83]. We will quote important paragraphs from that paper and number them for later references. Note that although the language defined in [Shapiro 83] was first called "a subset of Concurrent Prolog", it has simply been called "Concurrent Prolog" among the community ever since.

2.1 Syntax

[1] (Section 3.1) A Concurrent Prolog program is a finite set of guarded clauses. A guarded clause is a universally quantified axiom of the form

A :- G1, G2, ..., Gm | B1, B2, ..., Bn. m,n>=0

where the G's and the B's are atomic goals. The G's are called the guard of the clause and B's are called its body. When the guard is empty the commit operator is omitted. The clause may contain variable marked "read-only".

[2] (3.1) The commit operator generalizes and cleans sequential Prolog's cut. Declaratively, it reads like a conjunction: A is implied by the G's and the B's. ...

2.2 Semantics

[2] (Section 3.1) ... Operationally, a guarded clause functions similarly

to an alternative in a guarded-command. It can be used to reduce process A1 to a system B if A is unifiable with A1 and, following the unification, the system G is invoked and terminates successfully.

[3] (3.1) The unification of a read-only term X? with a term Y is defined as follows. If Y is non-variable the 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?.

...

[4] (3.1) This definition of unification implies that being "read- only" is not an inherited property, i.e. variables that occur in a read-only term are not necessarily read-only. Stating it differently, the scope of a read-only annotation is only the principal functor of a term, but not its arguments. ...

[5] (3.1) The definition of unification also implies that the success of a unification may be time-dependent: a unification that fails now, due to violation of a read-only constraint, may succeed later, after the principal functor of a shared read-only variable is determined by another process, in which this variable does not occur as read-only.

[6] (3.2) The execution of a Concurrent Prolog system S, running a program P, can be described informally as follows. Each process A in S tries asynchronously to reduce itself to other processes, using the clauses in P. A process A can reduce itself by finding a clause A1 :- G | B whose head A1 unifies with A and whose guard system G terminates following that unification. The system S terminates when it is empty. It may become empty only if some of the clauses in P have empty bodies.

[7] (3.2) The computation of a Concurrent Prolog program gives rise to a hierarchy of systems. Each process may invoke several guard systems, in an attempt to find a reducing clause, and the computation of these guard systems in turn may invoke other systems. The communication between these systems is governed by the commitment mechanism. Subsystems spawned by a process A have access only to variables that occur in A. As long as a process A does not commit to a reducing clause, these subsystems can access only read-only variables in A, and all binding 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. Upon commitment to a clause A1 :- 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.

[8] (3.2) A more detailed description of a distributed Concurrent Prolog interpreter uses three kinds of processes: an and-dispatcher, an or-dispatcher, and a unifier; these processes should not be confused with the Concurrent Prolog processes themselves, which are unit goals.

[9] (3.2) The computation begins with a system S of Concurrent Prolog processes, and progresses via indeterminate process reduction. After an and-dispatcher is invoked with S, the computation proceeds as follows:

- o An and-dispatcher, invoked with a system S, spawns an or-dispatcher for every Concurrent Prolog processes A in S, and waits for all its child or-dispatchers to report success. When they do, it reports success and terminates.
- o An or-dispatcher, invoked with a Concurrent Prolog process A, operates as follows. For every clause A1 :- G | B, whose head is potentially unifiable with A, it invokes a unifier with A and the clause A1 :- G | B. Following that the or-dispatcher waits for any of the unifiers to report success. When one such report arrives, the or-dispatcher reports success to its parent and-dispatcher and terminates.
- o A unifier, invoked with a Concurrent Prolog process A and a guarded-

clause A1 :- G | B, operates as follows. It attempts to unify A with A1, storing bindings made to non read-only variables in A on private storage. If and when successful, it invokes an and-dispatcher with G, and waits for it to report success. When this report arrives, the unifier attempts to commit, as explained below. If the commitment completed successfully it reports success, but in either case it terminates.

[10] (3.2) At most one unifier spawned by an or-dispatcher may commit.

...

[11] (3.2) To commit, a unifier first has to gain a permission to do so. The mutual exclusion algorithm must guarantee that if at least one unifier wants to commit, then exactly one unifier will be given permission to do so. After gaining such a permission, the unifier attempts to unify the local copies of its variables against their corresponding global copies. If successful, then the commitment completes successfully.

[12] (3.2) ... Another useful optimization is the deletion of brother unifiers, once the first such process is ready to commit.

[13] (3.2) 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.

...

[14] (3.2) Since a unification that currently fails may succeed later, the phrase "attempts to unify" in the description of a unifier should be interpreted as a continuous activity, which terminates only upon success. This can be implemented using a busy-waiting strategy, but several optimizations can be incorporated. ...

We correct a small error in Paragraph [9] here. An or-dispatcher should report a set of processes B to its parent and-dispatcher instead of the simple message "success". The and-dispatcher must recognize these processes as newly created processes which replace the original one, and must spawn or-dispatchers for them. The and-dispatcher reports success and terminates when S is reduced to an empty set of goals. Alternatively, an or-dispatcher may invoke a new and-dispatcher for the body of a clause whose corresponding unifier has reported success, and let the report of this and-dispatcher be the report of the or-dispatcher in question.

2.3 Parallel Programming in Concurrent Prolog

[15] (3.1) A system of processes corresponds to a conjunctive goal, and a unit goal to a process. The state of a system is the union of the states of its processes, where the state of a process is the value of its arguments. And-parallelism--solving several goals simultaneously--provides the system with concurrency. Or-parallelism--attempting to solve a goal in several ways simultaneously--provides each process with the ability to perform indeterminate actions. Variables shared between goals serve as the process communication mechanism; and the synchronization of processes in a system is done by denoting which processes can "write" on a shared variable, i.e. unify it with a non-variable term, and which processes can only "read" the content of a shared variable X, i.e. can unify X with a non-variable term T only after X's principal functor is determined, possibly by another process. ...

3. MULTIPLE ENVIRONMENTS AND A COMMITMENT OPERATION

A commitment operation as described in Paragraphs [11] and [13] is a process rather than an event. It is preceded by the permission, and it completes when the unification of local and global copies of variables has terminated successfully. It is not explicitly specified when the commitment starts, but it should be some time after the permission and not later than the start of the unification.

The most controversial issue is the nature of the permission. The second sentence of Paragraph [11] says that the mutual exclusion algorithm must guarantee that if at least one unifier wants to commit, then exactly one unifier will be given permission to do so. However, it is not clear whether

(1) this permission is revoked when the unification of local and global copies of variables does not succeed, thus providing the other clauses with the opportunity of commitment, or
(2) this permission is eternal, i.e., the other clauses can no longer attempt a commitment operation once some clause has gained a permission.

The failure of the unification can happen when a global variable is further instantiated by some other goals after its local copy is created, but Paragraphs [7], [9], and [11] defines only the successful case. If the permission is never revoked, failure of the unification means the failure of its grandparent and-dispatcher, i.e., the parent of its parent or-dispatcher. If the permission can be revoked, the unification must be performed in a way in which the intermediate result of the unification is invisible from other processes. This is because this unification may eventually fail, in which case the other clauses must retain the possibility of commitment.

Paragraphs [5] and [14] say that unification is a continual activity which terminates only upon success. So one interpretation could be that the commitment operation is also a continual activity which terminates only upon success of the unification involved in it. This suggests that the permission need not be revoked. However, this interpretation is unfortunately inconsistent with the description in Paragraph [9] that the unifier terminates whether or not the commitment completed successfully, though Paragraph [9] is problematic in that it does not say at all how the unifier can terminate when the commitment has not completed successfully.

The complete lack of the description of a looking operation, which is shown below to be necessary when the permission can be revoked, suggests that the permission is of an eternal nature. However, the interpreter shown in [Shapiro 83] adopts the opposite interpretation. It seems impossible to derive a correct answer from the original description; the better way should be to make a thorough examination of the merits and demerits of the both alternatives.

3.1. The First Alternative: Permission Is Revocable

We first assume that the permission of commitment is revoked when the unification of local and global copies of variables does not succeed. When the permission is revoked, the result of unification must be kept invisible from other goals. Since it is only upon success of the unification that the revocation of the permission is known to be unnecessary, we must always perform the unification of local and global information in a way in which its partial result is invisible from other goals. The problem is how to implement a commitment operation which meets the above requirement.

The only possible solution would be to perform the following operations in the given order:

(a) Lock (at least) all the relevant global data, that is, all variables appearing in goal arguments for which local copies have been made.
(b) Try to export local bindings by unifying local copies with its global counterpart. This could start before Step (a) is finished, as long as no bindings are made to unlocked global variables.
(c) (c-1) If the unification is successful, then simply unlock the global data locked in Step (a).
(c-2) If the unification is unsuccessful, then undo all the bindings made in Step (b), and then unlock the global data locked in Step (a). The unlocking can start before undoing is finished, as long as no global variable is unlocked without being unbound. Independently of these operations, return the permission of commitment.

The problem lies in the locking operation in Step (a). The simplest locking scheme would be to lock the whole memory whenever commitment is attempted, but this is definitely unacceptable because it serializes all commitment operations. If we do not want to lose parallelism, we must minimize the locked area.

The smallest unit of locking is a single variable. However, variable-by-variable locking is not easy when we have to lock two or more variables in one commitment operation, as studied in the area of distributed databases and operating systems. Assume there are two clauses (say A and B) attempting to be selected and that both of them have to lock the variables X and Y. If Clause A tries to lock X first and Clause B tries to lock Y first, they may deadlock.

This deadlock problem can be easily resolved if we can order the variables. If we can give an invariant ordering to a set of variables to be locked, each clause has only to lock them in that order. However, it is hard to consider such an invariant ordering, because two variables may be unified at any time and after that unification they must have the same order. All the above considerations lead us to the conclusion that the first alternative is unacceptable.

One may think we could detect ununifiability of local and global information earlier than commitment. This is enabled by checking unifiability of the local and the global values of a variable whenever new global or local binding for that variable is created. However, this 'eager' checking never eliminates the unification upon commitment, and this unification must still satisfy all the requirements we stated above.

3.2. The Second Alternative: Permission Is Eternal

Let us then consider the other alternative that the permission of commitment is of an eternal nature. We no longer need locking operations because it is now impossible for the other clauses to export bindings later. Unification may be done just in a usual manner. This alternative increases the chance of failure of a program in which two or more conjunctive goals try to instantiate the same variable upon commitment. However, it does not cause so much inconvenience. In actual programs we have written, most predicates are effectively (possibly nondeterministic) functions each of which returns only one result. Such a result is usually prepared in a guard and exported upon commitment, but we usually receive it by a variable, in which case no failure can happen.

However, there still remains some semantical problems. Although it is unnecessary to return a permission of commitment once it is obtained, we have to define when a 'unifier' (in terms of Paragraph [9]) can attempt to gain such a permission. In other words, we have to define what kind of global information supplied by goal arguments must be respected when we regard the head unification and the execution of a guard as successful. Some information which may come later than the attempt of commitment could

be ignored, but other information which is evident to arrive 'early' must be considered for clause selection. Consider the following goal:

```
:- p(a).                                                    (3.1)
```

This goal says two things: call the unary predicate 'p' and set its argument to 'a'. The question is whether or not the argument setting must be completed before the call. This is not an absurd question; we are examining Concurrent Prolog as a truly parallel language. If the argument setting must precede the call, the above goal never fails when the predicate 'p' is defined as follows:

```
p(a).                                                       (3.2)
p(b).                                                       (3.3)
```

Only Clause (3.2) succeeds in head unification.

However, if the argument value is allowed to come arbitrarily late, the above goal may fail. Clause (3.3) may be tested earlier, and the value 'b' in the head may be recorded locally for later unification if its counterpart on the side of the goal has not arrived. In this case ununifiability of 'a' and 'b' will be detected after Clause (3.3) has gained a permission of commitment, and the original goal finally fails. To generalize, any clause can be selected for a given goal regardless of the goal and the head arguments as long as the guard succeeds, and hence can make the whole system fail. This should be extremely inconvenient: We cannot write a predicate intended to check argument values like 'p' above.

Therefore, at least any information specified textually in a goal must be considered for clause selection, i.e., any inconsistency with local information must be detected before attempting a commitment operation. This means that Clause (3.3) in the above example must detect inconsistency between 'a' and 'b' at head unification and must never create a local copy of its argument as long as the permission of commitment is not revocable.

Note that the above conclusion applies also to our first alternative on the semantics of commitment that the permission of commitment is revocable. For if we allow an immediate argument value to come arbitrarily late, it may come too late--after commitment has completed.

The question of allowed delay of information rather belongs to the semantics of the unification of Concurrent Prolog, and it will be discussed further in Chapter 4.

3.3. Access to Local/Global Information

Another problem which arises by allowing local and global copies of a variable is to which copy we must have access in each of the following cases:

(1) A variable in a goal is textually marked read-only.
(2) A variable in a goal is not textually marked read-only and its local copy has been made.
(3) A variable in a goal is not textually marked read-only and its local copy has not been made.

The first possibility for a goal variable is that it is marked read-only textually. Then candidate clauses must watch its global value. The global value must be watched because this is necessary for making suspended unification succeed and putting computation forward. For such a variable, local copies are not created (with the exceptional case shown below), since no bindings can be given directly to read-only variables in a goal from a clause head or a guard.

However, there also exists a rather pathological case where a local

copy must be created for a read-only variable. Consider the following example:

```
Call:       :- p(X?, X), X=a.                    (3.4)
Program:       p(a,  A) :- A=a | true.           (3.5)
```

The predicate '=' unifies its two arguments. The first argument may be instantiated in two ways:

(a) global instantiation by the goal 'X=a' running in parallel, and
(b) local instantiation by the goal 'A=a'.

In the second case, a local copy of X, which appears with (and without) read-only annotation, must be created. Note that the goal 'A=a' must locally instantiate the first argument: The goal textually specifies that its two arguments be identical (except for the annotation), so this identity must be respected for clause selection as we concluded in Section 3.2. This example illustrates also that suspension of unification due to a goal variable marked read-only may be resolved in two ways, globally and locally. Hence it is generally inadequate to wait only for global instantiation of a read-only variable; we have to implement multiple waits.

The second possibility for a variable contained in a goal is that it is not marked read-only textually and that some clause has created its local copy. In this case, that clause should see the local copy. Ignoring it and seeing only its global counterpart might suspend some unification which would otherwise succeed. However, it is not clear whether the clause should be allowed to see also the global value which may get instantiated after the local copy is created. Paragraph [7] seems to disallow it, but it is possible that the clause can solve its guard only by using the global value of some variable for which a local copy has been made.

The third possibility is that a variable contained in a goal is not textually marked read-only and that its local copy has not been created either. This is further divided into two subcases:

(a) The variable is uninstantiated when head unification starts.
(b) The variable has become non-variable or read-only when head unification starts.

For each case, there are three possible interpretations:

(1) The clause should watch its global instantiation.
(2) The clause can ignore it.
(3) The clause SHOULD ignore it.

Paragraph [7] seems to support Alternative (3) for Case (a) and Alternative (1) for Case (b).

Let us consider Case (a) first. Alternative (3) implies that the value of a variable in a goal which has become read-only or become non-variable AFTER the goal is called should be ignored. For instance, the following program

```
:- p(X), X=a.                                   (3.6)
   p(X) :- X?=a | true.                         (3.7)
```

should never succeed as long as p(X) is called before X=a is executed.

However, we cannot adopt the same alternative for Case (b). In this case, the value of the variable in the goal should not be ignored. Otherwise, the 'protected data' technique [Hellerstein and Shapiro 84] [Takeuchi and Furukawa 85] would not work correctly.

Typical use of read-only annotation is to attach it to the argument

variables of goals which consume (or decompose) the value of those variables. However, the protection against instantiation by the consumer can be achieved also by making the generator of a data structure protect its uninstantiated part, and 'protected data' means such uninstantiated but protected variables created by the generator. When we use this technique, read-only annotations do not appear textually in the consumer goal (except for the top level) but it is sent from the generator goal. If such dynamic protection should be ignored as Alternative (3) says, this technique could not be used.

Cases (a) and (b) can occur depending on the relative timing of the unification of a goal and a clause head, and the instantiation of a variable in the goal (by some other goal). Moreover, head unification is not an instantaneous operation, so it it is undesirable to assign the mutually exclusive behaviors (1) and (3) to these two cases. Alternative (1) or (2) should be a better choice for Case (a).

Our claim that Alternative (3) is undesirable could be understood and justified also from the following observation. If Clause (3.6) were given as

```
:- p(a).                                                    (3.8)
```

it should succeed because the textually specified argument value must not be ignored. This means that partial evaluation of 'X=a' in Clause (3.6) to get Clause (3.7) undesirably change the semantics of the program from suspension to success.

4. ATOMIC OPERATIONS IN UNIFICATION

The semantics of unification must clearly state what are atomic operations. Consider the following unification:

```
:- ..., X = f(a), ...                                       (4.1)
```

How this unification can be performed, assuming that X are uninstantiated? Possible solutions may be:

(1) Create a term f(a) (in any manner). Then set X to this term.
(2) Create a term f(A?) where A is a new variable. Then set X to this term, and in parallel with this set A to 'a'.
(3) Create a most general unary term with the principal functor 'f'. Call its argument A (this is not part of the operation). Then set X to this term, and in parallel with this set A to 'a'.

Alternative (1) regards the unification of a variable and a non-variable term as an indivisible operation. Alternative (2) tries to allow the principal functor and its arguments to be determined in parallel, but it protects non-variable arguments by read-only annotation. Alternative (3) states that the above unification can be done as if it were specified as follows:

```
:- ..., X = f(A), A = a, ...                                (4.2)
```

where A is a variable not appearing elsewhere.

Let us explain the differences among these alternatives in other words. Alternative (1) says that when a principal functor has been determined as the value of some variable, its textually specified arguments have also been determined. Alternative (2) says that the argument values may come later, but that the uninstantiated variables which appear in the transient state are protected. Alternative (3) says that the arguments may

come later and that the variables are not protected.

Note that for unification between two non-variable terms also, we can conceive three alternatives corresponding to the above ones.

The granularity of atomic operations is smallest in Alternative (3) and largest in Alternative (1). So, under the 'high-parallelism' criterion, Alternative (3) is the best. We will see, however, that Concurrent Prolog cannot adopt Alternative (3) in Section 4.1.

One remark on the parallelism in unification must be made here. It is known that the unification problem has a sequential nature in general, that is, parallelism cannot significantly help [Dwork et al. 84] [Yasuura 84]. However, this result should not discourage finding a good formulation of unification in parallel logic programming languages.

4.1 The Smallest Granularity Alternative

We have seen in Section 3.2 that any information textually specified in a goal must be available when head unification commences. To put it differently, a goal can be called only after all its arguments have been loaded. Thus the clause

```
:- p(a).                                                    (4.3)
```

and the clause

```
:- p(X), X=a.                                               (4.4)
```

should be defined as different. The word 'different' may be too strong, but at least we can say that Clause (4.3) more restrictive than Clause (4.4). The above difference is related to the semantics of the unification of a goal and a clause head, so the conclusion of Section 3.2 should be regarded as claiming also that the following two goals be different:

```
:- Head = p(a).                                             (4.5)
:- Head = p(X), X=a.                                        (4.6)
```

And this claim clearly rejects Alternative (3).

The following example would better show the importance of whether information is textually specified in a goal or not:

```
:- ..., p(A?), ....                                         (4.7)
```

We have attached a read-only annotation to A not simply because we do not want A to be instantiated by 'p' but because we want the predicate 'p' to wait and respect the value of A for clause selection unless the clause selection can be done without any reference to the argument value. Therefore, we can never rewrite Clause (4.7) to

```
:- ..., p(X), X=A?, ...                                     (4.8)
```

even though the 'read-only' property is inherited by unification.

Among important concepts in programming languages is referential transparency, which means that if the expression E1 and E2 denote the same value in the same context, we can textually replace E1 in a program by E2. Referential transparency in logic programming languages, if any, would require the following property: Whenever a term E appears in some goal (atomic formula), we can replace E by a new variable X and in conjunction with that goal put a new goal which unifies (equates) X and E. The above example, however, shows this property does not hold in Concurrent Prolog.

Logic programming languages have long been claimed to provide a good framework for mechanical handling of programs, e.g., program synthesis and

program transformation. So, properties such as referential transparency should be respected as much as possible. Losing it would make the language rules complex and mechanical handling of programs difficult.

4.2 Other Alternatives

What Alternative (2) means is as follows. Determination of a principal functor and the setting of its arguments can be done in parallel, but when the principal functor is determined and its arguments become accessible, they must be 'protected' if necessary, that is, if they are to be instantiated further. This solution looks consistent with the requirement that any information textually specified in a goal must be respected for clause selection, while retaining parallelism inherent in unification.

However, we have to examine in more detail. The problem is that this solution is rather ad hoc and that it exploits only a limited part of parallelism lost in Alternative (1). Arguments to be filled with non-variable terms can be protected by read-only annotations. However, there seem to be no means to protect the two arguments of 'q' in the following example.

```
:- q(Y, Y).                                                    (4.9)
```

This should not be defined as identical to

```
:- q(A, B), A = B.                                             (4.10)
```

because Clause (4.9) textually specifies that the two arguments of 'q' be identical, while Clause (4.10) has moved this information out of the goal. Clause (4.9) cannot select the clause

```
q(a, b).                                                       (4.11)
```

to reduce itself while Clause (4.10) can.

Clause (4.9) is not identical to the following one either:

```
:- q(A?, B?), A = B.                                           (4.12)
```

This is too protective. Clause (4.9) can select the clause

```
q(a, a).                                                       (4.13)
```

while Clause (4.12) cannot.

Considering all the above problems, Alternative (1) seems to be the best solution in Concurrent Prolog. When the value of the principal functor is available, any textually specified information on its arguments should be available also. This means that some sequentiality must be assumed for unification.

The above result urges us to examine the semantics of so-called 'metacall', i.e., a facility of 'call'ing some term (say T) as a goal. The goal T will be possibly incrementally generated by some other goal. So we must have some means to guarantee that all the argument information which should be assumed to be textually specified in the goal has been set up to T. Serialization by the commit operator will have to be used for this purpose. The following examples illustrate this.

```
Goal_1:     :- p.                                              (4.14)
Program_1:  p :- G=p(X), X=5 | call(G).                        (4.15)
```

```
Goal_2:     :- p(G), call(G?).                          (4.16)
Program_2:  p(G2) :- G2=p(X), X=5 | true.               (4.17)
```

In both of the above examples, it is possible to regard G as having the value p(5) right after commitment. So the goals call(G) and call(G?) can be defined to work as if they were specified textually as p(5).

To generalize, the value of a variable which is guaranteed to exist right after commitment can and should be treated like a textually specified value after that. The value of a variable which is guaranteed to exist is the value formed by the bindings made by unifications which is guaranteed to be finished by the language rules. The readers may think that the above discussion is too obvious, but it is never obvious. In Clause (4.16) above, the value of G is determined upon commitment by unification. However, as we have seen so far, it is not at all clear whether the principal functor 'p' and the argument value '5' arrives at the goal 'call(G?)' at the same time or not. This depends on the semantics of commitment and the semantics of unification, both of which are most delicate.

So far we have defined the semantics of unification involving non-variable terms. We must further define the property of logical variables.

We may well be tempted to define the semantics of the goal

```
:- p(X), q(X?).                                         (4.18)
```

as equivalent to

```
:- p(X), X=Y, q(Y?).                                    (4.19)
```

because they are 'logically' identical. Defining these two goals as identical means that communication by shared logical variables may have potential delay. This delay is allowed in another parallel logic programming language Guarded Horn Clauses [Ueda 85], but in Concurrent Prolog this cannot be allowed.

Firstly, [Shapiro 83] seems to assume no delay for shared variables, since it contains the specification of binary merger as follows (only recursive clauses are shown):

```
merge([X|Xs], Ys, [Z|Zs]) :- merge(Ys, Xs?, Zs).       (4.20)
merge(Xs, [Y|Ys], [Z|Zs]) :- merge(Ys?, Xs, Zs).       (4.21)
Goal:   :- merge(As?, Bs?, Cs).                        (4.22)
```

If we allow delay, the first argument Ys in the body goal of Clause (4.20) will be instantiated by the head argument '[X|Xs]' of the same clause upon recursive call. If there is no delay, it cannot be instantiated because Ys has been unified with Bs? in the head unification. Secondly, the 'protected data' technique also assumes there is no delay between two or more occurrences of a shared variable. For otherwise the information of protection would be delayed and hence might be violated.

Therefore, we must not assume any delay for shared variables: All occurrences of the same variable must denote the same value at the same time. We consider that allowing delay for shared variables, even though possible, would considerably change the rules of Concurrent Prolog, which would amount to designing another language.

5. PROCESSING HEADS AND GUARDS

It must be clearly specified what kind of parallelism should be allowed for processing heads and guards. In this regard, the semantics of

Concurrent Prolog shown in [Shapiro 83] and [Shapiro and Takeuchi 83] has the following problems.

5.1. Head Unification

The rules of Concurrent Prolog do not mention the order of unification of head arguments at all. At least four solutions seem to be candidates:

(1) Head unification is performed in parallel. A pseudo-parallel implementation is allowed, but no sequentiality is assumed conceptually.
(2) Head unification is performed sequentially in some order not defined by the language. The implementation can arbitrarily choose one of the possible orders. A program that depends on a particular order is erroneous.
(3) The mixture of (1) and (2) above. That is, the head unification is performed sequentially in some order, or in parallel. A program that depends on a particular strategy is erroneous.
(4) The head unification is performed sequentially, from left to right.

Solution (1) is preferred because sequentiality in the language rule should be minimized according to the principles stated in Chapter 1. The set of possible results of the execution of a program should remain unchanged when we systematically change the order of arguments of some predicate throughout the program and a goal clause. The 'result' mentioned above may include at least the following:

(a) Whether the computation terminates or not,
(b) If it terminates, whether it succeeds or not, and
(c) If it succeeds, what bindings are made to the variables in the goal clause.

The choice of the solution (1) has an influence on allowable implementations: Sequential unification from left to right becomes inadequate. Consider the following program:

```
Call:    :- p(a,  a).        (5.1)
Program:    p(A?, A).        (5.2)
```

The left-to-right head unification suspends while the rule states it must succeed.

In fact, any implementation of head unification which assumes a specific order is inadequate. For example, there is no specific order in which both of the following two goals succeed:

```
Call_1:  :- p(A,  A?).       (5.3)
Call_2:  :- p(A?, A ).       (5.4)
Program:    p(a,  a ).       (5.5)
```

Note that the above arguments apply to any unification performed in computation, for example unification upon commitment (see Chapter 2).

5.2. Head Unification and Guard Execution

The rules of Concurrent Prolog specify that the execution of a guard start after head unification has succeeded (Section 2.2, Paragraphs [2], [6] and [9]). We propose a different solution: Head unification and the execution of a guard are done in parallel. The reasons follow.

Consider the following example:

```
Call:          :- p(a,  a ).                                  (5.6)
Program: (a)      p(A,  B ) :- A=X?, B=X | true.              (5.7)
         (b)      p(X?, B ) :- B=X       | true.              (5.8)
         (c)      p(X?, X ).                                  (5.9)
```

According to [Shapiro 83], Program (a) succeeds and Program (b) suspends. Program (b) suspends because the unification B=X is performed only after the unification a=X? has succeeded. The result of (c) is not specified. If the unification of arguments is allowed to be performed sequentially, Program (c) may suspend; if the unification must be performed in parallel (as we recommended in Section 5.1), it succeeds. However, why should these three programs be not identical?

We propose to define (a) as a standard form of (b) and (c), and to make all the above programs succeed. The standard form of a clause must have a head whose arguments are distinct simple variables. Clauses (b) and (c) are considered as shorthand of (a). One justification of this proposal is that all these clauses are 'logically' identical. Defining the semantics of a clause in terms of its standard form may simplify the description of the semantics. The similar approach has been adopted in PARLOG [Clark and Gregory 84].

Our proposal could be justified also from a practical point of view. Each clause performs head unification and executes its guard to determine whether it can be selected. Looking back our programming style, we usually write in the head what we can write either in the head or the guard, and we write in the guard only what we cannot write in the head. An example of the former is the syntactic check of an input argument, and an example of the latter is the arithmetic comparison of two input values.

As far as we see, this choice has been made by the content of check, not by the its order. There seems to be no program that cannot be written without using the fact that head unification precedes the execution of a guard. The only reason why we use head unification for more than argument passing is that the use of head unification is good for concise description.

6. UNIFICATION OF TWO READ-ONLY VARIABLES

Paragraph [3] does not explicitly state the semantics of unification of two read-only variables. Kusalik took up this subject in [Kusalik 84] and argued that if a clause such as

```
p(X?) :- guard(X) | body(X).                                  (6.1)
```

is to be allowed, the head unification invoked by the call

```
:- p(A?).                                                     (6.2)
```

should succeed. Then he proposed two possible revisions:

(a) Let the unification of two read-only variables X? and Y? succeed, and make X and Y (identical) non-read-only variables.
(b) Disallow read-only variables appearing in a head.

However, neither of these solutions is desirable:

(a) Assume that

```
:- X?=Y?, X=a.                                                (6.3)
```

is executed. If X?=Y? is executed first, X and Y become an identical non read-only variable. This means that the annotated variable Y?

becomes instantiated by the partner of the unification by the execution of X=a, which is inconsistent with the general property of annotated variables.

(b) As Kusalik himself says, a read-only variable in a head has useful applications [Hellerstein and Shapiro 84][Takeuchi and Furukawa 85] and should not be prohibited. Moreover, disallowing read-only variables in a head destroys the symmetric nature of unification.

An alternative solution to (a) might be to let the unification of two read-only variables succeed and then to make them identical read-only variables. This preserves the propagative nature of read-only annotations. However, consider the following goal:

```
:- X?=Y?, X=a, Y=a.                                    (6.4)
```

This goal suspends if X?=Y? is executed first, and succeeds otherwise. This looks like a new, undesirable kind of nondeterminacy which arises from the order of unification: Besides this, the only source of nondeterminacy is the commitment operation.

Shapiro's original interpreter makes the unification of two read-only variables suspend. This solution looks better than all the above alternatives, but Aida [unpublished] and Tanaka [unpublished] claimed that the unification of two identical read-only variables should succeed. Tanaka implemented Concurrent Prolog [Tanaka et al. 84] and found that it is inconvenient that the goal

```
:- X=Y?, X=Y?.                                         (6.5)
```

suspends while the goal

```
:- X=Y?.                                               (6.6)
```

succeeds. So the best solution will be to let the unification of two read-only variables suspend unless these read-only variables are identical, in which case it should succeed.

One consequence of this slight revision is that the implementation might become complicated a little bit. Without this revision, suspension of a goal is released only when the read-only variable which caused the suspension gets instantiated to some non-variable term. Now suspension of a goal can be released also when the read-only variable that caused the suspension is unified with some other variable, as long as the suspension is caused by the unification between two read-only variables.

Let us consider the goal 'X?=Y?' for example. This goal suspends if X and Y are both uninstantiated and not identical. One way to release this suspension is to instantiate both X and Y to non-variable terms. Our claim is that now there is another way to release the suspension, that is, to unify X and Y. This means that the suspended goal 'X?=Y?' should watch the instantiation of X and Y to uninstantiated variables as well as to non-variable terms.

7. THE PREDICATE 'otherwise'

The predicate 'otherwise' was first introduced in [Shapiro and Takeuchi 83]. An 'otherwise' goal that occurs in a guard succeeds if and when all other parallel-Or guards fail. The commonest use of this construct will be to use only one 'otherwise' as the sole guard goal of the last clause which handles the 'default' or 'exceptional' case. However, the above simple rule itself is not so restrictive; it implies the following.

(1) 'Otherwise' can appear in a clause other than the last clause: There is no order among clauses constituting a predicate.
(2) 'Otherwise' need not appear as the sole goal in a guard. If the guard of a clause containing 'otherwise' also contains other goals, that clause may not be selected even if all the other clauses have failed. However, the clause with 'otherwise' cannot anyhow cover all exceptional cases without restricting the clause head to the most general one, and there seem to be no reasons to restrict the form of a clause with 'otherwise'. The only possible restriction might be to inhibit two or more 'otherwise' goals in one guard, since this is harmless but useless.
(3) 'Otherwise' can appear in more than one guard. Assume that the following clauses are contained in the definition of 'p'

```
p(nil,       ...) :- otherwise | ...        (7.1)
p(cons(A,B), ...) :- otherwise | ...        (7.2)
```

and that all other clauses are proved to be unselectable. Then, if the first argument of 'p' is 'nil', Clause (7.1) will be selected since Clause (7.2) is unselectable. If the first argument of 'p' has the form 'cons(X,Y)', Clause (7.2) will be selected. Of course, if two clauses which are not mutually exclusive have 'otherwise' goals in their guards, deadlock may result:

```
p(X, Y) :- otherwise | ...                  (7.3)
p(X, Y) :- otherwise | ...                  (7.4)
```

It is, however, the responsibility of a programmer to avoid such deadlock.

Although the original definition may look too general, it is recommended for its simplicity. It is fairly easy to implement: 'otherwise' needs only to monitor the number of failing clauses and succeeds when it reaches the number of the other clauses. We know that this construct is useful for writing non-trivial programs, but we have to examine further where the generality of 'otherwise' stated above is really useful.

CONCLUDING REMARKS

We have discussed some subtle points on Concurrent Prolog. In Chapter 3, we examined the semantics of multiple environments and a commitment operation. We showed that we have to further define at least the following things:

(1) Timing of unification of local and global information,
(2) Availability of the values of goal arguments for clause heads and guards, and
(3) Access rule to local and global copies of variables

In Chapter 4, we examined the semantics of unification and the allowed communication delay between two or more occurrences of a shared variable. We saw that a non-variable term (say T) specified in a source text must have an 'indivisible' nature: The unification of T with some variable must be done as an atomic action. To put it differently, we must assume some sequentiality for unification. Another conclusion is that all the occurrences of the same variable must denote the same value at the same time. We must not assume delay for a shared variable which is instantiated

by some goal and whose value is referenced by another goal. These conclusions means that some transformation of a program clause allowed in the first-order logic may change its semantics.

In Chapter 5, we considered how to execute head unification and the corresponding guard. In Chapter 6, we examined the semantics of unification of two read-only variables. In Chapter 7, we examined the semantics of the predicate 'otherwise'.

It must be noted that the arguments in this paper applies also to Flat Concurrent Prolog [Mierowsky et al. 85]. Flat Concurrent Prolog is different from Concurrent Prolog in that only predefined test predicates are allowed in guards. Management of nested environments is no longer necessary, and implementation on a sequential machine is greatly simplified by doing clause selection as an indivisible operation. However, Flat Concurrent Prolog never removes the need of multiple environments conceptually, since head unification may instantiate global variables. Therefore, if Flat Concurrent Prolog is intended to be a language for a parallel machine, it must resolve the problems discussed in this paper.

The results of this paper should be helpful for defining the precise semantics of Concurrent Prolog. However, the resulting semantics would be more complex than we thought even if defined informally, and it would require considerable efforts to have a formal semantics. An alternative research direction will be to revise the language. The language Guarded Horn Clauses [Ueda 85] was found by the author in this direction. He abolished the multiple environment mechanism and read-only annotations at the same time, and got a much simpler language with slight loss of the expressive power of Concurrent Prolog.

ACKNOWLEDGMENTS

The author would like to thank Takashi Chikayama, Akikazu Takeuchi, Toshihiko Miyazaki, Jiro Tanaka, and other ICOT members, as well as the members of ICOT Working Groups, for useful discussions. Some of the problems discussed in this paper were found in our attempt in ICOT to implement Concurrent Prolog [Miyazaki et al. 85]. Thanks are also due to Anthony Kusalik, Steve Gregory, and Ehud Shapiro for their comments on the earlier versions of this paper. Katsuya Hakozaki, Masahiro Yamamoto, Kazuhiro Fuchi, and Koichi Furukawa provided stimulating research environments.

This research was done as part of the R&D activities of the Fifth Generation Computer Systems Project of Japan.

REFERENCES

(This document is a substantially extended version of [Ueda 84].)

[Clark and Gregory 84] Clark, K.L. and Gregory, S., PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Dept. of Computing, Imperial College, London, 1984.

[Dwork et al. 84] Dwork C., Kanellakis P.C., and Mitchell J.C., On the Sequential Nature of Unification, J. Logic Programming, Vol.1, No.1, pp.35-50 (1984).

[Furukawa et al. 84] Furukawa, K., Kunifuji, S., Takeuchi, A., Ueda, K., The Conceptual Specification for Kernel Language Version 1 (KL1), ICOT Tech. Report TR-054, Institute for New Generation Computer Technology, 1984.

[Hellerstein and Shapiro 84] Hellerstein L. and Shapiro E., Implementing Parallel Algorithms in Concurrent Prolog: The MAXFLOW Example, Proc. 1984 Int. Symp. on Logic Programming, pp.99-115 (1984).

[Hirata 84] Hirata, M., Operational Semantics of Pure Concurrent Prolog, Proc. First Conf. of Japan Society of Software Science and Technology, pp.255-258 (1984).
[Kusalik 84] Kusalik, A.J., On Unification of Read-only Terms in Concurrent Prolog, discussion paper, Computer Science Dept., Univ. British Columbia (1984).
[Mierowsky et al. 85] Mierowsky, C., Taylor, S., Shapiro E., Levy, J., and Safra, M., The Design and Implementation of Flat Concurrent Prolog, Technical Report CS85-09, Dept. of Applied Mathematics, Weizmann Institute of Science (1985).
[Miyazaki 85] Miyazaki, T., Takeuchi A., and Chikayama T., A Sequential Implementation of Concurrent Prolog Based on the Shallow Binding Scheme, Proc. 1985 Symposium on Logic Programming, pp.110-118 (1985).
[Shapiro 83] Shapiro, E.Y., A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report TR-003, Institute for New Generation Computer Technology (1983).
[Shapiro 84] Shapiro, E.Y., Systems Programming in Concurrent Prolog, Conf. Record of the 11th Annual ACM Symp. on Principles of Programming Languages, pp.93-105 (1984).
[Shapiro and Takeuchi 83] Shapiro, E.Y. and Takeuchi, A., Object Oriented Programming in Concurrent Prolog, New Generation Computing, Vol.1, No.1, pp.25-48 (1983).
[Takeuchi and Furukawa 85] Takeuchi A. and Furukawa, K., Bounded Buffer Communication in Concurrent Prolog, New Generation Computing, Vol.3, No.2, pp.145-155 (1985).
[Tanaka et al. 84] Tanaka, J. et al., Sequential Implementation of Concurrent Prolog--Copying Approach to Multiple Environments, First National Conference of Japan Society of Software Science and Technology, pp.303-306 (1984).
[Ueda 84] Ueda, K., Comments on the Concurrent Prolog Language Rules, Part 1-3, unpublished (1984).
[Ueda 85] Ueda, K., Guarded Horn Clauses, ICOT Tech. Report TR-103 (1985).
[Yasuura 84] Yasuura, H., On Parallel Computation Complexity of Unification, Proc. Int. Conf. on Fifth Generation Computer Systems 1984, Institute for New Generation Computer Technology, pp.235-243 (1984).