6-26-1987

An interpreter for Parallel Prolog, a study and implementation

Morad Benyoucef

Rochester Institute of Technology
School of Computer Science and Technology

An Interpreter for Parallel Prolog,

A Study and an Implementation

by
Morad Benyoucef

A thesis submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

Andrew Kitchen 6/26/87
Professor Andrew Kitchen

John A. Biles 6/26/87
Professor Al Biles

Jim Heliotis 6/26/87
Professor Jim Heliotis

june 26, 1987

Rochester Institute of Technology
School of Computer Science and Technology

An Interpreter for Parallel Prolog,

A Study and an Implementation

by
Morad Benyoucef

I, Morad Benyoucef, prefer to be contacted each time a request for reproduction is made.

I can be reached at the following address:

Morad Benyoucef 06/29/1987

Morad Benyoucef date

ABSTRACT

In april 1982, a new institute named ICOT (Institute for New Generation Computer Technology) was created in Japan. The institute was to support the FGCS (Fifth Generation Computer Systems) project. The project is a tremendous effort to enter the fifth generation of computing. The idea is to have a Logic Programming Language as a base language for the project.
" The goal is to develop basic computer technology to build an intelligent computer system and its prototype which will have an inference function and an intelligent interface function." [18]

The inference machine to be developed will be a parallel Logic Programming Machine consisting of hundreds of processing elements, a structured memory and a network element.

Assuming that the technology can provide us with a multiprocessor capable of supporting the execution of several procedures or processes in parallel, the problem is to build an interpreter for Concurrent Prolog called TCP (Toy Concurrent Prolog). TCP is to be implemented on a single processor with simulated concurrency. The implementation will provide some program annotation schemes to make communication between concurrent processes possible.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

1. Introduction :

The idea of Parallel Prolog is very recent. Since Prolog was chosen as a kernel language for the Japanese Fifth Generation Project, the idea has been subject to ample discussions.

Parallelism, in Prolog in particular, and in Logic Programming Languages in general, became a main topic in recent conferences on Logic Programming and Computer Architecture.

The goal is to develop methods for extracting Parallelism from standard Prolog in order to achieve faster execution on a multiprocessor. The classic computational model of Prolog involves pattern matching in terms of logic. The matching expands the computation into a proof tree until terms are shown to be true or false.

In sequential execution, the proof tree is expanded in a depth-first manner, the goals of a clause are executed left to right, and the goals of the most recent clause entry are matched first. To execute a goal, the system searches for the first clause whose head matches or unifies with the goal. The unification process creates a binding of variables to terms and other variables.

Example:

```
father(john,peter).

father(adam,peter).

brother(X,Y) :- father(X,Z) , father(Y,Z).
```

Let us pose the query: | ?- brother(john,adam).

1. X is instantiated to john
2. Y is instantiated to adam

3. The left-most goal of the clause is tried first: father(john,Z) is called and Z is instantiated to peter.

4. The second goal is tried: father(john,peter). The matching succeeds.

5. The two calls were successful, the answer is yes.

When a matched clause is activated, each of its goals is executed from left to right with the same bindings that unification found for the head. If at that time the system fails to find a match for a goal, it backtracks, rejecting the most recently activated clauses and undoing any bindings made for their clause head. The system then looks for another clause to be tried. If it finds one, it continues execution from there. If not, the original goal fails.

The described model has been used since the introduction of Prolog by Roussel and Colmeraurer in Marseille in 1972. Because of its simplicity, and even more because of the fact that Prolog didn't have the wide range of applications it has today, the sequential model seemed sufficient. With today's needs for efficient AI languages and fast query languages of which Prolog is potentially a major one, the sequential model is not enough.

My goal is to investigate the possibilities of Parallelism in Prolog, study the models already being established, and implement a parallel interpreter for an extention of a subset of Prolog called TCP for Toy Concurrent Prolog. I intend to use Modula-2 for the inplementation. It is portable, has rich data structures and offers an elegant way of running concurrent processes.

2. Prolog - A brief overview:

2.1. Introduction:

Contrary to what most people think, Prolog's main application is not to prove theorems in predicate calculus. Prolog is a programming language based on predicate calculus.

A Prolog program is a sequence of rules of the form:

```
⟨head⟩ :- ⟨tail⟩.

⟨head⟩.
```

Example:

```
a :- b, c, d.

b :- e.

c.

d.
```

In the first rule, 'a' is the head, 'b', 'c', and 'd' form the tail.

The syntax of Prolog rules is:

```
⟨rule⟩         ::= ⟨clause⟩.| ⟨unit clause⟩.
⟨clause⟩       ::= ⟨head⟩ :- ⟨tail⟩
⟨head⟩         ::= ⟨literal⟩
⟨tail⟩         ::= ⟨literal⟩ {,⟨literal⟩}
⟨unit clause⟩  ::= ⟨literal⟩
```

By literal, just for now, I mean an identifier starting with a lower case letter like a, b, and c without any variables (i.e parameterless procedures).

Execution of a Prolog program can be triggered by a "query", which is syntactically equivalent to a ⟨tail⟩.

To interpret a Prolog rule, we assume that a ⟨literal⟩ is a goal

to be satisfied.

The rule: a :- b, c, d.

can be interpreted as:

"Goal 'a' can be satisfied if goals 'b', 'c', and 'd' can be satisfied". The query succeeds if the goals can be satisfied using the rules of the program.

2.2. Unification:

To this point, we assumed that a (literal) was an identifier starting with a lower case letter (like a parameterless procedure in Pascal). Actually, Prolog makes use of variables (and constants) the same way other programming languages do.

A common programming language uses variables to store values, compute expressions, etc. A variable gets a value by the assignment operator (X:=Y*2; in Pascal). Prolog uses "unification" to assign values to variables, or to make two variables point to the same entity. Unification is present in all the programming languages that offer the possibility of calling procedures or functions (CALL SUM(2,3,S) and SUBROUTINE SUM(X,Y,Z) in Fortran).

Unification in Prolog is regarded as the main idea behind the language. In order to talk about unification, we need to complete the syntax given above, introducing the BNF grammar for variables and constants.

⟨literal⟩ ::= ⟨composite⟩

⟨composite⟩ ::= ⟨functor⟩ (⟨term⟩ {,⟨term⟩}) |

⟨functor⟩

⟨functor⟩ ::= ⟨identifier starting with a lower case letter⟩

⟨term⟩ ::= ⟨constant⟩ | ⟨variable⟩ | ⟨composite⟩

⟨constant⟩ ::= ⟨integer or identifier starting with

a lower case letter⟩

⟨variable⟩ ::= ⟨identifier starting with

an upper case letter or _⟩

Some examples of terms : Max, line(point(X,3),point(4,3)), etc. This BNF is incomplete but it describes the main structure of a Prolog program. Unification tests whether two terms T1 and T2 can be matched by binding (assigning a value to) some of the variables in T1 and T2.

One unification algorithm was presented by J.A Robinson, and is summarized in the following table .

Terms: T2 / T1	⟨constant⟩ C2	⟨variable⟩ X2	⟨composite⟩ T2
⟨constant⟩ C1	succeed if C1 = C2	succeed if X2 := C1	fail
⟨variable⟩ X1	succeed if X1 := C2	succeed if X1 := X2	succeed if X1 := T2
⟨composite⟩ T1	fail	succeed if X2 := T1	succeed if (1) T1 and T2 have same arity and functor (2) The matching of corresponding children succeeds

Important:

"c1 = c2" means that c1 and c2 are equal.

"X := c" means that X can be assigned the value c.

That covers what I need to say about Prolog at this stage.

Points not covered here will be discussed when necessary.

3. Parallelism in Prolog:

3.1. Introduction:

It has been usually assumed that logic programs were to be executed on a single processor. Prolog programs were interpreted as procedure calls, executed one at a time by one processor. The semantics of Prolog however permits programs to be executed in parallel by several processors. To make that possible, we need to provide a mechanism for communication between concurrent processes. Let us look at the possibilities of Parallelism in a Prolog program. There are four of them.

3.1.1. OR Parallelism:

OR Parallelism is possible when the goal unifies with the head of more than one clause.

Example:

```
mother(mary,peter).
father(john,sam).
father(john,peter).
parents(X,Y,Z) :- mother(X,Z) , father(Y,W) , Z == W.
```

Notice that there is no need for the third clause. The first two clauses can share the variable Z. It is written this way just to emphasize the point.

To know who the son of john and mary is, we pose the query:

```
| ?- parents(mary,john,S).
```

First, let us see how this would work in sequential Prolog.

1. The first goal of "parents" matches the only "mother" clause. Z is instantiated to peter.

2. The second goal of "parents" matches against the first "father" clause. W is instantiated to sam.

3. The third clause of "parents" is tried but fails because Z (= peter) and W (= sam) are not equal.

4. The system backtracks and tries the second "father" clause. W is instantiated to peter.

5. The third clause is tried with the new value of W and succeeds.

6. The main goal succeeds and the answer would be:

S = peter.

With a multiprocessor, a process is created for every alternative of "father". We could start the two processes for "father" at the same time. The two processes return two different values for W. The third clause is tried with W (= sam) and fails. There is no need for backtracking since we have another value for W (= peter) available. With OR Parallelism, backtracking is no longer needed. OR Parallelism is also the easiest form of Parallelism. The reason is that OR processes do not consume values from each other. They can be run in parallel and the only thing we need to worry about is synchronizing their execution. We will address this problem in the chapter about Backtracking.

3.1.2. AND Parallelism:

When the goals in the body of a procedure (clause) are started simultaneously this is referred to as AND Parallelism.

Example:

```
mother(mary,peter).

father(john,peter).

parents(X,Y,Z) :- mother(X,Z), father(Y,Z).
```

Let us pose the query:

```
| ?- parents(M,F,peter).
```

meaning: who are the parents of peter?

We know how this would be executed in sequential Prolog. Let us see how it would work with AND Parallelism.

- Z is bound to "peter".
- The two clauses "mother" and "father" in the "parents" clause are started at the same time.

We get an answer (M = mary, F = john) in about half the time it would take sequential Prolog to get it.

That was the ideal case. AND Parallelism is not that straightforward. It is the most complicated form of Parallelism and certainly the most talked about.

The problem with AND Parallelism is that the bindings (i.e. unifications of variables in a clause) are often interrelated. So, a good deal of communication is needed.

AND Parallelism will be covered in detail in section 3.2.

3.1.3. Stream Parallelism:

This form of Parallelism deals with structures (like lists). The idea is for a procedure to consume a data structure while it is still being produced.

Example:

```
q(List) :- p(List) , s(List) .
```

and the query:

```
| ?- q([1,2,3,4]).
```

"p" starts. It produces (or does something with) the head of the list. At this point, "s" starts processing the head.
p and s run in parallel except that s is delayed by the time it takes p to produce one element of the list. All this with the assumption that "p" produces "List" and that "s" consumes it.

3.1.4. Search Parallelism:

Search Parallelism is useful when dealing with a data base of clauses. The clauses are partitioned into disjoint sets, then concurrent processes are used to search each set separately.

3.2 AND Parallelism - A closer look:

As I mentioned before, AND Parallelism is the most complicated form of Parallelism in Prolog. This section describes by examples the different cases encountered when trying to realize AND Parallelism.

Suppose there is a conference in Dallas and we want our department to be represented. We need to send one faculty member who specializes in Computer Science and another one who specializes in Mathematics.

Let us make a rule:

```
send(X,Y) :- computer(X), math(Y).
```

Suppose we had a data base of facts:

```
math(arnold).
math(jones).
computer(perrier).
```

Let us pose the query: | ?- send(perrier,jones).

Notice that it is possible to check if "computer(perrier)" and if "math(jones)" in parallel.

This is the most practical case of AND Parallelism. It is straightforward, and the performance is significantly high. But this is not always the case.

Consider another example:

```
child(X,Y,Z) :- father(Y,X) , mother(Z,X).
```

meaning: X is child of Y and Z if Y is father of X and Z is mother of X. Let us pose the query:

```
| ?- child(X,peter,mary).
```

meaning: who is the child of peter and mary?

The goals father(X,Y) and mother(Z,X) cannot run in parallel. The reason is: they might not produce the same value for X. We have a binding conflict because the two goals share the same variable X. To solve the problem, we go back to our sequential execution model.

First, father(peter,X) is executed , returning a value for X (X is instantiated or becomes ground). Then mother(mary,'value of X') is executed. If mother(mary,'value of X') fails, we use backtracking.

We can now state:

" If the goals in a clause share one or more variables, they cannot be executed in parallel."

The rule just stated is not enough to insure us from binding conflicts. The reason is in the following example.

Let us go back to our "send" rule. We would like to save some money by sending only one faculty member to Dallas. This person must be a Computer Scientist and Mathematician at the same time. So we pose the following query:

```
| ?- send(X,X).
```

meaning: is there anybody whose major is Computer Science and Mathematics at the same time?

Now X and Y in the "send" rule represent one person. We say: "X and Y share or are dependent ".

Even though the two clauses share no variable, they cannot be executed in parallel.

Now for the last case, consider the query:

```
| ?- child(bob,Y,Z).
```

meaning: who are the parents of bob? and the rule :

```
child(X,Y,Z) :- father(Y,X), mother(Z,X).
```

We know that the two goals share the same variable X. But they can run in parallel because X is ground (instantiated to a term containing no variables) before the goals are called.

So we can state that:

" Two or more goals that share a variable can run in parallel only if that variable is ground before the call to the goals in question."

We just covered the problems that may occur when trying to execute two or more goals in parallel. The problems are due to the binding conflicts. We will look at the different approaches being taken in order to solve the binding conflicts problem.

3.3. Different approaches:

3.3.1. Introduction:

In this chapter, we will describe the different approaches taken in order to solve the binding conflicts problem.

We saw that only with AND Parallelism do we face the problem. Without getting into the details of the implementation, we will see some of the solutions and discover the goals behind them.

Many solutions have been proposed in order to solve the problem. The dominant one requires the user to annotate some variables in the clauses so that the goals involving these variables wait until they are fully instantiated. A clause that binds a value to a variable is called "producer", and one that uses that bound variable is called "consumer". That is the data flow approach. The approach is taken in Concurrent Prolog (Shapiro,Ehud Y.), Parlog (Clark,K.L. and Gregory,S.) and IC-Prolog (Clark,K.L. and McCabe,G.).

The annotation solution, although acceptable, does not meet the main goal of Logic Programming: that is hiding as much as possible control and related issues from the user. We will look at one language that uses this solution: IC-Prolog.

The second approach tries to solve the binding conflicts with minimal information from the user. It requires a compile-time and/or run-time analysis. We will look at the RAP (Restricted AND Parallelism) approach (DeGroot,D) in more detail.

3.3.2. IC-PROLOG:

IC-PROLOG was the first parallel Prolog to be designed (1979 by K.L.Clark and F.G.McCabe at the Imperial College of Science and Technology - London).

3.3.2.1. Procedures:

An IC-PROLOG procedure is an implication of the form:

B (-- L1 & .. & Lk , k)= 0

where B is an atomic formula, and each Li is a literal(i.e. an atomic formula or its negation).

Atoms are of the form R(t1,..,tn), where R is a relation name and each ti is a term.

3.3.2.2. Queries:

An IC-PROLOG query is of one of the three forms:

(i) : L1 & .. & Lk

(ii) t : L1 & .. & Lk

(iii) { t : L1 & .. & Lk }

- t is a term and each Li is a literal.
- Let X1,..,Xn be all the variables of the conjunction L1&..&Lk.
- Let Y1,..,Yk be the subset of these that do not appear in t.

The queries are read:

(i) For some X1,..,Xn , L1 & .. & Lk.

(ii) A t such that for some Y1,..,Yk , L1 & .. & Lk.

(iii) All the t's such that for some Y1,..,Yk , L1 & .. & Lk.

The natural numbers can be denoted in IC-PROLOG by the "successor" function of the usual decimal numerals. Thus, "S(S(0))" is "2" and "S(S(X))" is "any number greater than 1".

TIMES and LESS are built-in predicates. There are no input/output restrictions on their use. So, TIMES can be used to divide.

The following query (of the third category)

{ ⟨S(S(X)),Y⟩ : TIMES(S(S(X)),Y,36) & LESS(S(X),Y) }

means: Find all the pairs ⟨r,s⟩ such that r*s = 36 and r ⟨= s.

The following set is returned:

{ ⟨2,18⟩ , ⟨3,12⟩ , ⟨4,9⟩ , ⟨6,6⟩ }.

The procedure:

even(Y) ⟨-- TIMES(X,2,Y).

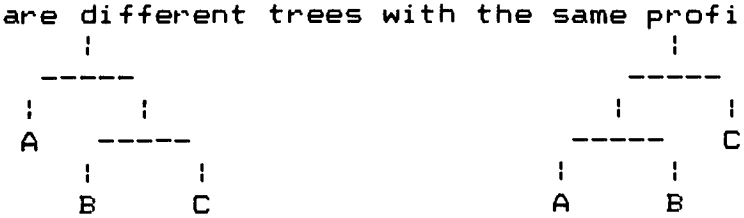
defines the property of being even.

3.3.2.3. Parallelism in IC-PROLOG:

We will make use of an example given in [11] to illustrate the annotation scheme characterizing IC-PROLOG.

Example:

Checking if 2 binary trees have the same leaf profile. The following trees are different trees with the same profile.



We use:

- "t(X,Y)" to denote a tree with subtrees X and Y.
- "L(u)" for a tree with just the label u.
- "u.X" for the list with head u and tail X.
- NIL for the empty list.

The procedure that checks if two trees X and Y have the same leaf profile will be:

```
sameleaves(X,Y) <-- profile(W,X) & profile(W,Y)

profile(U.NIL , L(U))

profile(U.Z , t(L(U),Y)) <-- profile(Z,Y)

profile(W , t(t(X,Y),Z)) <-- profile(W , t(X,t(Y,Z)))
```

Notice that IC-PROLOG is very similar to Prolog. The "sameleaves" example will be executed sequentially. We have not seen how to specify Parallelism in an IC-PROLOG program yet.

A sequential execution means that the profile of the tree X is generated first then tested against the profile of tree Y. This would be fine if the trees have the same profile. But if not, it would be wasteful to generate the leaf profile of X beyond the point at which they differ.

The best way is to execute the two "profile" calls in parallel. This is done by replacing "&" by "//".

```
sameleaves(X,Y) <-- profile(W,X) // profile(W,Y)
```

Two processes are generated. They will run in parallel. Now the idea of "producer" and "consumer" of the variable W is up to the system.

Example:

```
p(X,Y) <-- r(X) // s(X)
```

- Two processes are generated for r and s and put in a queue.
- The process that gets the chance to bind a value to X, becomes the producer. The other process becomes the consumer.

But would it not be better if the programmer was given the choice to decide which procedure should be the producer and which one should be the consumer of a certain variable? This can be handled.

3.3.2.4. Data Flow Annotations:

The parallel evaluation can be restricted so that only one process can generate the binding for a variable (in our "sameleaves" example, the variable W). We can either annotate W in the producer process with a "^" or in the consumer process with a "?". Thus,

```
sameleaves(X,Y) (-- profile(W,X) // profile(W^,Y)
```

makes the second call to "profile" the producer of W. W becomes a read-only variable for the first process (consumer). The consumer process checks a leaf whenever the producer process finds one.

3.3.2.5. Data Triggered Coroutining:

With the "^" and "?" annotations the programmer was given a way to choose a data flow scheme for his variables. If we go back to the annotated parallel version of the sameleaves example, we can see that the consumer process does not do any unnecessary work. But the producer might produce extra leaves, not knowing that the consumer has already found out that the profiles were different. A solution is to go back to the sequential model but using the data flow annotations.

Example:

```
sameleaves(X,Y) (-- profile(W,X) & profile(W^,Y)
```

There is no Parallelism but now, the second call to "profile" acts as a lazy producer of the binding for the variable W.

3.3.2.6. The "!" Annotation:

"!" is another way of controlling parallel evaluation. It is called the delay primitive.

Example:

```
p(X,Y) (-- s(X!,Y) // r(X)
```

s is suspended until the variable X is bound (instantiated) by another process (r in our example). If the two processes are suspended, the annotation is ignored.

3.3.2.7. The ":" Annotation:

This annotation is called the guard annotation.

Example:

```
B (-- G: A1 // .. // An
```

With the ":" annotation, we want to make sure that G is found to be true before starting a parallel execution of the Ai processes.

3.3.2.8. Stream Parallelism:

IC_PROLOG makes use of Stream Parallelism in an unusual way. It is called Stream I/O and is used with the READ and WRITE functions.

Example:

```
Y : READ(X) & p(X,Y)
```

X is a list of characters. As characters are read in, they are stored in the list binding for X that READ is lazily producing. p(X,Y) does not wait until the READ operation completes. It starts using the characters read right away.

3.3.3. Restricted AND Parallelism:

Before we get to the RAP model, let's describe the two methods that led to it.

3.3.3.1. Conery's Model:

Conery designed a series of run-time algorithms that determine which goal in a clause is the producer, and when two or more goals are to be executed in parallel.

Conery's algorithms can efficiently extract the Parallelism present in each particular clause invocation. The only weakness of Conery's Model is the enormous run-time overhead created.

3.3.3.2. Activation Mode Solution:

This model relies on the information supplied by the programmer: that is the activation mode.

One parallel Prolog that uses activation modes is that of P. Borgwardt of the University of Minnesota. The following example from [3] gives an idea on how activation modes are used.

Example:

```
?- mode insertx(+,+,-).
insertx(X,[A,L],[A,NewL]) :-
        A =( X , ! , insertx(X,L,NewL).
insertx(X,L,[X,L]).
```

The program inserts a number in a sorted list and produces a new list.

The - mode guarantees that the argument will be an uninstantiated variable.

The + mode guarantees that the argument will be already bound

(i.e. will be ground).

As was the case for IC-PROLOG's data flow annotations, this solution is not completely user-transparent. The other problem is that the worst execution graph of all cases is selected because there are no run-time checks.

3.3.3.3. Restricted AND Parallelism:

This is a compromise solution between the run-time (Conery) and the compile-time (Activation Mode) solutions. RAP makes use of the CGE (Conditional Graph Expressions).

Example:

```
child(X,Y,Z) :- father(Y,X), mother(Z,X).
```

Normally, "father(Y,X)" and "mother(Z,X)" cannot in general run in parallel unless

1. X was ground and
2. Y and Z were independent (did not share).

This information can be encoded in a CGE by rewriting the above clause as:

```
child(X,Y,Z) :- ( ground(X) , indep(Y,Z) |
                  father(Y,X) & mother(Z,X) ).
```

The logic of the clause has not changed. The checks before the "|" are not part of the semantics of the clause. The execution scheme becomes:

1. Try to unify "child(X,Y,Z)" with the calling goal. If not successful, fail.
2. Check if "X" is ground and if "Y" and "Z" are independent. In that case, start execution of "father(Y,X)" and "mother(Z,X)" in

parallel.

3. If the check fails, execute "father(Y,X)" first and then "mother(Z,X)" -i.e. execute them sequentially.

The CGE is determined at compile-time but the checking is done at run-time.

As we can see, the solution is really a compromise between the first two solutions. With Conery's solution, everything is done at run-time, and in the Activation Mode solution everything is done at compile-time.

The CGEs are to be generated automatically at compile-time. Let us just say that the generation is possible and that research on the subject is under way. We will not worry about how the CGEs are generated and suppose that our clauses have the CGEs embedded in them.

3.3.3.4. Conditional Graph Expressions:

Since they form the newest solution to the binding conflicts problem, CGEs deserve to be described but with little detail. A complete description can be found in [1].

A CGE can be defined as a series of conditions followed by a conjunction of goals, i.e.

((CONDITIONS) | goal1 & goal2 & ... & goalN)

(CONDITIONS) are checks on a list of variables. There are two sorts of them:

- ground((variable-list))
- independent((variable-list))

- ground evaluates to true if and only if all the variables in

(variable-list) are ground -i.e. they are instantiated to a term with no uninstantiated variables.

- independent evaluates to true if and only if all the variables in (variable-list) are independent (i.e. can be instantiated simultaneously by two processes without generating a conflict).

- (variable-list) is a collection of variables which have their first occurrence in the head of the goal or to the left of the current graph expression.

3.3.3.5. Forward Execution:

The forward semantics of CGEs is:

" if (CONDITIONS) evaluates to true, then all expressions inside the CGE can execute in parallel. Otherwise, they have to be executed sequentially in the order they appear in the expression."

3.3.3.6. Backward Execution:

Backward execution represents the actions to be taken following a failure in the head or body of a procedure (clause). If we were in sequential execution, backtracking to the point before failure would be the solution. But since some of the goals in the body of a clause have been executed in parallel, the notion of "point before failure" is lost. The backtracking scheme is complicated and time-consuming. A description of one approach (algorithms) taken in order to solve the problem can be found in [2].

3.3.3.7. Conclusion:

We saw that the Restricted AND Parallelism solution with CGEs was one that met the standards of Logic Programming: that is of hiding control issues from the user.

We also saw that an important overhead is created first at compile-time while determining the CGEs, then at run-time while interpreting the CGEs and in the worst case of backtracking.

The problem of deciding between user-transparency with complicated and slow execution, and non-user-transparency with simpler and faster execution has always been subject to heated arguments.

4. Implementation Issues in AND Parallelism:

4.1. Introduction :

Most sequential Prolog implementations make use of two run-time stacks.

The trail: This is a stack, or ordered list of all the variables that have been bound, in the order they were bound. On backtracking, the system uses this list to reset those variables back to an 'unbound' state, and thus back up the computation to an earlier state, from which another alternative can be tried.

The environment stack: As in block-structured languages (Pascal, Algol,...), the environment stack is the stack of Activation Records (AR). Each AR corresponds to one node in the search tree and contains the values of the variables new at that node. If a variable that was defined (and left unbound) in an earlier AR gets bound, it is put in the trail along with its new value (or a pointer to it). All the ARs in the stack are involved in the representation of the values of the variables present at the current active node.

Could the sequential model be used for a parallel implementation of Prolog? The answer is yes, but with some changes.

Since we are going to have more than one active node at the same time, there may be a conflict over the contents of an AR that is on the path to the root of two different active nodes (executed by two different processes). That AR will have to encode directly the values of the variables for those two different active nodes. In other words, in order to avoid a conflict, the variables must be read-only variables.

The sequential execution model is simple:

1. a variable gets bound in its AR,

2. if that AR is not the one being allocated, the address of the variable is placed in the trail stack associated with the new AR, so that on backtracking it can be set to unbound.

Our goal is to make it possible for several processes to share the same AR. In the following discussion, I will rely on the paper presented by D. S. Warren (an authority on the subject) [12]. Warren's Model for the implementation of Concurrent Prolog is very general and the only thing that it requires is a Multiprocessor.

To my knowledge, his idea has not been used in the known implementations of concurrent Prolog. I should say that the originality of his idea resides in the fact that it stays very close to the sequential model. A good description of the sequential model can be found in [14].

4.2. Warren's Model:

4.2.1. Forward Lists:

Instead of the 'trail' used in the sequential model, Warren proposes the use of a 'Forward List' (FL). An entry to the FL consists of a pair: a variable name and its binding.

During unification, if a variable not in the current AR becomes bound, then the AR containing that variable (the one where the variable first appeared and was defined in) is not changed; instead, the pair consisting of the variable name and the new binding is added to the FL of the current AR. As in the

sequential model, variables in the current AR that are bound during its allocation contain their bindings directly. Variables not bound during allocation of the AR in which they appear are indicated as free ('i-free' variables). To find the current binding of an 'i-free' variable, the system must scan the FLs of all the ARs from the current active node up the path to the root until either a binding for that variable is found or the 'i-free' variable is encountered (in which case, the 'i-free' variable is unbound).

In the following example,

if_n: represents a reference to an 'i-free' variable,

#: represents the 'i-free' counter, the next 'i-free' number to use,

FL: is the contents of the forward list of an AR,

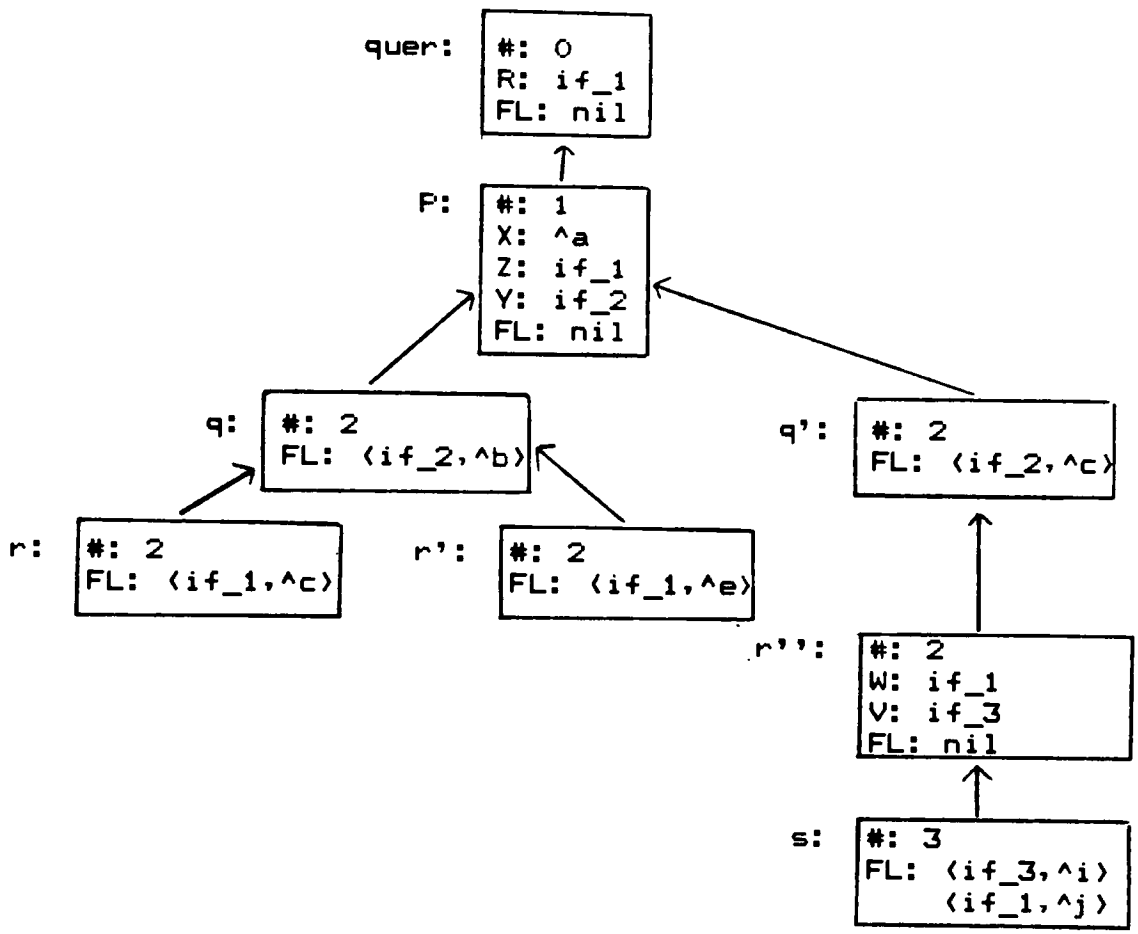
^: pointer to,

The boxes represent Activation Records (ARs).

The Tree of Activation Records represents all the possible answers to the query "p(a,R)" in a sequential environment. In a parallel environment the Activation Records will remain basically the same. The shape of the Tree will obviously change.

Example:

```
Program:    p(X,Z):- q(X,Y),r(Y,Z).
            q(a,b).
            q(a,c).
            r(b,c).
            r(a,d).
            r(b,e).
            r(c,W):- s(h,V,W).
            s(h,i,j).

Query:      p(a,R).
```



The algorithm is no other than the sequential model. Instead of directly accessing a current value by processing the trail, we process the Forward List (FL). We need do nothing to variables on backtracking. We only need to free the AR over which we are backtracking. The algorithm requires an extensive search to find the

current value of a variable.

4.2.2. Binding Arrays:

To avoid the search overhead caused by the first solution, Warren proposes the Binding Array solution. Any processor that is actively involved in extending the search tree maintains a private array, called a Binding Array (BA). A processor's BA is the set of forward lists in the ARs from the point at which the process is extending the tree up the path to the root.

For each (if_n, bindinptr) in the FLs, BA[if_n] = bindingptr, otherwise BA[if_n] = 'free'. The obvious advantage here is that a process working at an active node can find the binding of an 'i-free' variable, if_n, by examining the value of BA[if_n].

Example:

In the previous example, the process executing at node r'' will have a BA = [free,c,free]. The length of the BA is 3 which is the number of 'i-free' variables in the path to the root. The content of the BA suggests that if_1 and if_3 are still unbound, and that c is the binding for if_2.

4.2.3. Construction of the BA:

During unification, when an 'i-free' variable becomes bound, the processor's BA is updated in addition to the Forward List. On backtracking (if implemented), the BA can be updated by using the Forward List of the AR being backed over. If a processor moves to another active site (we will see how later), it can construct

the BA from that location by initializing it to 'free' and then running through the Forward Lists of the ARs on the path up to the root, using the bindings of the 'i-free' variables it finds there to set the new BA.

The BA is a private copy for a processor of all the places in the stack that various processes may conflict. Because we want Parallelism in the search of the tree (instead of the depth-first search used in sequential Prolog), it is likely that we will have more than one active node at the same time. The algorithm requires only one BA for each processor (and not one for each active node in the tree). When a processor moves from one node to another, it re-initializes its BA for the new location and continues processing from there.

4.2.4. Optimization:

The overhead of re-constructing the BA every time the processor moves to a new node may prove to be too high. An optimization is the incremental initialization of the BA (i.e. "lazy evaluation"). Until now, a BA entry may contain a 'free' indicator or an address pointing to the binding. We introduce a third value: 'unknown'. When a processor moves from one node to a new one, its BA is initialized to 'unknown'. Then when the binding of an 'i-free' variable is needed and its entry in the BA is 'unknown', the FLs are used to determine its value and update the BA entry.

The scanning of the FLs (from the current leaf up to the root) continues until we find a binding for the variable in ques-

tion, or we get to the AR in which the 'i-free' variable was initially allocated, in which case it must be still free.

During the scanning, the BA entries will be updated according to the FLs encountered ('free' or pointer to the binding). There is no need to traverse any FL more than once. We will have to keep an indication of the AR closest to the root whose FL has been already processed. Later, if another variable is encountered that has an 'unknown' entry in the BA, the scanning of the FLs can be picked up from that AR.

5. Modula-2

5.1. Introduction:

Modula-2 is one of the most successful parallel languages available today. Like Pascal, it was created by Niklaus Wirth. It is very similar to Pascal syntactically and semantically, but it outperforms it in many aspects. Modula-2's main additions with regard to Pascal are:

1. The "module" concept, and in particular the facility to split a module into a "definition" part and an "implementation" part.
2. The concept of coroutine as the key to multiprogramming facilities.
3. The PROCEDURE type which allows procedures to be dynamically assigned to variables.

The idea behind using Modula-2 to implement Concurrent Prolog is the fact that Modula-2 offers a well-structured way of running concurrent processes. Concurrent Prolog as described in the previous chapters is implemented as a finite set of processes, each executing a branch of the search tree. There is not much to say about Modula-2 that is not common to Pascal except the idea of processes, coroutines and modules.

5.2. Concurrency in Modula-2:

1. Processes:

In most implementations, Modula-2's library provides a module called "processes". It offers the necessary facilities for multiprogramming at a high level of abstraction. Unfortunately,

the imlementation we have does not. I had to implement the module processes myself, relying on Niklaus Wirth's implementation. I customized the module to fit my needs and that will be discussed shortly. The Definition Module will remain the same.

See point 1. in the appendix for a description of the Definition Module for the module "processes".

To ensure communication between processes, Modula-2 uses common variables and signals.

A. Common variables: They are used to transfer data among processes. These variables should be encapsulated in a module which guarantees mutual exclusion of processes. Such a special module is called a "monitor".

B. Signals: They carry no data and serve to synchronize processes. A process may send a signal and it may wait for it. Every signal denotes a certain condition or state among the program's variables, and sending the signal must imply that this condition has arisen. A process waiting for a signal may then assume that this condition has been met whenever the process is resumed. Sending a signal activates at most one process. Sending a signal for which no process is waiting is considered a null operation.

As an example, the producer-consumer problem is the best illustration of Modula's dealing with concurrency. See point 2. in the appendix for the code.

2. Coroutines:

A coroutine is a sequential program essentially like a pro-

cess. The differences between a process and a coroutine are:

A. Coroutines are known to be executed quasi-concurrently. Therefore their use avoids the difficult problem of interaction of genuinely concurrent processes.

B. The processor is switched from one coroutine to another by an explicit transfer statement. Execution of the destination coroutine resumes at the point where it was suspended by its own last transfer statement.

C. In every transfer, the destination coroutine is explicitly identified, and this contrasts with the WAIT and SEND statements used for synchronizing processes.

To use coroutines, a Modula-2 program must import the following necessary procedures from the module SYSTEM:

- PROCEDURE TRANSFER(VAR source, destination: ADDRESS);

A call to TRANSFER suspends the source (in order to be resumed later on with the statement following the transfer) and the destination to be resumed at its current point of suspension. In order to transfer control to the destination, a coroutine has to be created for it. That is what the NEWPROCESS procedure is for.

- PROCEDURE NEWPROCESS(P: PROC; A: ADDRESS; n: CARDINAL; VAR new: ADDRESS);

A call to NEWPROCESS creates a coroutine.

- P denotes the parameterless procedure which constitutes the program for the new coroutine.

- A is the origin address of a workspace needed to allocate the local variables of the coroutine and to store the coroutine's state while it is suspended.

- n denotes the size of this workspace in storage units.
- The variable new will point to the created coroutine. When control is transfered to the new coroutine, execution starts at the beginning of P.

First, we present one implementation of the module Processes in terms of coroutines (taken from [13]). See point 3. in the appendix for the code. Here are some comments about it.

Comments: taken from [13].

When a process is started with a call of StartProcess(P,n), a descriptor of the process and a workspace for its associated coroutine are allocated. The descriptor is inserted in a circular list (ring) containing all process descriptors created so far. The variable cp designates (the descriptor of) the currently executed process. By traversing the ring, any process can be reached. The successor process is denoted by the descriptor's field called "next". The crucial question is that of the representation of signals. Whereas at the user's level of abstraction a signal represents an arising condition, at the level of implementation it represents the set of processes that are waiting for that signal. Since the number of these processes is unknown, a sensible solution is to organize them as a linked list (in the descriptor, the field called "queue" gives access to that list). Hence, a signal variable represents the head of the list, and every process descriptor contains a field linking to the next process waiting for that same signal. Its value is of course NIL, if no such process exists.

From this description, the functions of the procedures SEND

and WAIT become obvious. SEND(s) takes the first element off the list s and transfers control from the sending process (identified by cp) to that process. Wait(s) appends the calling process (again identified by cp) at the end of the list s. Appending at the end embodies the requested fainess, ensuring that waiting processes cannot overtake other processes waiting for the same signal, because the list represents a first-in first-out queue. In principal, any process which is not waiting could now be resumed. Fairness is here achieved by simply proceeding through the ring starting at cp; the additional descriptor field called "ready" is used to quickly determine whether or not a process is ready for resumption.

For the working version with the changes I made, see point 4. in the appendix. The definition module remains the same.

Comments:

1. About Procedure Send(s): In the original version, the process waiting for s will be removed from the queue (i.e. s:=queue). Here we do not remove it for the following reason: In order for it to be resumed later after it gives up the processor, it has to execute a wait (the only way for it to go back to the queue). To get a larger number of interactions, I would like every processor to be resumed every time it gets a signal.

2. About Procedure Wait(s): When a process is activated by "Send", it is not removed from the stack. When a process executes a wait, it is already in the stack, so there is no need to put it back in. The reason we are able to do this is that we have only one process waiting for one message at any given time.

6. TCP (Toy Concurrent Prolog):

6.1. Introduction:

In this chapter, I will describe the system I implemented. It will be called TCP for Toy Concurrent Prolog. It is written in Modula-2 in a UNIX environment residing on a Digital Equipment Corporation's VAX-11/780 (Atlantis).

In the following discussion, examples will be used whenever necessary to clarify a concept.

6.2. Major Decisions:

6.2.1. The Grammar:

Since my goal is to study the problems of concurrency in Prolog, TCP will not be a commercial product but rather a case study. Therefore the interpreter will run the following simplified grammar:

```
⟨rule⟩          ::=  ⟨clause⟩.  |  ⟨unit_clause⟩.
⟨clause⟩        ::=  ⟨head⟩ :- ⟨tail⟩
⟨head⟩          ::=  ⟨goal⟩
⟨tail⟩          ::=  ⟨goal⟩ { ,⟨goal⟩ }
⟨unit_clause⟩   ::=  ⟨goal⟩
⟨goal⟩          ::=  ⟨functor⟩ (⟨term⟩ {,⟨term⟩})  |  ⟨funtor⟩
⟨functor⟩       ::=  ⟨identifier starting with a lower case letter⟩
⟨term⟩          ::=  ⟨constant⟩  |  ⟨variable⟩
⟨constant⟩      ::=  ⟨integer⟩
⟨constant⟩      ::=  ⟨identifier starting with a lower case letter⟩
⟨variable⟩      ::=  ⟨identifier starting with an upper case letter⟩
```

TCP will not support structures or operators. The heart of concurrent Prolog being "Unification", I intend to use simple variables to emphasize that point. An example program would be:

```
male(peter).
male(david).
male(john).
male(andy).
female(mary).
female(ann).
female(tina).
parent(peter, david).
parent(john, andy).
parent(david, andy).
parent(mary, david).
parent(mary, tina).
parent(peter, tina).
sibling(X,Y) :- father(X,F), father(Y,F), mother(X,M), mother(Y,M).
father(S,F) :- male(F), parent(S,F).
mother(S,M) :- female(M), parent(S,M).
```

6.2.2. Mode Declarations:

The grammar given above will be extended to enable the programmer to declare Execution Modes for the clauses in the program as in the implementation described in [3].

A Mode Declaration has the form:

mode: { clause_name ({+ or -}) }

"clause_name": It is the name of the clause (procedure) to which

the declaration will apply.

"+": appearing at the nth position in the list {+ or -} indicates that the nth parameter of clause_name will be instantiated (ground) upon call.

"-": appearing at the nth position in the list {+ or -} indicates that the nth parameter of clause_name will be uninstantiated upon call.

Example:

```
parents(X,Y,Z) :- father(X,Y), mother(X,Z).
```

This clause states that "Y" and "Z" are the parents of "X" if "Y" is the father of "X" and "Z" is the mother of "X".

```
mode: parents(+ - - )
```

This mode declaration states that: "When the clause 'parents' is called, the parameter 'X' will be instantiated, 'Y' and 'Z' will not."

The proper use of Mode Declarations can save a great deal of processing time. As an example, consider the following query applied to the "parents" clause.

```
| ?- parents(bob, F, M).
```

meaning: Who are the parents of bob?

Knowing that the two goals "father" and "mother" share the variable "X", the only way they can run in parallel is if the interpreter knew that "X" would be ground upon call to "parents". In the next sections we will see how the interpreter makes use of the mode declarations.

6.2.3. Producer/Consumer Relationship:

We already saw how IC-PROLOG uses the Data Flow Annotation to indicate which goal in the body of a clause will produce a certain variable and which one(s) will consume it. I think that giving the programmer the possiblility to choose a producer goal for a variable makes this implementation issue less transparent. Instead of the Data Flow Annotation, I used the following rule: "The goal with the leftmost occurrence of a variable is the producer of that variable. Any other goal that variable appears in is the consumer of that variable". The rule applies to the goals in the body of the clause but not to the head.

There is no need for annotations. The interpreter assumes that, in the body of a clause, the goal that a variable appears in for the first time is the producer of that variable, any other goal that variable appears in is the consumer of that variable. This approach was also taken by P. Borgwardt [3].

Example:

Consider the following rule:

```
p(X, Y) :- q(X, Z), r(X, T), s(Z, X).
```

q: produces:(X, Z), consumes:NIL

r: produces:(T), consumes:(X)

q: produces:NIL, consumes:(Z, X)

Establishing the Producer/Consumer relationships is done at Compile-time in just one pass.

6.2.4. Sharing:

Even though two or more goals share no variables, they might not run in parallel if they share some variables that are "dependent".

Example:

Consider the following rule:

```
p(X,Y) :- q(X), r(Y).
```

and the query:

```
| ?- p(Z,Z).
```

The goals "q" and "r" can not run in parallel because "X" and "Y" are "dependent". They share the same binding as the variable "Z".

How will TCP deal with this problem?

The idea is to take advantage of the Producer/Consumer relationships that exist among goals in the body of a procedure (clause). In the example given above, the Producer/Consumer relationships are determined at Compile-time and that results in the following:

p: produces: (X), consumes:NIL

r: produces: (Y), consumes:NIL

With the query | ?- p(Z,Z). we do not want "q" and "r" to run in parallel because "r" depends on "q" for the binding of "X".

Going along with the assumption that the leftmost goal in the body of a clause is the producer of the variables that appear in it, "q" will have to run first. When "q" succeeds, "r" can run with a binding for "Y" reported by "q".

All TCP has to do to insure that "q" runs before "r" is to add the variable "X" to the list of variables consumed by "r" (This is done at run-time). That way when a processor tries to run "r", it will find that it has to wait for "q" to succeed first.

This is a very convenient solution because it does not affect the overall architecture of the system. It fits perfectly in the Producer/Consumer strategy on which the system relies heavily during most phases of the interpretation. We will see how in the next sections.

6.3. Implementation:

6.3.1. Introduction:

As expected, TCP will support AND Parallelism. There is no notation in the syntax to tell the interpreter to run certain goals in parallel and others sequentially. The interpreter has to figure it out by making use of the Mode Declarations and the Producer/Consumer relationships between the goals in the body of a clause. The interpreter takes a clause with the intention of running all the goals in its body in parallel. If there is possibility of Parallelism, the interpreter will take advantage of it, otherwise the goals will run sequentially.

6.3.2. Tree of Activation Records:

6.3.2.1. Introduction:

We already saw what an Activation Record is in 5.2. Now is time to show how the tree of Activation Records (ARs) would look when developed by TCP.

Even when there is no possibility of Parallelism, TCP develops the tree in a parallel fashion.

Example:

Consider the following program of parameterless clauses:

The numbers appering before the clauses are just for documentation purposes.

(1) p :- q, r, s.

(2) q :- u, v.

(3) r.

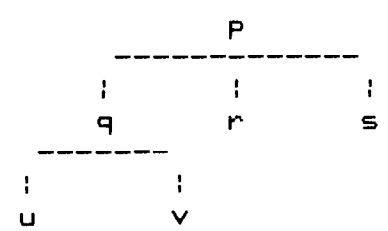
(4) s.

(5) u.

(6) v.

And the query: | ?- p.

The tree of ARs for the program will have the following structure:



Every node in the tree is an AR created by "Unification" of a goal with the head of a clause. Suppose we have three processors, and that all goals can run in parallel.

- Processor1 takes the initial query "p". It creates an AR for the Unification of "p" with the head of a clause (1)

- Processor1, Processor2 and Processor3 take the goals "q", "r" and "s" from the body of clause (1). They create ARs for them in parallel by doing the following:

Processor1 unifies "q" with clause (2)

Processor2 unifies "r" with clause (3) ("r" is solved).

Processor3 unifies "s" with clause (4) ("s" is solved).

- Processor1 and Processor2 take the goals "u" and "v" from the body of clause (2). They create ARs for them in parallel by doing the following:

Processor1 unifies "u" with clause (5) ("u" is solved).

Processor2 unifies "v" with clause (6) ("v" is solved).

An AR is the result of matching a goal (caller) to a procedure (callee) using the Unification rules in 2.2.

While a processor is creating an AR it only sees the ARs in its own stack (i.e. from the current AR up to the root).

- A processor creating the AR for goal "u" only sees the stack ("u","q","p")

- A processor creating the AR for goal "s" only sees the stack ("s","p")

Instead of a simple stack as in Sequential Prolog, TCP develops a Tree Structured Stack (i.e. Cactus Stack) for its search for a solution.

6.3.2.2. Construction of the tree of ARs:

A. Introduction:

At this point, it is not necessary for the reader to know how the processors interact with each other in order to understand this section. We will see how an AR is created and how it relates to the other ARs in the tree. The following example will be used all the way through the discussion.

Example:

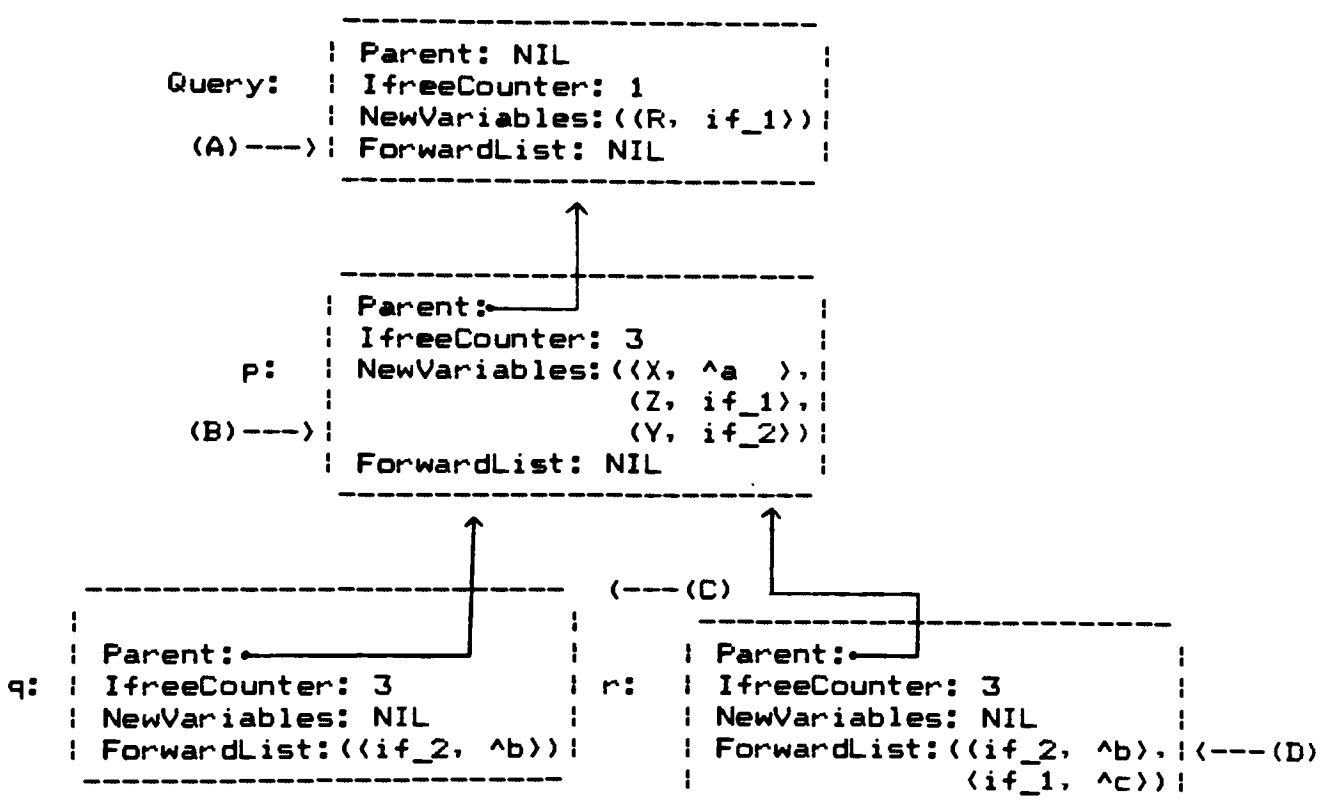
Consider the following TCP program:

```
p(X, Z) :- q(X, Y), r(Y, Z).

q(a, b).

r(b, c).

mode(+ -)
```

and the query:

```
| ?- p(a,R).
```

Here is the tree of ARs created by running this program:



B. Contents of an AR:

An AR contains the following information:

1. Parent:

A pointer to the AR above (i.e. the parent of the current AR).

In the example, the ARs for "q" and "r" have the same parent because they appear in the body of clause "p".

2. NewVariables:

The list of variables and their bindings that are new at (i.e. defined in) the current AR.

In the example, "X", "Z" and "Y" and their bindings are new at the AR associated with goal "p".

3. ForwardList:

The list of variables and their bindings that were defined in another AR but bound in the current AR.

In the example, "Y" is defined in the AR associated to "p", and bound in the one associated to "q". So ForwardList of "q"'s AR contains the i-free number and its binding.

4. IfreeCounter:

i-free (indicated as free) variables are defined in the current AR but they are not bound yet (they are still free). Whenever such variable is added to NewVariables, its binding is assigned the value of IfreeCounter. IfreeCounter is then incremented by 1.

In the example, "Z" is defined in the AR associated with "p" but it is still free. So its binding is assigned the value of IfreeCounter.

C. Construction of an AR:

The construction of an AR is an indivisible operation performed by a processor by matching a goal with the head of a clause.

Special Case:

The AR for the initial query is the first one to be created. It contains the variables that appear as parameters in the initial query. If the initial query succeeds, the bindings for those variables will be returned.

In the example, the AR named "Query" is the one associated with the initial goal "p(a,R)". "R" is defined but not bound yet, so it is indicated as free. If the goal succeeds, the binding for "R" will be returned.

General Case:

1. The first step a processor takes is to initialize all the Binding Array (BA) entries to "Unknown". See 5.2. for a description of Binding Arrays and Forward Lists.

2. The second step is, if the current goal (the one for which we are constructing an AR) consumes variables and these variables are not ground, the processor will have to look at the producers (they must be completed by now) and report the bindings of the consumed variables into the current FL and into the current BA.

In the example, a processor is running goal "r". It knows that "r" consumes the variable "Y" (defined in the parent AR as if_2) from goal "q". It also knows that "Y" is not affected by the Mode Declaration for "p". So "Y" is not ground upon call. The binding for "Y" ((if_2, ^b)) is reported from q's AR into the

current BA and into the current FL.

3. The third step: Now the processor can do the "Unification" between the goal (which we will refer to as "caller") and the head of the matching clause (which we will refer to as "callee"). The caller and the callee must have the same number of parameters. For every parameter in the goal and the corresponding parameter in the head of the matched clause, the processor will do the following depending on the present case:

Case 1:

The parameter in the caller is a constant (Const1),

The parameter in the head of the callee is a constant (Const2).

The two constants Const1 and Const2 must be equal, otherwise the goal fails.

Case 2:

The parameter in the caller is a constant (Const1),

The parameter in the head of the callee is a variable (Var2).

The variable Var2 is entered in NewVariables of the current AR with Const1 as its binding.

In the example, consider the caller "p(a,R)" and the callee "p(X,Z):-q(X,Y),r(Y,Z)". "X" unifies with "a", so we add (X,^a) to NewVariables in the AR of "p".

Case 3:

The parameter in the caller is a variable (Var1),

The parameter in the head of the callee is a variable (Var2).

The binding of Var1 is taken from the parent AR. Var2 is added to NewVariables of the current AR with the binding of

Var1 as its binding.

In the example, consider the caller "p(a,R)" and the callee "p(X,Z):-q(X,Y),r(Y,Z)". "R" unifies with "Z", so the binding for "R" (i.e. if_1) is taken from the parent AR. (Z,if_1) is added to newVariables of the AR of "p".

Case 4:

The parameter in the caller is a variable (Var1),

The parameter in the head of the callee is a canstant (Const2).

If Var1 is still free (i.e. the binding for Var1 in the parent AR is if_n) then look at the value of BA[if_n]. If BA[if_n] is "Unknown", perform the scanning of the ARs on top of the current AR as shown in 4.2.4. If BA[if_n] is "Free" then we have a binding for it. Add the pair (if_n,Const2) to the FL of the current AR.

If Var1 is bound, then its binding must be equal to Const2, otherwise the goal fails.

In the example, consider the caller "q(X,Y)" and the callee "q(a,b)".

"Y" is "Free". Add (if_2,^b) to the FL of the current AR.

"X" is not "Free" and its binding is equal to "a".

Now that the processor is done with the head of the matched clause, it will have to process the goals in the body of that clause.

For every goal do the following:

Enter into NewVariables all the variables in the goal that do not appear in the head of the clause (these variables will be

defined in the current AR but will not be bound).

In the example, the variable "Y" is added to NewVariables in the AR associated with "p(a,R)". Its binding is if_2. IfreeCounter is incremented and becomes 3.

6.3.3. The Concept of Process Descriptors:

6.3.3.1. Introduction:

The most delicate task in dealing with a set of interacting processes running in parallel is to insure an efficient way of communication between them.

TCP will simulate three processors (the way it is set up, the logic of the system remains true for any number of processors). The three processors participate in creating the Tree of Activation Records therefore they should do it in an organized manner.

Without getting into the details of what a processor does (that will be covered later), let us just say that:

"The job of a processor is to take a goal that is ready to run, find a matching clause for it, then do the Unification in the Tree of ARs."

The processor obviously needs to know which goal to take, when to take it, where to connect the AR it will create for it, and a number of other facts about the goal that would be needed in the course of backtracking.

All a processor needs to know about a goal is contained in a Data structure called Process Descriptor.

6.3.3.2. The Contents of a Process Descriptor:

The following example will be used all through the discussion:

Example:

Consider the following TCP program:

(1) p(X, Z) :- q(X, Y), r(Y, Z).

(2) q(a, b).

(3) r(b, c).

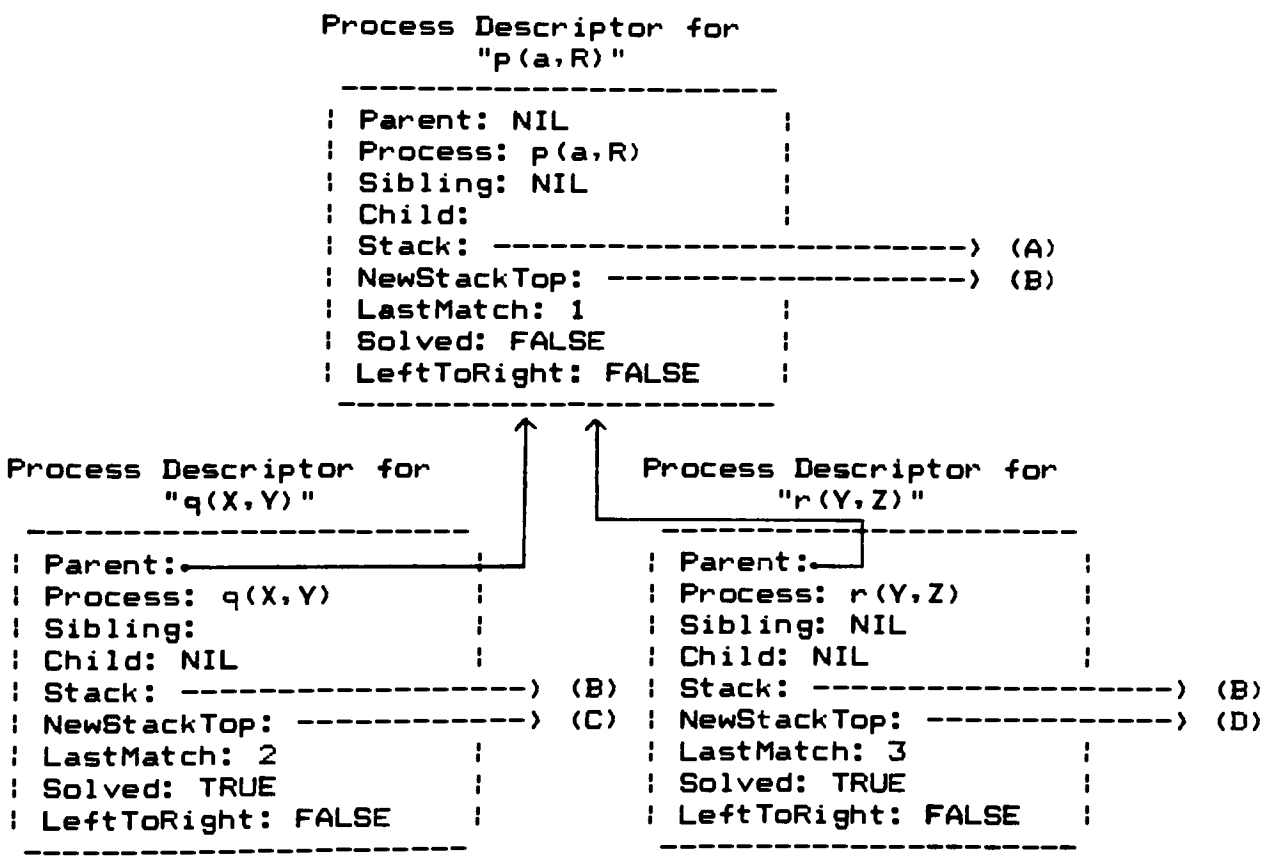
mode (+ -)

The numbers appearing before the clauses are for documentation purposes.

Consider the query:

| ?- p (a,R).

The Process Descriptors for the program would be:



(A), (B), (C) and (D) are pointers to the Tree of Activation Records on page 46.

A Process Descriptor contains the following information related to a particular goal.

1. Parent:

A pointer to the Process Descriptor for the goal that generated this goal.

In the example, the Process Descriptors for goals "q" and "r" have the same parent (Process Descriptor for "p"). The later has no parent because it is associated with the initial query.

2. Process:

A pointer to the code of the goal in question.

Important: A goal is always the "caller" and not the "callee" which is the head of a clause. Therefore, in the example,

"p(a,R)" is a goal, "p(X,Z)" is not.

"q(X,Y)" is a goal, "q(a,b)" is not.

3. Sibling:

A pointer to the Process Descriptor of the goal to the right of the goal in question.

In the example, the Sibling of q's Process Descriptor is r's Process Descriptor. The Sibling of r's Process Descriptor is NIL.

4. Child:

A pointer to the Process Descriptor of the leftmost goal in the body of the clause matched by the goal in question.

In the example, the Child for p's Process Descriptor is q's Process Descriptor.

5. Stack:

Pointer in the Tree of Activation Records to the AR at which the AR created for the goal in question will be attached.
In the example, Stack in p and r's Process Descriptors point to the same AR in the Tree because p and r appear in the body of clause p.

6. NewStackTop:

Pointer in the Tree of Activation Records to the AR created by matching the goal in question with a clause in the program.
In the example, NewStackTop for q's Process Descriptor points to the AR created by matching goal "q(X,Y)" and clause "q(a,b)".

7. LastMatch:

The index of the clause that last matched the goal in question.

8. Solved:

A flag, initially FALSE, that is set to TRUE if the goal in question is solved (has an empty body)

9. LeftToRight:

Will be used in the process of Backtracking.

6.3.3.2. Process Lists:

To every processor we associate an array of Process Descriptors called Process List. The process List for a processor is initially empty, and will contain pointers to the Process Descriptors of all the goal "taken" or "run" by that processor.

A processor's List (Process List will be reffered to as List) should be accessible to the other processors. Therefore all the Lists will reside in a Monitor that gives access to all the processors and most importantly insures Mutual Exclusion between them.

When a processor finds a goal to execute, it puts the Process Descriptor for that goal in its own List.
If the goal matches a clause that has an empty body, that means that the goal in question is solved. The Solved flag will be set to TRUE.

If the goal matches a clause with a body, that means the processor will have to create Process Descriptors for all the goals in the body of that clause. The "Child" field of the current Process Descriptor will be assigned a pointer to those children.
In the example, processor1 matches goal "p(a,R)" with clause "p(X,Z):-q(X,Y),r(Y,Z).". Then it creates the Process Descriptors for the goals in the body of the clause.

6.3.3.3. Creation and Use of Process Descriptors:

In order for a goal to be taken, it has to be ready.
A goal is ready if:
1. There is a Process Descriptor already created for it in the Process List of one of the processors.
2. The goal is not solved (the Solved flag is still FALSE).
3. The goal consumes variables and those variables are declared "ground" upon call to the head of the clause in which the goal

appears. Or

4. The goal consumes variables and the goals that produce those variables are solved. Or

5. The goal consumes no variables.

The processors are always looking for goals that are ready so they can run them (do the Unification for them in the Tree of ARs).

Special Case:

The initial query is a special case. Obviously, it is always ready. It is given to processor1 by the scheduler (main program). The Process Descriptor for it is created by the scheduler itself and put in processor1's Process List.

In the example, the Process Descriptor for the query "p(a,R)" is created by the scheduler and given to processor1. The only characteristic of this Process Descriptor is that it has no parent.

After the initial query is taken care of, all the processors are started. Processor1 will be the first one to find work because it has a Process Descriptor for a goal that is ready in its List.

General Case:

In order to find work, a processor first looks into its own List. If it finds a ready goal it takes it, otherwise it will look in the Lists of the other processors. If there are no ready goals in there either, it executes a Wait.

Now suppose that one processor finds work. It will do the following:

1. look for a clause in the program that matches the ready goal.

If there is no match, it will have to Backtrack.

If there is a match then it will do the following:

2. Do the Unification in the Tree of ARs. The AR created will be attached to "Stack" (a field in the Process Descriptor).

3. If the clause just matched has an empty body then:

The goal just executed is solved. The processor will have to "Reduce" (that will be discussed separately).

If the clause has a body then:

For all the goals in the body of the matched clause, it creates Process Descriptors and adds them to its Process List.

In the example, processor1 runs the goal "p(a,R)". Since the matching clause "p(X,Z):-q(X,Y),r(Y,Z)." has two goals in its body, processor1 creates two Process Descriptors for the goals "q(X,Y)" and "r(Y,Z)" and adds them its own List.

When the Process Descriptors are completed and added to the List, the processor sends Signals to the other processors telling them that there is work available. After the signals are sent, the processor goes back to the first step looking for work.

6.3.4. Reducing:

This task is performed by a processor when it solves the last goal in the body of a clause. A goal is solved when it has an empty body.

Example:

Consider the following TCP program:

(1) p(X, Z) :- q(X, Y), r(Y, Z).

(2) q(a, T) :- t(T).

(3) r(S, W) :- t(S), s(h, W, V).

(4) s(h, i, j).

(5) t(c).

mode(+ -)

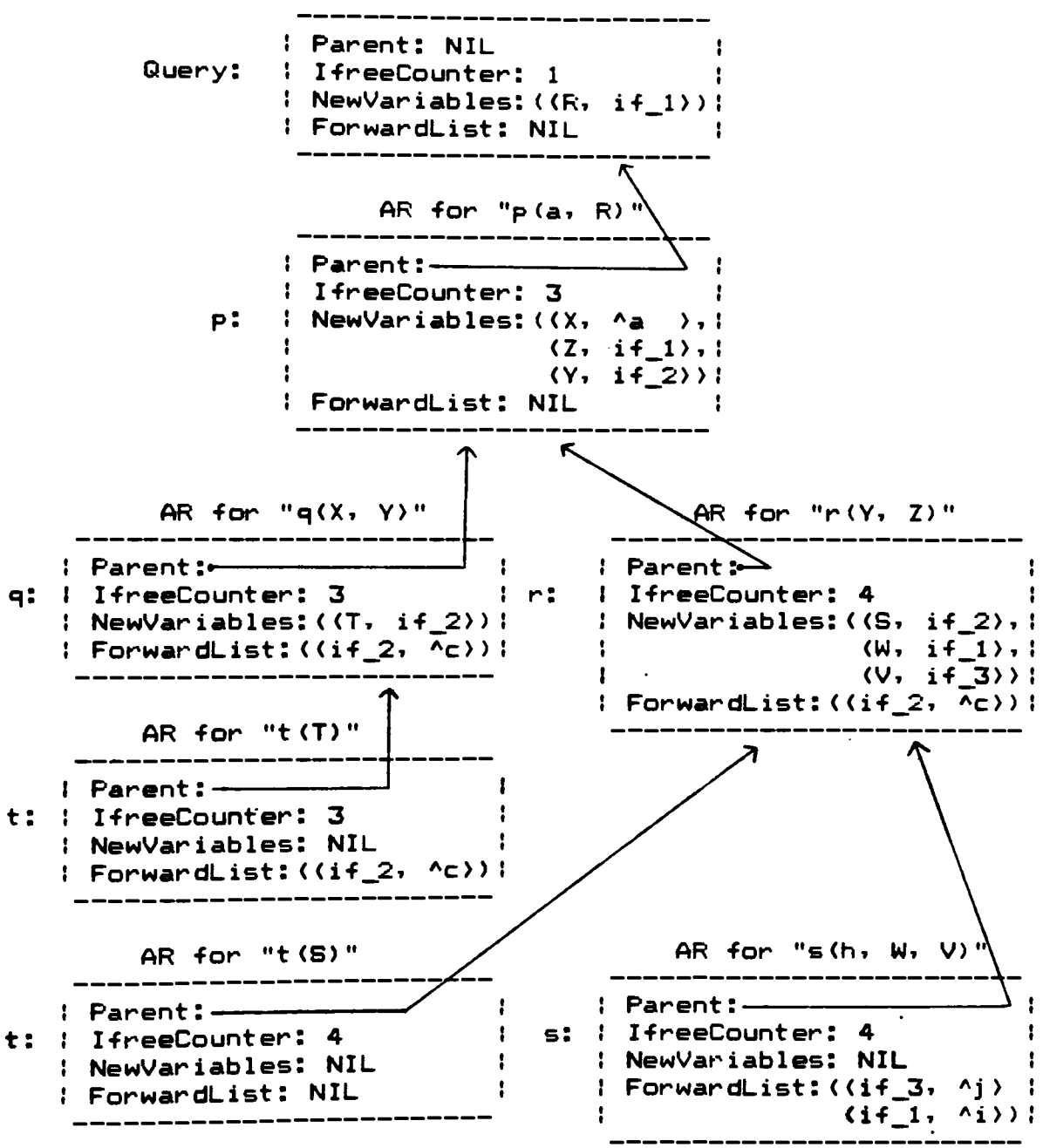
and the query: | ?- p(a,R).

When "t(T)" in clause (2) is solved (clause (5) has an empty body) the binding for "T" which is the binding for "Y" is reported to the goal "q(X,Y)".

If we look at the Tree of ARs for this example, we can see that: The AR associated with "t(T)" contains the binding for "T" which is the binding for "Y". That binding (i.e. (if_2, ^c)) is reported to the AR associated with "q(X,Y)".

Reducing uses a simple rule: " When all the goals in the body of a clause are solved, all the pairs (if_n, binding) in their Forward Lists such that n < Parent's IfreeCounter, are copied to the Parent's Forward List."

Here is the Tree of ARs created by running this program. At this point, the AR for goal "s(h,W,V)" is completed. Reduction for "t(S)" and "s(h,W,V)" is about to begin.



6.3.5. Resatisfying:

In the process of Backtracking, a goal that was solved earlier will be tried again for a new set of bindings. That is called "resatisfying" a goal.

Suppose that goal "p" is to be resatisfied. The following procedure will be applied:

```
procedure Resatisfy(p);
begin
  if (the AR associated with "p" in the Tree is not a leaf)
  then
     - Mark "p" Unsolved;
     - Undo the effects of Reducing "p"'s children into
       "p"'s AR;
     - Get the rightmost goal in the clause that was last
       matched with "p". That goal is "q";
     - Resatisfy(q)
  else
     We have a leaf in the Tree of ARs.
     The corresponding goal will have to run again.
     - Mark it Unsolved. It will be picked up by the next
       available processor
  end
end;
```

Example 1:

Consider the following program:

```
a :- b, c, d.
```

b :- e, f.

c :- g.

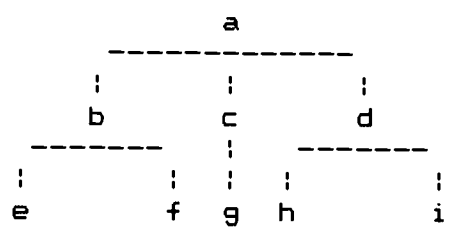
d :- h, i.

e. f. g. h. i. j.

i :- j.

and the query: | ?- p.

The Tree of ARs will have the following shape:



Resatifying "a" will be done as follows:

- Mark "a" Unsolved;

- Undo the effects of Reducing "b", "c" and "d" into "a";

- Mark "d" Unsolved;

- Undo the effects of Reducing "h" and "i" into "d";

Goal "i" will be picked up by the next free processor and tried for another set of bindings.

Example 2:

Consider the following TCP program: (The numbers appearing before the clauses are just for documentation.)

(1) p(X, Z) :- q(X, Y), r(Y, Z).

(2) q(a, T) :- t(T).

(3) r(c, W) :- s(h, W, V).

(4) s(h, i, j).

(5) t(d)

(6) t(c).

and the query: | ?- p(a, R).

A processor takes "r(Y,Z)" and tries to match it with clause (3). The variable "Y" was bound in "q(X,Y)" and its binding (i.e. (if_2, ^d)) was reported into the current ForwardList and the current BA. The processor fails to match the goal with the head of clause (3) because of "case 4" of the Unification process. There is no other clause to try for "r(Y,Z)" so this goal fails. The processor will have to resatisfy the producer of "Y" (We will see how this decision is made when we talk about Backtracking). The producer of "Y" is "q(X,Y)". When resatisfying "q(X,Y)", the processor will do the following:

1. Undo all the effects of Reduction: When "t(T)" matched clause (5) the binding for "Y" (i.e. (if_2, ^d)) was reported into the AR associated with "q(X,Y)". That has to be undone.

2. The rightmost goal in clause (2) is "t(T)". Its AR in the Tree is a leaf. Match "t(T)" with clause (6). The variable "T" will be bound to "c" and so will "Y". Now the Tree of ARs will look exactly as the one on page 60.

6.3.6. Backtracking:

6.3.6.1. Introduction:

In this section we will discuss the concept of Backtracking in TCP.

In Sequential Prolog, when a goal fails and there is another clause to test it against, there is no problem. Otherwise the system must backtrack.

Example:

Consider the following Prolog program:

(1) p :- q(X,Y), r(Z), s(Y).

(2) q(a, b).

(3) q(a, c).

(4) r(c).

(5) r(d).

(6) s(c).

(7) s(d).

and the query: | ?- p.

We know how this program would be executed in Sequential Prolog. Let us suppose that the goal "s(Y)" is being tested against clause (6). Since "Y" was bound to "b" earlier, "s(Y)" fails. "s(Y)" is tested against clause (7) and again fails. Now we back-track. The following will happen.

- "r(Z)" is resatisfied. It is matched with clause (5) (i.e. the last alternative for "r(Z)").
- "s(Y)" is tried unsuccessfully against clause (6), then clause (7). We backtrack again.

Important: Notice the irrelevance of resatisfying goal "r(Z)".

That action has no affect on "s(Y)", the reason being that "r(Z)" does not produce any variables that are consumed by "s(Y)". There is an enormous loss in processing time spent on resatisfying "r(Z)". This is one of the weaknesses of Sequential Prolog.

- "r(Z)" is tried out (i.e. there is no more clause to try it against). We backtrack again.
- "q(X,Y)" is tried against clause (3). "Y" is bound to "c".
- "r(Z)" is tried against clause (4) and succeeds.
- "s(Y)" is tried against clause (7) and succeeds.
- The initial query succeeds.

Sequential Prolog was developed to run on a single processor. With the now popular idea of muliprocessors, it is possible to avoid backtracking. The alternative is OR Parallelism.
Let us briefly see how the same example would run in an OR Parallel environment.

With the assumption that we have enough processors the following will happen.

- One processor takes goal "q(X,Y)". There are two possible clauses this goal can match (i.e. clause (2) and (3)).Two processes are created to explore the two possibilities concurrently.
- One processor takes goal "r(Z)". Same thing as before, two processes are created to explore clauses (4) and (5) concurrently.
- One processor takes goal "s(Y)". Same thing as before, two processes are created to explore clauses (6) and (7) concurrently.

When goal "s(Y)" fails (with Y = b), it asks goal "q(X,Y)" for another set of bindings (X = a and Y = c). That set of bindings is already available. "s(Y)" runs with (Y = c) and succeeds.

OR Parallelism will be discussed in more details in the next section.

6.3.6.2. OR Parallelism - A Closer Look:

Even though TCP does not use OR Parallelism, this section is devoted to giving a brief description of one of the implementations that use it. This implementation of Parallel Prolog is that of P. Borgwardt. The following discussion is based on his paper [3].

When a goal matches more than one clause, the processor running this goal forks one process for each alternative. The processes are called "OR forks".

When an OR fork occurs, all processes inherit the stack of Activation Records starting at the current AR and up to the root of the Tree of ARs. It is important to notice that in this case, the Tree of ARs is distributed (every processor keeps the part of the Tree it has developed so far in its own local memory). TCP on the other hand uses a non-distributed Tree of ARs.
Each of the processes then grows its own top to the stack in its own processor memory space.

Since Prolog is a single assignment language (i.e. when a variable is bound its value can not be changed), there is no problem with the OR Processes sharing the bottom of their stacks for reading.

However, OR Parallelism causes problems in writing to the inherited stack. Some of the variables that an OR fork inherits in the stack may be as yet uninstantiated (in TCP those are called "i_free" variables).

Unification of the head of a clause by one OR fork may bind some variable that is linked to one of these undefined variables, thus giving it a value valid only to that OR fork.

Example:

Consider the following Prolog program.

(1) p(Y) :- q(Y), r(X, Y).

(2) q(a).

(3) q(b).

(4) r(b, c).

and the query: | ?- p(Y).

In an OR Parallel environment, the following would happen.

Two OR forks are created for the unification of "q(Y)" with the heads of clauses (2) and (3).

"Y" is still undefined.

We need a mechanism that allows each OR fork to define the variable "Y" in the inherited stack and yet not see the value given to it by the competing OR fork.

Note: With AND Parallelism, an AND process actually needs to know the value given to a variable by an other AND process.

AND processes collaborate as opposed to compete.

In the example, the process for clause (2) will bind "a" to "Y". The process for clause (3) will bind "b" to "Y".

To make it possible for one uninstantiated variable to have more than one binding given to it by several OR forks, this implementation uses "Hash Table Windows". See [12] for a complete description.

Problems with OR Parallelism:

OR forks are created to do calculations that will only be needed in the case of backtracking. However, if backtracking is not needed then the OR forks are wasting resources (i.e. the processes that run the OR forks) and time (i.e. the time spent managing these resources).

OR forks should be used only when the interpreter knows they can improve the processing time. Because of its overhead, an OR fork should never be used when the clause it is testing the goal against is only a boundary condition as in the following example.

Example:

Consider the following program which inserts the element X into a sorted list to produce a new list.

```
insertx(X, [A|L], [A|NewL]) :-
        A=(X,! insertx(X, L, NewL).
insertx(X, L, [X|L]).
```

and the query: | ?- insert(3, [1,2,4], L).

When using OR Parallelism, the interpreter should not create OR forks to try the two "insertx" clauses concurrently. The second clause should be tried only when the condition is true (i.e. X < the head of the list).

One reason why OR Parallelism is not suited for TCP is the fact that TCP uses a completely shared memory. All processors use the same Tree of Activation Records.

In order to implement OR Parallelism, TCP would have to to differentiate between AND nodes and OR nodes in the Tree of ARs. For that it would probably need to use a distributed memory. It would have to solve the problem of competing OR forks by allowing them to give different bindings to an "i_free" variable. It would also have to know when to fork OR processes and when not to. A program annotation would probably be needed to that affect (i.e. additions to the syntax).

TCP uses a sophisticated form of backtracking. That is the subject of the next section.

6.3.6.3. Backtracking in TCP:

Whenever a goal fails and there is no other alternative to try it against, TCP uses backtracking. Let us start the discussion by giving an example.

Example 1: Consider the following TCP clause:

```
p :- g1(X), g2(Y, Z), g3(X, Y), g4(Y).
```

The Producer/Consumer Relationships are:

```
g1: Produces: (X),    Consumes: NIL
g2: Produces: (Y, Z), Consumes: NIL
g3: Produces: NIL,    Consumes: (X, Y)
g4: Produces: NIL,    Consumes: (Y)
```

When clause "p" is matched against goal "p", the following will happen:

- "g1" and "g2" run in parallel.
- As soon as "g1" and "g2" are done, "g3" starts.
- As soon as "g2" is done, "g4" starts.

Suppose that "g3" fails with no other alternative to try it against. TCP will backtrack.

Since "g3" consumes "X" and "Y", TCP looks for the first producer (going from right to left) of one of these variables. That goal is "g2".

TCP cancels "g2". Since "g2" produces "Y" and "Z", it also cancels all consumers of "Y" and "Z" (i.e. "g3" and "g4").

TCP resatisfies "g2".

If "g2" succeeds (with a new set of bindings for "Y" and "Z") then "g3" and "g4" are started again with the new bindings.

If "g2" fails, TCP uses right-to-left backtracking to resatisfy

"g1", for "g3" might succeed with a different value of "X"; all possible pairs of "X" and "Y" must be tried.

This execution scheme is based on the following rule [3]. "If a goal fails and there is another clause to test it against there is no problem. Otherwise the system must look further for a "choice point" goal. It can look among those goals that trigger (produce variables for) the failing goal. In resatisfying the "choice point" goal, all the goals that consume any variable produced by the "choice point" goal must be cancelled. When this method does not reveal a "choice point", then simple right-to-left backtracking must take over to make sure that all the possibilities are covered."

The idea behind right-to-left backtracking is that there might be more than one producer for the failing goal's variables. We have to try all the possible combinations of those producers.

Example 2: Consider the following TCP clause.

```
p :- g1(X, Y), g2(Z, Y), g3(Z, S), g4(Z, X), g5(S, X).
```

Suppose "g5" fails with no other alternative to test it against.

- TCP gets the producers of "g5"'s variables (i.e. "g1" and "g3").
- "g3" is the "choice point", so TCP resatisfies it.
- If "g3" fails, TCP does right-to-left backtracking. The reason for that is the following: "g2" has an affect on "g3", so it acts as a producer for "g5"'s variables (even though it is not).

The Algorithm:

The procedure in point 5. of the appendix is called every time a processor fails to satisfy a goal.

Example 3: Consider the following TCP clause.

a(A,B,C,D,E) :- b(A,B), c(B,C), d(D), e(E), f(A,C), g(D,E), h(C).

mode(- - - - -)

and the query: | ?- a(V,W,X,Y,Z).

Case 1:

suppose goal "f(A,C)" fails with no other alternative to try it against. The following will happen: (using the algorithm above)

- Get the first goal (going from right to left) that produces a variable for "f(A,C)". That goal is "c(B,C)".
- Cancel all the goals that consume variables from "c(B,C)". These goals are "f(A,C)" and "h(c)".
- Resatisfy "c(B,C)".

Suppose "c(B,C)" fails. The following will happen.

- Cancel "c(B,C)".
- Cancel the consumers of "b(A,B)"'s variables (i.e. "c(B,C)" and "f(A,C)") in case there are active.
- Resatisfy "b(A,B)".

Case 2:

Suppose goal "e(E)" fails with no other alternative to try it against. Since this goal does not consume any variables, we know that resatisfying any other goals in the clause will not make it succeed. The initial query must fail.

Note 1:

Notice in case 1, when "f(A,C)" failed, we had to kill all the consumers of "c(B,C)" which are "f(A,C)" and "h(C)". Actually, if the system could know that "f(A,C)" failed because of the variable "A" and not because of "C", it would not have to cancel "h(C)". TCP has no way of knowing which variable caused the failure of a goal. Otherwise it would be interesting to implement this feature to make backtracking more efficient.

Note 2:

In TCP, there could never be more than one backtracking point because of the same variable at the same time.
In Example 1, "g3(X,Y)" and "g4(Y)" could fail at the same time. The first one to backtrack will cancel the other.

Note 3:

There could be more than one backtracking point because of different variables at the same time. In Example 3, "f(A,C)" and "g(D,E)" could fail at the same time. The "choice points" will be respectively "c(B,C)" and "d(D)".

Note 4:

There could be one backtracking point because of different variables at the same time. In the clause

```
p :- g1(X,Y), g2(X), g3(Y).
```

"g2" and "g3" could fail at the same time. They will have the same "choice point" which is "g1".

7. Conclusions:

It is always hard to decide beween the following.

i) Having the programmer involved in the implementation issues by requesting special annotations in the program. These annotations will be used at run-time to speed up the execution.

ii) Having the interpreter extract all the information it needs from the program without any help from the programmer.

I faced this decision during three important phases of the implementation of the TCP interpreter.

1. Mode Declarations:

The interpreter needs to know if a variable in the parameter list of a goal is "ground" or not. The importance of that information was discussed in the previous chapters. In this case I opted for the first solution.

The programmer is expected to tell the interpreter (by means of Mode Declarations) which variables in the head of a clause will be ground and which ones will not. This approach was also taken by P. Borgwardt [3].

One argument to justify this choice is its simplicity. The Mode Declarations are compiled and the information they represent is stored once and for all. The interpreter has access to that information at any time in the interpretation process.

On the other hand, the second solution would work as well. The interpreter would have to get the information from the Tree of Activation Records. It is possible to check if a variable is

ground or not before the goal it appears in is activated (i.e. picked by a processor). The system would have to scan the ARs in the Tree until it finds a binding for that variable (in which case it is ground) or else it finds the AR in which the variable was defined (in which case, if it is still free, the variable is not ground).

It would be very efficient if the interpreter could use the BA (Binding Array) to look up the variable. But that is impossible because a BA is constructed only after a goal is activated. And in order to activate a goal, the system has to know if the variables in the consumer list of that goal are ground or not.

2. Producer/Consumer Solution:

As seen in the previous chapters, the notion of producer and consumer of a certain variable is vital to any interpreter of Concurrent Prolog.

Most implementations make use of the Data Flow Annotations to mark a variable as being produced by a goal or as being consumed by it.

An interpreter that uses the Data Flow Annotation will have to order the goals in the body of a clause in such a way that a goal that produces a variable is always executed before the one(s) that consumes that variable. That task can be very time consuming.

TCP assumes that: In the body of a clause, the goal that a variable appears in for the first time is the producer for that variable. Any other goal that variable appears in is the

consumer.

The Data Flow Annotation imposes on the programmer to know which goal will produce a variable and which one(s) will consume it. Instead of doing that, why does not the programmer just order the goals in the body of a clause by having the producer goal appear before the consumer goal(s)? The time of ordering the clauses will be saved at run-time.

3. The Concept of Concurrency:

Some implementations of Parallel Prolog use a special annotation to indicate which goals in the body of a clause will run in parallel and which ones will run sequentially.

In IC-PROLOG, the only time two or more goals are executed in parallel is when they are separated by "//" in the program.

Example:

Consider the following IC-PROLOG program.

```
r(a).
s(b).
u(a, b).
v(b).
p(X, Y) :- r(X) // s(Y).
q(X, Y) :- u(X, Y) & v(Y).
```

The two goals "r" and "s" will run in parallel.

The two goals "u" and "v": will run sequentially.

TCP does not use any annotations. It takes a clause with the intention of running all its goals in parallel. If there is possibility of Parallelism, it will take advantage of it.

Another feature of TCP is that a goal in the body of a clause is executed (i.e. picked by a processor) whenever it is ready and there is a processor available. The order of a goal's appearance in the body of a clause is irrelevent in this case.

Example:

Consider the following TCP program.

```
q(a, b).

r(a).

s(b).

t(c).

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

and the query: | ?- p.

The two goals "q(X, Y)" and "t(Z)" will run in parallel because they are ready (i.e. they consume no variables).

As soon as "q(X, Y)" succeeds, the two goals "r(X)" and "s(Y)" are started in parallel.

APPENDIX

1. Definition Module for the module "processes":

```
DEFINITION MODULE Processes;
 TYPE SIGNAL;

 PROCEDURE StartProcess(P:PROC; n:CARDINAL);
 (* Start a concurrent process with program P
    and workspace of size n. PROC is a standard type
    defined as PROC = PROCEDURE. *)

 PROCEDURE SEND(VAR s: SIGNAL);
 (* One process waiting for s is resumed. *)

 PROCEDURE WAIT(VAR s: SIGNAL);
 (* Wait for some other process to send s. *)

 PROCEDURE Awaited(s: SIGNAL):BOOLEAN;
 (* Awaited(s) = "at least one proces is waiting for s". *)

 PROCEDURE Init(VAR s: SIGNAL);
 (* compulsory initialization. *)

END Processes.
```

2. Producer-Consumer Problem:

```
MODULE Buffer[1];
 EXPORT deposit, fetch;
 IMPORT SIGNAL, SEND, WAIT, Init, ElementType;

 CONST N = 128; (* buffer size *)
 VAR n :[0..N]; (* number of deposited elements *)
  nonfull: SIGNAL; (* n < N *)
  nonempty : SIGNAL; (* n > N *)
  in, out: [0..N-1]; (* indices *)
  buf: ARRAY[0..N-1] OF ElementType;

 PROCEDURE deposit(x: ElementType);
 BEGIN
  IF n = N THEN WAIT(nonfull) END;
  (* n < N *) n := n+1; (* 0 < n <= N *)
  buf[in] := x; in := (in+1)MOD N;
  SEND(nonempty)
 END deposit;

 PROCEDURE fetch(VAR x: ElementType);
 BEGIN
  IF n = 0 THEN WAIT(nonempty) END;
  (* n > 0 *) n := n-1; (* 0 <= n < N *)
  x := buf[out]; out := (out+1)MOD N;
  SEND(nonfull)
 END fetch;

 BEGIN n := 0; in := 0; out := 0;
  Init(nonfull); Init(nonempty)
 END Buffer
```

3. Implementation Module for "processes":

```
IMPLEMENTATION MODULE Processes[1];

 FROM SYSTEM IMPORT ADDRESS,TSIZE,NEWPROCESS,TRANSFER;
 FROM STORAGE IMPORT Allocate;

 TYPE SIGNAL = POINTER TO ProcessDescriptor;


 ProcessDescriptor=
  RECORD next: SIGNAL; (* ring *)
  queue: SIGNAL; (* queue of waiting processes *)
  cor: ADDRESS;
  ready: BOOLEAN
 END;


 VAR cp: SIGNAL; (* current process *)


 PROCEDURE StartProcess(P: PROC; n: CARDINAL);
  VAR s0: SIGNAL; wsp: ADDRESS;
 BEGIN s0 := cp; Allocate(wsp,n)
  Allocate(cp, TSIZE(ProcessDescriptor));
  WITH cp^ DO
    next := s0^.next; s0^.next := cp;
    ready := TRUE; queue := NIL
  END;
  NEWPROCESS(P, wsp, n, cp^.cor); TRANSFER(s0^.cor, cp^.cor)
 END StartProcess;


 PROCEDURE SEND(VAR s: SIGNAL);
  VAR s0: SIGNAL;
 BEGIN
  IF s # NIL THEN
   s0 := cp; cp := s;
   WITH cp^ DO
     s := queue; ready := TRUE; queue := NIL
   END;
   TRANSFER(s0^.cor, cp^.cor)
  END
 END SEND;
```

Implementation Module for "processes" (continued):

```
PROCEDURE WAIT(VAR s : SIGNAL);
 VAR s0, s1: SIGNAL;
BEGIN (* insert cp in queue s *)
 IF s = NIL THEN s := cp
 ELSE s0 := s; s1 := s0^.queue;
  WHILE s1 # NIL DO
   s0 := s1; s1 := s0^.queue
  END;
  s0^.queue := cp
 END
 s0 := cp;
 REPEAT cp := cp^.next UNTIL cp^.ready;
 IF cp = s0 THEN (* deadlock *) HALT END;
 s0^.ready := FALSE; TRANSFER(s0^.cor, cp^.cor)
END WAIT;


PROCEDURE Awaited(s: SIGNAL): BOOLEAN;
BEGIN RETURN s # NIL
END Awaited;


PROCEDURE Init(VAR s: SIGNAL);
BEGIN s := NIL
END Init;


BEGIN Allocate(cp, TSIZE(ProcessDescriptor));
 WITH cp^ DO
  next := cp; ready := TRUE; queue := NIL
 END
END Processes.
```

4. Working Version of "processes":

```
implementation module processes[1];
        from system import address, tsize, newprocess,
             transfer, process;

        type signal = pointer to ProcessDescriptor;

        ProcessDescriptor=
           record
           next: signal;  (* ring *)
           queue: signal; (* queue of waiting processes *)
           cor: process;
           ready: boolean
        end;

        var cp: signal;     (* current process *)

        procedure StartProcess(p: proc; wsp: address);
                var s0: signal;
        begin
                s0 := cp;
                new(cp);
                with cp^ do
                   next := s0^.next;
                   s0^.next := cp;
                   ready := TRUE;
                   queue := NIL
                end;
                newprocess(p,wsp,800,cp^.cor);
                transfer(s0^.cor,cp^.cor)
        end StartProcess;

        procedure Send(var s: signal);(*Different from the original.*)
                var s0: signal;
        begin
                if s # NIL then
                   s0 := cp;
                   cp := s;
                   with cp^ do
                      ready := TRUE;
                      queue := NIL
                   end;
                   transfer(s0^.cor,cp^.cor)
                end
        end Send;
```

Working Version of "processes" (continued):

```
        procedure Wait(var s: signal);(*Different from the original.*)
                var s0, s1: signal;
        begin (* insert cp in the queue s *)

                s := cp; (* Necessary only the first time
                            a process executes a Wait.  *)
                s0 := cp;
                repeat cp := cp^.next until cp^.ready;
                if cp = s0 then
                   (* deadlock *)
                   HALT
                END;
                s0^.ready := FALSE;
                transfer(s0^.cor,cp^.cor)
        end Wait;

        procedure Awaited(s: signal): boolean;
        begin
                return s # NIL
        end Awaited;

        procedure Init(var s: signal);
        begin
                s := NIL
        end Init;

begin
        new(cp);
        with cp^ do
           next := cp;
           ready := TRUE;
           queue := NIL
        end
end processes.
```

5. Procedure "Backtrack":

```
procedure Backtrack(ProcDesc: ProcessList);
                    (*
                       ProcDesc is the Process Descriptor for
                       the goal that is being backtracked over.
                    *)

VAR     ParentDesc, (* Process Descriptor of the parent goal.    *)
        Producer,   (* Process Descriptor of the producer goal. *)
        LeftDesc: ProcessList; (* Process Descriptor left to
                                  ProcDesc.                      *)

BEGIN
        ParentDesc := ProcDesc^.Parent;
        (* Parent of the goal being backed over. *)

        if(ParentDesc = NIL) then
          (*
             The initial query has failed with no other
             alternative to try.
             ==> End of Computation.
          *)
          writef(output,"NO.0); (* Final answer is NO. *)
          HALT
        end; (* if *)

        if( not ProcDesc^.RightToLeft) then
          (*
             We are not in the process of backtracking from
             right to left.
          *)

           if( GetProducer(ProcDesc, Producer) ) then
             (*
                Get the first goal (going from right to left)
                that produces variables for the failing
                goal in ProcDesc.
             *)

             CancelConsumers(Producer);
             (*
                Cancel all the goals that consume variables
                from Producer, including ProcDesc itself.
             *)

             Resatisfy(Producer);
             (*
                Producer was Solved before, try to
                resatisfy it.
             *)
```

Procedure "Backtrack" (continued):

```
            Producer^.RightToLeft := TRUE;
            (*
               This flag indicates that this goal is to be retried.
               - If it succeeds, the flag becomes FALSE,
               - If it fails with no more alternatives to try,
                 then we would have to use left-to-right
                 backtracking starting at the goal to its
                 immediate left.
                 ( if no such goal then failure of the parent ).
               The reason for this is to explore all the
               combinations of the failing goal's producers.
               Every time a goal is Solved, RightToLeft becomes
               FALSE.
            *)

         else

            (*
               There are no producers for the failing goal
               to resatisfy (i.e. the failing goal does not
               consume any variables).
               Since ProcDesc failed without alternatives,
               resatisfying other goals will not make it succeed.
               ==>
               The parent (i.e. head of the clause) should fail.
            *)

            Cancel(ParentDesc);
            ParentDesc^.Child := NIL;
         end (* if *)

      else (* We are backtracking right to left. *)

         Cancel(ProcDesc);
         (* Cancel the failing goal. *)
         ProcDesc^.Child := NIL;
         ProcDesc^.Solved := FALSE;
         ProcDesc^.LastMatch := 0;
         (*
            LastMatch is used as follows: When a Process
            Descriptor is first created for a goal, LastMatch
            is set to zero. When the goal is activated, it will
            match a clause in the program. LastMatch gets
            the index of that clause in the program. In case
            this goal is resatisfied later, only the clauses
            with index > LastMatch are tried.
         *)
```

Procedure "Backtrack" (continued):

```
            LeftDesc := ParentDesc^.Child;
            (* The first goal in the body. *)

            if(LeftDesc = ProcDesc) then
              (*
                 There is no left hand side sibling
                 ==> Parent must fail.
              *)
              Cancel(ParentDesc);
              ParentDesc^.RightToLeft := TRUE;
              ParentDesc^.Child := NIL;
            else
              (*
                 There is a left hand side sibling
                 ==> resatisfy it.
              *)
              while( LeftDesc^.Sibling # ProcDesc ) do
                   LeftDesc := LeftDesc^.Sibling
              end; (* while *)
              LeftDesc^.RightToLeft := TRUE;
              CancelConsumers(LeftDesc);
              Resatisfy(LeftDesc)
              (* Resatisfy the goal to the left of
                 the failing goal. *)
            end (* if *)
         end (* if *)
END Backtrack;
```

REFERENCES

1. M.V.Hermenegildo, "A Restricted And-Parallel Execution Model and Abstract Machine for Prolog Programs". MCC report number: PP-104-85.

2. M.V.Hermenegildo and R.I.Nasr, "Efficient Management of Backtracking in And-Parallelism". MCC report number: AI-181-85.

3. P.Borgwardt, "Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors". 1984 IEEE International Symposium on Logic programming. Atlantic City, NJ.

4. Naoyuki Tamura and Yukio Kaneda, "Implementing Parallel Prolog on a Multiprocessor Machine". 1984 IEEE International Symposium on Logic programming. Atlantic City, NJ.

5. Shinji Umeyama and Koichiro Tamura, "A Parallel Execution of Logic Programs". 1983 ACM Conference on Computer Architecture.

6. K.B.Irani and Yi-fong Shih, "Implementation of very large Prolog-Based Knowlege Bases on Data Flow Architectures". 1984 IEEE Computer Society on Artificial Intelligence Applications.

7. G.J.Lipovski and M.V.Hermenegildo, "B-LOG: A Branch and Bound Methodology for the Parallel Execution of Logic Programs". 1985 IEEE Conference on Parallel Processing.

8. D.H.D.Warren and L.M.Pereira, "Prolog- The Language and its Implementation Compared to Lisp". SIGPLAN Notices- ACM Symposium on AI and Programming Languages. Rochester, NY.

9. M.J.Wise, "EPILOG = PROLOG + DATA FLOW: Arguments for Combining Prolog with a Data Driven Mechanism". SIGPLAN Notices, Volume 17, Number 12, December 1982.

10. D.DeGroot, "Restricted And-Parallelism". Proceedings of the International Conference on Fifth Generation Computer Systems. 1984.

11. K.L.Clark and F.G.McCabe and S.Gregory, "IC-Prolog Language Features". Logic Programming, edited by K.L.Clark and S-A.Tarnlund.

12. D.S.Warren, "Efficient Prolog Memory Management for Flexible Control Strategies". 1984 IEEE International Symposium on Logic Programming. Atlantic City, NJ .

13. N.Wirth, "Programming in Modula-2". Springer-Verlag 1985.

14. M.Bruynooghe, "The Memory Management of Prolog Implementations". Logic Programming, by K.L.Clark and S.-A.Tarnlund. Academic Press, 1982.

15. J.Cohen, "Describing Prolog by its interpretation and compilation". Communications of the acm, december 1985.

16. E.J.Joyce, "Modula-2: a seafarer's guide and shipyard manual". Addisson-Wesley 1985.

17. J.Beidler and P.Jackowitz, "Modula-2". psw publishers 1986.

18. S.Uchida, "Inference Machine: From Sequential to Parallel". 1983 conference on Computer Architecture.

19. G.Lindstrom and P.Panangaden, "Stream-Based Execution of Logic Programs". 1984 International Symposium on Logic Programming.

20. P.Palmer, "Selective Depth-First Search in Prolog". IEEE 1984 Computer Society on Artificial Intelligence.

21. Y.Lytvin, "Parallel Evolution Programming Language for Data Flow Machines". SIGPLAN Notices, Volume 17, Number 11, November 1982.

22. K.Oflazer, "Partitioning in Parallel Processing of Production Systems". 1984 Conference on Parallel Processing.

23. J.Crammond, "A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic languages". IEEE 1985 Conference on Parallel Processing.

24. K. Fuchi, "Logical Derivation of a Prolog Interpreter". Proceedings of The 1984 International Conference on Fifth Generation Computer Systems.

25. H.Nakashima, S.Tomura and K.Ueda, "What is a Variable in Prolog?". Proceedings of The 1984 International Conference on Fifth Generation Computer Systems.

26. H.Diel, "Concurrent Data Access Architecture". Proceedings of The 1984 International Conference on Fifth Generation Computer Systems.

27. J.S.Conery and D.F.Kibler, "AND Parallelism in Logic Programs". Proceedings of the 1983 IJCAI.