TR-090

A Sequential Implementation of Concurrent Prolog based on the Shallow Binding Scheme

Toshihiko Miyazaki, Akikazu Takeuchi
and
Takashi Chikayama

November, 1984

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

A Sequential Implementation of Concurrent Prolog based on the Shallow Binding scheme

Toshihiko Miyazaki, Akikazu Takeuchi, Takashi Chikayama

Institute for New Generation Computer Technology
Mita-Kokusai Building, 21F.
4-28, Mita 1-chome, Minato-ku, Tokyo 108
Japan

ABSTRACT

In this paper, we will present an efficient implementation scheme on sequential computers of Concurrent Prolog which is one of the stream-and-parallel logic programming languages. The key issues of the implementation are (1) the scheduling of suspended computation and (2) or-parallel evaluation of clauses in order to select one clause for the goal resolution. The solution to the first problem is briefly explained and we will focus on the solution to the second problem. The second problem is divided into two sub-problems: (2-1) realization of multiple environments for clauses executed in parallel and (2-2) realization of value access control which is necessary when guards are deeply nested.

The proposed implementation scheme is based on the so-called shallow binding scheme and introduces two new low level constructs, a trail cell and a local environment number. The former realizes multiple environments and the latter realizes value access control.

1. Introduction

Recently many logic programming languages based on stream-and-parallelism have been proposed and it becomes clearer that these languages are very powerful for parallel programming. Some of these languages are Relational Language [Clark and Gregory, 1981], Concurrent Prolog [Shapiro, 1983] and PARLOG [Clark and Gregory, 1984]. The common features of these languages are (1) and-parallelism, (2) communication through shared logical variables and (3) don't care nondeterminism based on committed choices.

Implementations of these languages were and are being performed by several researchers [Shapiro, 1983], [Clark and Gregory, 1984], [Levy, 1984], [Nitta, 1984], [Ueda and Chikayama, 1984]. The key issues of the implementation are (1) the scheduling of suspended computation and (2) or-parallel evaluation of clauses in order to select one clause for the goal resolution.

In this paper, we will present a new sequential implementation algorithm for the interpreter of Concurrent Prolog, which solves the above problems efficiently. The implementation scheme is based on the so-called shallow binding scheme, which is a well-known scheme for implementing the variable-value binding environment in LISP.

The structure of the paper is as follows. In the section 2, general computation model of Concurrent Prolog will be presented. In the section 3, the key issues of the implementation will be described. In the section 4, the shallow binding scheme will be described in detail. In the section 5, advantages and disadvantages of the scheme will be discussed compared with other schemes. Readers of this paper are assumed to be familiar with the stream-and-parallel logic programming languages.

2. Computation model of Concurrent Prolog

As in the case of Prolog, the computation model of Concurrent Prolog can be represented by an AND-OR tree. In the following discussion, AND nodes and OR nodes are referred to as AND-processes and OR-processes, respectively. An AND-process and an OR-process correspond to a literal and a guarded clause in Concurrent Prolog respectively.

Figure 1 shows a slightly modified AND-OR tree which illustrates the state of Concurrent Prolog computation at some time [Shapiro, 1983b].

A loop formed by $\longleftrightarrow$ is called an AND-loop, and a loop formed by $\Longleftrightarrow$ is called an OR-loop. Each loop has one parent process and several child processes.

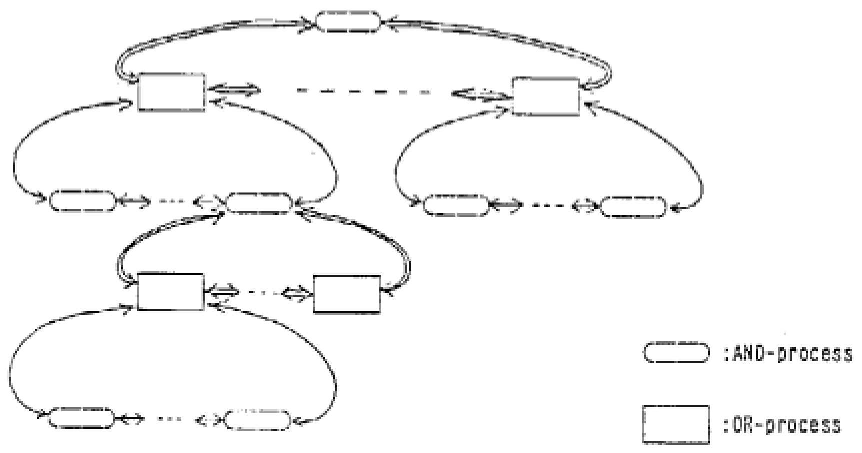


Figure 1. An AND-OR tree

In an AND-loop, the parent process is an OR-process and the child processes are AND-processes which correspond to conjunctive goals in the guard part of the OR-process. In an OR-loop, the parent process is an AND-process and the child processes are OR-processes which are candidate clauses of the AND-process. The clauses which may resolve a goal are called the candidate clauses of the goal. Without loss of generality, we assume the top level goal statement always consists of a single goal, which means that the root of the tree is always an OR-loop including this goal as a parent AND-process.

If one of the AND-processes in an AND-loop fails, then the parent OR-process fails. If all the AND-processes succeed, then the parent OR-process succeeds. Similarly since the child OR-processes in an OR-loop represent disjunctive clauses, if one of such OR-processes succeeds then the parent AND-process succeeds, and if all the OR-processes fail then the parent AND-process fails.

When all the AND-processes in an AND-loop succeeded, that is, the computation of the guard part of the clause represented by the parent OR-process in the AND-loop is successfully finished and reaches the commit operator, then the OR-process succeeds and does commitment. In the commitment, the OR-process kills all the other OR-processes belonging to the same OR-loop that it belongs to and replaces its parent AND-process by its literals in the body part. In other words, in the commitment the selection of the clause (the OR-process) which resolves the goal is established and all the other choices are discarded.

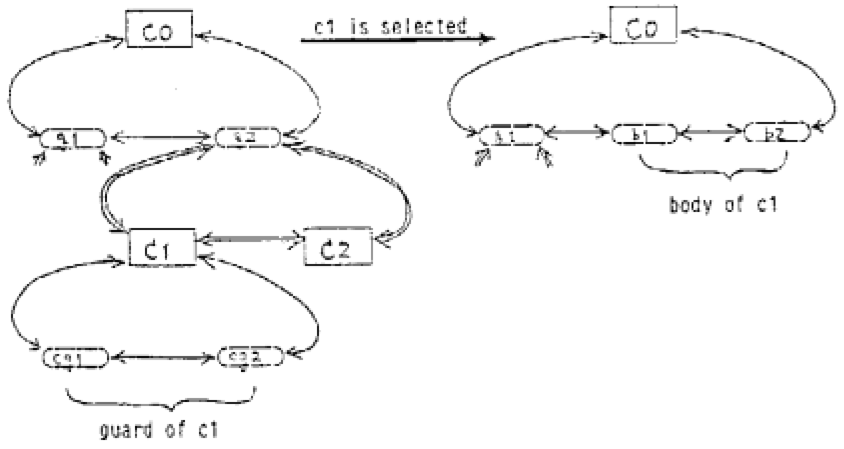


Figure 2 Commitment

Generally, in an AND-OR tree, all the leaf processes are active and other processes are waiting for the completion of the computation done by its child processes. In this sequential implementation, the system scheduler serves them one by one according to the scheduling algorithm. There are several scheduling algorithms, breadth-first, depth-first and bounded depth-first. In the breadth-first scheduling algorithm, the scheduler serves them in the same order as it traverses the tree in breadth-first order and in the depth-first scheduling, it serves them in the same order as it goes down the tree in depth-first order. The bounded depth-first scheduling algorithm is a mixture of both breadth-first and depth-first scheduling algorithm. In this algorithm, it serves them in the same order as it traverses the tree with some depth.

3. Key issues of the implementation of Concurrent Prolog

The key issues of an efficient implementation of Concurrent Prolog are the following two:

(1) The scheduling of suspended computation.

(2) The management of multiple computation environments for or-parallel evaluation of clauses.

There is a reasonable solution to the first problem [Shapiro, 1983b]. In this paper, we will concentrate on the second issue. However, before the second problem is specified in more detail, we briefly summarize the first problem and its answer.

Since Concurrent Prolog provides a read-only annotation as the synchronization primitive, when a goal can not be resolved without instantiating a variable annotated as read-only to a non-variable term, the computation suspends. The problem is how to schedule these suspended computation.

Since a computation suspends when it tries to instantiate a read-only variable to a non-variable term and it can resume when the read-only variable becomes instantiated to a non-variable term by another computation, it is natural to associate suspended computation with variables which caused suspensions. For this purpose, a suspension queue is introduces. A suspension queue is a queue which can keep suspended computation. The suspension queue is attached to an uninstantiated variable and it keeps only suspended computation which have read-only access to the variable and have tried to instantiate it to non-variable terms. The suspended computation kept in the suspension queue of some uninstantiated variable will become active when the variable will be instantiated to a non-variable term by another computation which has non read-only access to the variable.

The second problem, that is, the management of multiple environments, can be figured out by simple examples below.

```
goal1:    p(X), q(X)      /* X is an uninstantiated variable */

clause1:      p(1) :- guard1 | body1.
clause2:      p(2) :- guard2 | body2.

              Program 1.


goal2:   p(f(A)),r(A)    /* A is an uninstantiated variable */
clause3:   p(X) :- q1(X), q2(X) | ... .
clause4:   q1(f(1)) :- ... | ... .
clause5:   q1(f(2)) :- ... | ... .
clause6:   q2(f(B)) :- ... | ... .

              Program 2.
```

In the Program 1, during the parallel evaluation of two clauses, clause1 tries to unify the variable "X" in the goal "p(X)" with "1", while clause2 tries to unify the same variable "X" with "2". These two bindings to the same variable must be kept independently in their own local environment until one of the clauses will be selected, and must be made invisible from other environments such as the environment for the goal "q(X)".

In the Program 2, the goal "p(f(A))" invokes the guard part "q1(X), q2(X)", of clause3, and "q1(X)" invokes clause4 and clause5. During the parallel evaluation of clause4 and clause5, the two inconsistent bindings to the same variable "A" will be made and they must be kept independently as in the case of the Program 1. When either clause is selected, the value of "A" which is kept in the selected environment is exported to the environment of the clause3, and now the goal "q2(X)" can have access to the value of "A". However it must not be exported to the environment of the goal2, because clause3 is not yet selected. In other words, although the variable "A" first appeared in goal2, its value was made in either clause4 or clause5 and it is still kept in the local environment of clause3. Therefore goal2 does not have access to the value of "A" until clause3 is selected.

Generally, in Concurrent Prolog, when the candidate clauses for a goal are executed concurrently (OR parallel), head unification and guard computation of each candidate clause must be executed in its own local environment until one of the clauses will be selected. When one of the clauses reaches the commit operator, computation of other clauses are abandoned and the local environment of the selected clause will be exported to the environment of the goal. The essential difficulty in OR-parallel evaluation of candidate clauses is in the realization and the management of multiple environments corresponding to independent computation of candidate clauses and the control of the value access illustrated by Program 2.

4. Realization of multiple environments by the shallow binding scheme

We employ the shallow binding scheme for the realization of multiple environments. The differences among the multiple environments come only from different instantiations to the variables in the goal. Therefore, in order to realize multiple environments, it is enough to keep multiple bindings for each variable in the goal. The shallow binding scheme implements this idea. In the following, the detail of this scheme is described.

4.1 Shallow binding and environment switch

When an OR-process is invoked, the variable cells for the variables appearing in the clause are allocated first in its environment. These cells will have values or reference pointers to other cells [Warren, 1977], [Yokota et al., 1984].

Generally, when, in some local environment, an uninstantiated variable appearing in the goal is unified with a non-variable term during the head unification or guard computation, the pair of the address of the cell of the variable and its current content is saved in the local environment and the non-variable term is written into the variable cell itself. The pair is called the "trail cell" and plays the central role in the realization of multiple environments. Since the value written into the variable cell must not be seen from other processes, when an other process is to be scheduled next, the local value must be saved in the local environment and the original value must be restored to the cell.

```
goal:     p(X), q(X)    /* X is an uninstantiated variable */
clause1:   p(1) :- guard | body.

      Program 3
```

```
a variable cell of X     a trail cell         an original value
  +-----------+      +----------------+      +----------------+
  |     1     |<-----|--+     |    +--|----->| uninstantiated |
  +-----------+      +----------------+      +----------------+

       (a)  The trail cell is used for saving original value

  a variable cell of X        a trail cell       a saved local value
  +-----------------+      +----------------+      +-----------+
  | uninstantiated  |<-----|--+     |    +--|----->|     1     |
  +-----------------+      +----------------+      +-----------+

       (b) The trail cell is used for restoring the local value

                  Figure 3 The trail cell
```

In the Program 3, when the head unification is performed for the goal "p(X)" and clause1, the pair shown in Figure 3a is produced. However, since clause1 has not been selected yet, "X" must be uninstantiated for the goal "q(X)". This means that, in the resolution of the goal "q(X)", the value of "X" must be recovered using the trail cell, as shown in Figure 3b. When the computation of the guard of clause1 is resumed, the local binding is recovered again using the trail cell. This procedure is called an environment switch.

In general, the head unification and the execution of the guard of each clause may require trail cells. Assume that we have the OR-tree of the form shown in Figure 4 (Each OR-process is assumed to have one or more trail cells, and OR-process 7 is the parent process of the currently active AND-process and OR-process 4 will be scheduled next).

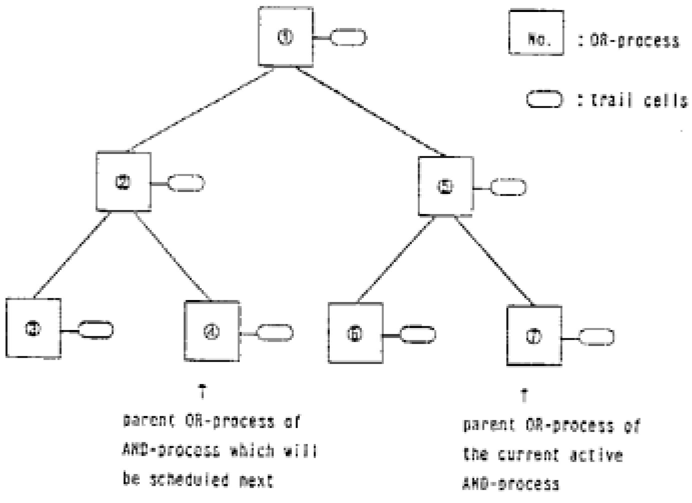

No. : OR-process
: trail cells
①
②
⑤
③
④
⑥
⑦
parent OR-process of AND-process which will be scheduled next
parent OR-process of the current active AND-process

Figure 4   An OR tree

In order to activate OR-process 4 next, the trail cells attached to OR-processes on the way between process 7 and process 4 (i.e. OR-process 7,5,2, and 4) are used to recover the local environment for OR-process 4 in the following way. (Notice that since the environment for OR-process 1 is shared between OR-processes 4 and 7, the trail cells in 1 are not used for this environment switch.)

1) If trail cells are used for saving the original bindings then their bindings are restored and the local bindings are saved.
2) Otherwise the local bindings are restored and the original bindings are saved.

In order to perform environment switching, the path between the OR-process of the current active process and the OR-process of the AND-process scheduled next must be identified quickly. Therefore, an environment pointer is introduced in order to enable a process to know the way to the current active process. An environment pointer points to either a parent or a child. When a pointer points to a child, there must be an active AND-process in the descendants. See Figure 5 for an example.

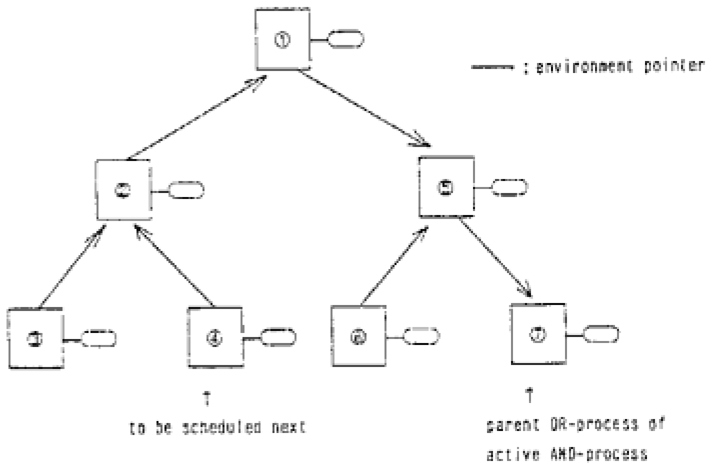


Figure 5 The OR tree with environment chain

Environment switching is described as follows:

1) Starting from the OR-process scheduled next, follow the environment pointers until the currently active process is reached, reversing the direction of pointers.
2) Follow the environment pointers in the reverse order until the OR-process scheduled next is found, exchange values, if there are any trail cells in the OR-process,
3) When the process is found, change its pointer to nil (The environment pointer of the active process is always nil).

Figure 6 shows the directions of pointers after environment switching has been completed.

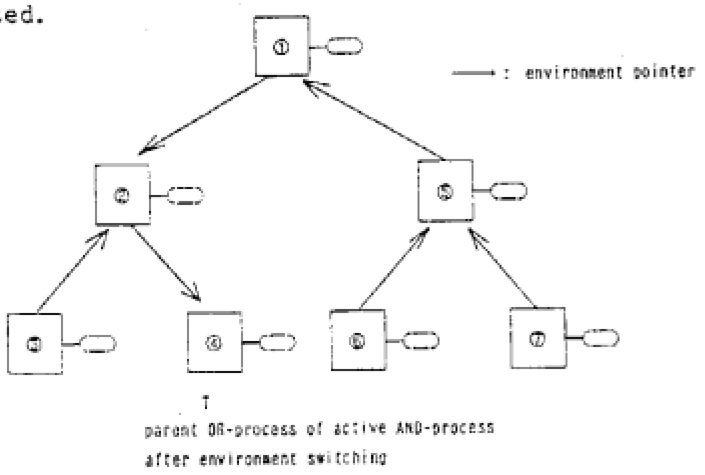

: environment pointer
1
2
3
4
5
6
7
parent OR-process of active AND-process after environment switching

Figure 6 The OR tree after environment switching

4.2 Value access control

As described in section 3, for the control of value access, it is necessary to determine whether a variable belongs to the local environment of the currently active process or not. Since the trail cell of the variable must be kept in the local environment if the variable does not belong to the local environment.

For the purpose of determining whether a variable belongs to an environment or not, a unique number (called a local environment number) is assigned to each local environment for identification. When an OR-process is created, a new local environment number is allocated and it is assigned to the associated environment. At the same time, the pointer to the cell containing the number is assigned to all the variable cells which are allocated in the environment, so that one can know whether an uninstantiated variable belongs to some environment or not by comparing their number. We show the simple example below.

```
             the cell of local
             environment number       variable cells
+---------+   +---------+              +----------+
|  OR  +-|----->|   21    |<-----------+--|--+       |
| process |    +---------+             |  +----------+
+---------+                            +--|--+       |
                                       |  +----------+
                                          |          |
```

Figure 7 All cells of uninstantiated variables point to the local environment number of the environment to which the variables belong.

In the commitment, the content of this local environment number cell is changed to a reference pointer to its parent(in the OR tree) local environment number cell. Figure 8 shows the situation where variable cells 2,3,4,5 and 6 belong to the same local environment after several commitments.

However, in some cases, several reference pointers may have to be followed in order to get a Local environment number. To avoid this overhead, once a reference chain is found, it is replaced by a direct reference pointer.

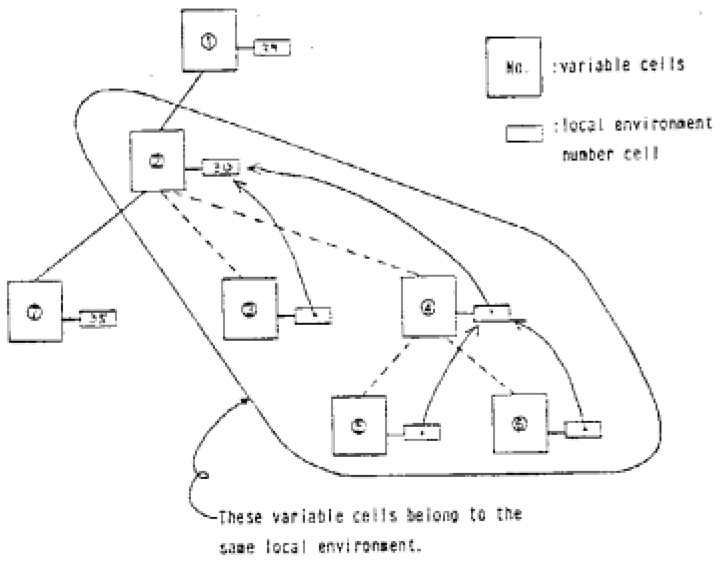

No. :variable cells
:local environment number cell
These variable cells belong to the same local environment.

Figure 8 local environment management

4.3 Dereference and head unification

When a normal(non read-only) variable is unified with another normal variable, a reference pointer is set up from one variable to another. When a read-only variable and a normal variable are unified, the read-only-reference pointer to the cell of the read-only variable is assigned to the cell of the normal variable. Thus, an attribute of "read-only" is represented by an read-only reference pointer. A variable cell consists of two fields, a tag field and a value field. The tag field of a cell indicates the attribute of the value field.

```
F : read only reference flag

dereference(VarType, VarCell)
   begin:
      if ( VarType = read-only-variable ) then F := on
                                           else F := off;
      return( do-deref(VarCell) )
   end;

do-deref(VarCell)
   begin:
      case( tag(VarCell) )
          undef:              return( VarCell );
          susp:               return( VarCell );
          non-variable term:  return( VarCell );
          reference:          do-deref( value(Varcell) );
          read-only-reference: F := on;
                                  do-deref( value(VarCell) );
   end;
```

Figure 9 Dereferencing algorithm

In general, a unified variable either refers to another variable via one or more reference pointers (or read-only-reference pointers) or has its own value. Obtaining a value by following reference pointers is called the dereference. A value obtained through the dereference is either an uninstantiated variable cell or a non-variable term (atom, integer, list, or vector). Figure 9 shows the algorithm for the dereference.

Table 1 Outline of unification.

C \ P	undef	susp	$undef^{ro}$	$susp^{ro}$	term
undef	C:=REF(P) or P:=REF(C)	C:=REF(P) or P:=REF(C)	C:=ROREF(P)	C:=ROREF(P)	C:=P
susp	C:=REF(P) or P:=REF(C)	C:=REF(P) or P:=REF(C)	C:=ROREF(P)	C:=ROREF(P)	C:=P
$undef^{ro}$	P:=ROREF(C)	P:=ROREF(C)	SUSPEND	SUSPEND	SUSPEND
$susp^{ro}$	P:=ROREF(C)	P:=ROREF(C)	SUSPEND	SUSPEND	SUSPEND
term	P:=C	P:=C	SUSPEND	SUSPEND	unify(P,C)

X^{ro} : means that the flag F is on.
P,C : result values of dereference

If "F" is on after the execution of "dereference" procedure, a read-only-reference pointer was found in the reference pointer chain. In this case, the value "v" obtained by the dereference is represented by "v^{ro}" in the table 1. The tag "susp" is used to represent a pointer to a cell containing a suspension queue. The tag "undef" is used to represent an uninstantiated variable without a suspension queue. These two tags indicate that the variable is uninstantiated.

In this table, "C:=REF(P) or P:=REF(C)" means either C:=REF(P) or P:=REF(C) may be used. (Care must be taken for unification between local variables when garbage collection must be taken into account).

SUSPEND means that unification will be suspended. In this case, the tag of a variable causing suspension is changed to "susp" if it is uninstantiated, and a suspension queue is produced. A suspended goal is put into the suspension queue. If the tag of the variable is "susp", the goal is added to the end of the suspension queue.

A detailed description of the unification algorithm used for a variable and a term is shown in Figure 10.

```
X  : variable cell
X^ : address of variable cell
T  : term

 if  instantiated(X) then
      unify(X,T);
 elseif  X belongs to the current environment  then
      X^ := T;
 else
      allocatetrailcellandsave(X^ ,X);
 endif;
```

Figure 10 Unification algorithm used for a variable and a term

4.4 Commitment

Commit operation is performed by an OR-process which is going to be selected. The procedure is as follows:

1) Abort other OR-processes in the same OR-loop.

2) Export the binding information in the local environment to the parent environment.

3) Create AND-processes for the goals in the body part of the clause, and replace the parent AND-process by them.

Export operation consists of two stages. In the first stage, for each trail cell kept in the local environment, the global value in the trail cell is unified with the current local value of the corresponding variable. In the second stage, if the first stage finishes successfully, the content of local environment number cell of the local environment is changed to the reference pointer to that of the parent environment as described in the section 4.2.

```
goal: p(f(A))        /* A is an uninstantiated variable */

clause1: p(X) :- q(X), ... | ... .
clause2: q(f(1)) :- ... | ... .
```

Program 4

In Program 4, the value "1" of the variable "A" is made visible only within the guard of clause1 when clause2 is selected. It is never seen from the goal. That is, when clause2 is selected, the trail cell for "A" is moved to the OR-process corresponding to clause1.

Unification performed in the exportation process differs from head unification in the following two points:

1) When a variable, the tag of which is "susp", is instantiated to a non-variable term, all the entries in that suspension queue are moved to the ready queue.

2) If unification suspends in the commitment, the OR-process is put in the corresponding suspension queue.

```
X^ :  Pointer to the cell
            of the variable "X" in the trail cell
V  :  Value saved in the trail cell
            (the current global value of the variable "X")

if  The variable cell pointed
        to by X^ belongs to the environment
        for the parent AND-process
then
    The current value of X and V are unified;
else  /* the variable does not belong to it */
    The trail cell is moved to the environment
                    of the parent AND-process;
endif;
```

Figure 11   Export

5. Discussion

There are several schemes by which multiple environments can be implemented. In this section, we will characterize some of them, copying scheme and binding scheme, and compare the performance of the shallow binding scheme with those of the other schemes.

a) Copying scheme

In this scheme, conceptually local copies of arguments of the goal are produced for each candidate clause. Head unification and guard computation can freely instantiate these local copies, without affecting the computation of other clauses. In the commitment, the local copies are unified with the arguments in the goal. The scheme can be classified into two schemes, eager copying scheme and lazy copying scheme, according to when local copies are made.

a.1) Eager copying scheme

In eager copying scheme, local copies of all arguments of the goal are made when the goal is invoked. This scheme is simple, however, it introduces a large copying overhead, and it may copy those not necessary for computation.

a.2) Lazy copying scheme

Lazy copying scheme was proposed by Levy [Levy, 1984],[Tanaka et al., 1984]. In the lazy copying scheme, terms are copied when they are tried to be instantiated. However, it needs complicated mechanism in order that goals in the same environment share locally instantiated terms.

b) Binding scheme

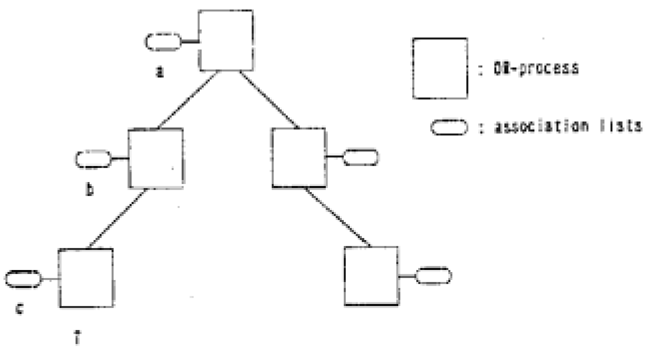
In this scheme, instead of making a local copy for each candidate clause, local bindings are made in an environment when variables in the goal are instantiated during the execution of a candidate clause. The scheme is classified into two schemes, deep binding scheme and shallow binding scheme, according to the implementation detail. In both scheme, local bindings for a clause are kept in the OR-process corresponding to the clause.

b.1) Deep binding scheme

In this scheme, local bindings for variables in the goal are kept as an association list in each environment [Ichiyoshi et al., 1984]. In the commitment, local values kept in the association list are unified with the current values of variables in the goal. This scheme requires a linear search for the association lists residing in the ancestor OR-processes in order to get a value of a variable when guard computation are deeply nested, because there is no way to know in which guard the variable is bounded to a term. Figure 12 shows a simple example environment representation of the deep binding scheme.

```
goal:      p(X)    /* X is an uninstantiated variable */
clause:    p(1) :- guard | body.


  the variable cell of X      the association list cell
    +-----------------+         +---------------+
    | uninstantiated  |<--------|--+    |   1  |
    +-----------------+         +---------------+
                                Association list is used
                                for saving local value.
```



The environment of this OR-process
consist of three association lists. a, b, c.

Figure 12   Deep binding scheme

b.1) Shallow binding scheme

This is the scheme described in this paper.

5.1) <u>Advantages of the shallow binding scheme</u>

Unification is fast compared with eager copying scheme, because shallow binding does not need copying of goals.

Dereference algorithm is clear, and very fast compared with the deep binding scheme. Because, in the deep binding scheme, it is always necessary to search association lists linearly for the value of a variable. The cost of getting values in the worst case is the order of the number of the ancestor OR-processes.

In lazy copying scheme, the first dereference to obtain the value of a variable is as expensive as in the deep binding scheme. Also once a term is accessed, it must be copied into all the environments between the environment for the currently active process and the environment to which the term belongs.

5.2) Disadvantages of the shallow binding scheme

The expensive operation of the shallow binding scheme is the environment switching which occurs when the next process is scheduled from the ready queue. The cost of environment switching depends on the distance between the currently active process and the process which should be scheduled next on the OR-tree. One possible relaxation remedy for this disadvantage is to employ the bounded depth-first scheduling method. We illustrate it by a simple example.

```
goal: ?- ap(M,A,A), ap(M,B,B). /* A,B are uninstantiated variable */

          ap(N,[a|X],XX) :-
               N > 0, N1 is N-1,
               m(XX?),
               ap(N1,X,XX)  | true.
          ap(0,[],).
          m([|]).
```

Program 5

In the Program 5, the number of reduction is

<the number of reduction> := (M * 4)+2

Predicate "ap" in Program 5 recursively calls itself in its guard. The number of recursive calls is specified by the first argument of "ap". A new trail cell will be created per each recursive call. In the breadth-first scheduling method, environment switch will take place per each recursive call. In the bounded depth-first scheduling, it will take place once every N times, where N is the depth.

6. Conclusion

In Figure 13, the performances of the Program 5 measured on different interpreters are shown.

The features of this shallow binding implementation are summarized below.

1) The realization of multiple environments by trail cells.

2) The realization of the value access control by the local environment number.

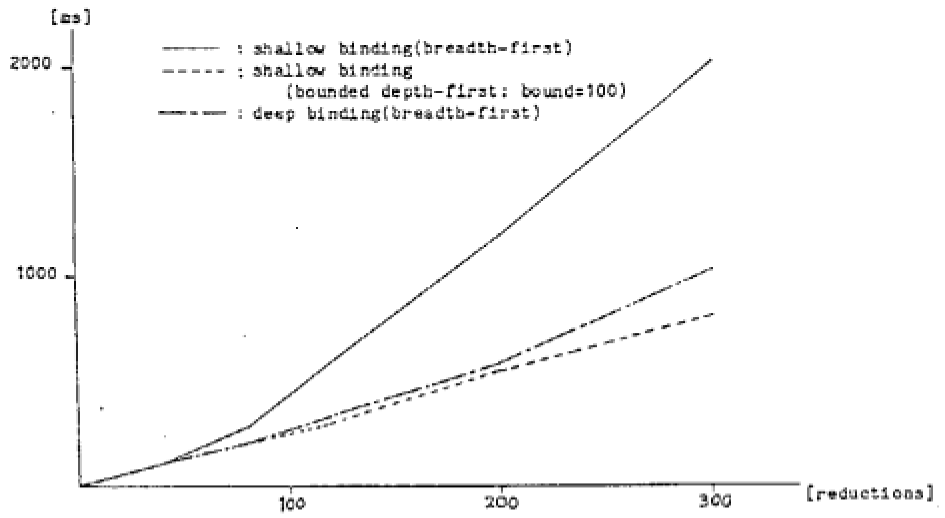


Figure 13 Results of Program 5

7. Acknowledgements

We thank Kazunori Ueda, Jirou Tanaka, Nobuyuki Ichiyoshi, Hitoshi Aida and other members of the group for the research of KL1 implementation for their valuable suggestions.

We would also especially like to thank Kazuhiro Fuchi, Director of ICOT Research Center, Kouichi Furukawa, Chief of second research laboratory of ICOT Research Center, and all the other members of ICOT, both for helping this research and for providing a stimulating place in which to work.

8. References

[Clark and Gregory, 1981] K.L.Clark, S.Gregory: A Relational Language for Parallel Programming, Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (1981).

[Clark and Gregory, 1984] K.L.Clark, S.Gregory: Notes on the Implementation of PARLOG, Research Report DOC 84/16, October (1984).

[Clark and Gregory,1984] K.L.Clark, S.Gregory: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, April (1984).

[Levy,1984] J.Levy: A Unification Algorithm for Concurrent Prolog, Proc. of 2nd International Logic Programming Conference, July, 1984.

[Nitta,1984] K.Nitta: On Concurrent Prolog Interpreter, Preprint of the 8th WGSF Meeting, Information Processing Society of Japan, 1984(in Japanese).

[Satou et al., 1984] H.Satou et al.: A Sequential Implementation of Concurrent Prolog -- based on Deep Binding scheme --, The First National Conference of Japan Society for Software Science and Technology, 1984

[Shapiro, 1983b] E.Y.Shapiro: Notes on Sequential Implementation of Concurrent Prolog, Summary of Discussions in ICOT, 1983 (unpublished).

[Shapiro, 1983] E.Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report TR-003 (1983).

[Tanaka et al., 1984] J.Tanaka, T.Miyazaki, A.Takeuchi: A Sequential Implementation of Concurrent Prolog -- based on Lazy Copying scheme -- , The First National Conference of Japan Society for Software Science and Technology, 1984

[Ueda and Chikayama, 1984] K.Ueda, T.Chikayama: A Practical Implementation of a Parallel Logic Programming Language, The First National Conference of Japan Society for Software Science and Technology, 1984

[Warren, 1977] Warren,D.H., Implementing PROLOG - Compiling Predicate Logic Programes, Vol.1-2, D. A. I. Research Report No. 39, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

[Yokota et al., 1984] Yokota,M., et al., A Microprogrammed Interpreter for the Personal Sequential Inference Machine, Proc. of FGCS'84, Tokyo, Nov. 1984.