PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

Parallel Execution of Logic Programs based on Dataflow Concept

Ryuzo Hasegawa and Makoto Amamiya

Musashino Electrical Communication Laboratory
Nippon Telegraph and Telephone Public Corporation
3-9-11 Midoricho,Musashino-shi,Tokyo 180,Japan

ABSTRACT

This paper presents a new parallel execution model for logic programs that is based on a dataflow concept. The execution model achieves OR-parallel and AND-pipeline processing by using eager and lazy evaluation mechanisms. It is assumed that this model is implemented on a list-processing-oriented dataflow machine, which supports non-strict structure data manipulation.

The eager evaluation is used for OR-parallel forking, and the lazy evaluation is used to prevent combinatorial explosion of the OR process activations. An activation control method with a counter is also presented here. This method enables incorporation of a sophisticated combination of depth-first and breadth-first search strategies. The eager and lazy evaluation mechanisms are demonstrated through examples as being effective in parallel processing of logic programs.

1 INTRODUCTION

Dataflow machines are promising in that they can exploit the parallelism inherent in programs and effectively execute programs written in functional languages [1,2,3,4,5,6]. We have proposed a list-processing-oriented dataflow machine architecture and structure memory construction method for implementing highly parallel non-numerical processing [6,7,8]. In addition, we have previously reported on an implementation of eager and lazy evaluations on the dataflow machine, and have showed its effectiveness in performing stream-oriented processing and non-determinate processing [9,10]. This paper discusses parallel processing of logic programs, and proposes an execution control scheme using the dataflow machine.

In the execution of logic programs, a high degree of parallelism can be implemented with use of AND-parallel and OR-parallel executions. To achieve this, several types of parallel execution models for logic programs have been proposed [11,12,13,14,15]. However, such problems as process activation control and variables management remain to be solved.

From the viewpoint of variables management, AND-parallel execution has the following problems: (1) it requires a consistency check when variables are used in common; (2) a Cartesian product has to be generated for the values obtained from each atom in a clause body; and (3) it is difficult to manage variable bindings in a parallel execution environment.

On the other hand, OR-parallel execution, which corresponds to breadth-first searching of a proof tree, may lead to a combinatorial explosion in the number of processes. How to suppress explosive process activations is a key issue under restricted resource circumstances.

With the aim of overcoming these problems, we present here a parallel execution model for logic programs that is based on a dataflow control concept. In this model, an AND/OR process accepts an environment (variable bindings) tree and produces a new environment tree. Eager evaluation is used to generate the environment tree and lazy evaluation is used to stop its generation. This execution model is effective in maximally exploiting parallelism under restricted resource circumstances. In addition, it has the merits that:

(1) It facilitates easy in gathering solutions found in leaf processes because these solutions are returned to the root process in a tree-structured form;
(2) Although AND processing is serialized, executions of body atoms can be overlapped using eager evaluation, thereby achieving concurrent execution in AND processing; and
(3) Eager/lazy evaluation makes it easy to describe concurrent process control and resource management in parallel execution of logic programs.

2 EAGER AND LAZY EVALUATION

This section provides a description of eager and lazy evaluations with dataflow control. For further details regarding dataflow implementation, see [10].

Eager/lazy evaluation is a general evaluation strategy for functions or expressions. If the evaluation of expressions proceeds from inner to outermost, it is called

an eager evaluation, and if it proceeds from outer to innermost, it is called a lazy evaluation. These evaluations are closely related to the parameter passing mechanisms. Pure data-driven computation achieves eager evaluation by means of a call-by-value mechanism. Lazy evaluation, which delays argument evaluation, implements call-by-name or call-by-need parameter passing.

The non-strict (lenient) evaluation concept can be introduced into these eager and lazy evaluations. Lenient evaluation means that a function evaluation may be started even if only a subset of parameters are available. When data-driven control is combined with lenient evaluation, it is possible to overlap a function evaluation with its parameters evaluation so that the parallelism inherent in the programs can be exploited to the maximum extent. However, this also induces such problems as that resources are wasted because of unnecessary parameter evaluation and that explosive function activation may occur.

On the other hand, if demand-driven control is combined with lenient evaluation, it makes it possible to delay parameter evaluation until necessary. Thus, explosive increases in activations can be avoided.

2.1 Structure Data Manipulation

Eager and lazy evaluations are especially effective in performing operations which generate such structure data as lists. Typical examples of these types of evaluation are lenient cons [6] and lazy cons [16].

Lenient cons is an effective mechanism for implementing a non-strict eager evaluation on a dataflow machine equipped with structure memories. In lenient cons, a cons operator creates a new cell in advance, and its operand values are written into the cell when they are obtained. In lazy cons, on the other hand, the cons operator creates a new cell without evaluating its operands. Operands evaluation is initiated only when the operands value is required by a car (or cdr) operator.

Lenient cons makes it possible for a list-generating function to send out elements (cells) each time they are created. Thus, the consumer function can concurrently proceed with its execution using them.

2.2 Concurrent Process Control

Stream-oriented concurrent programs can easily be described using lenient cons. For instance, in a program written in Valid [17] to find the n-th natural number, the natural number producer "ints" and its consumer "nths" can be executed concurrently:

```
function ints(i) = cons(i, ints(i+1)).
function nths(x,n) = if n=0 then car(x)
                     else nths(cdr(x),n-1).
  nths(x,10) where { x = ints(1) }.
```

However, the program contains a problem in that the function "ints" is evaluated infinitely. This infinity problem, however, can be solved with lazy evaluation.

The process for delaying evaluation is specified by introducing a delay operator. For instance, delaying of the evaluation of expression e(x,y) is specified as delay e(x,y). If the function "ints" is defined using the delay operator, such that

function ints(i) = cons(i, delay ints(i+1)), function "ints" is evaluated only when its value is required by the cdr operation in function "nths".

In the following sections, we will focus on parallel execution of logic programs that employ eager and lazy evaluation mechanisms, and present an implementation scheme using the dataflow machine.

3 PARALLEL EXECUTION MODEL

3.1 Parallel Processing of Logic Programs

This section discusses parallel processing of logic programs. Here, a set of Horn clauses is assumed to be a program. In general, a clause is given in the form

P <- A1,A2, ... , Am. ($m \geq 0$),

where P and Ai are atoms. A1, ... , Am are connected by logical conjunction. P and Ai are called a head atom and body atom, respectively. Invocation of Ai is also called a goal. An atom has the form R(t1, ... ,tn), where R is a relation name and ti is a term. A term may be a constant, variable or function. A set of clauses is called a database.

During execution of logic programs, parallelism can be exploited by four types of parallel processing: OR-parallel processing, AND-parallel processing, parallel substitution and non-strict structure data processing.

OR-parallel processing, where clauses that have heads unifiable with an atom (OR-connectives) are executed in parallel, can be implemented by data-driven control. In OR processing, fully parallel execution of OR-connectives is possible, since they do not share variables with each other. However, an explosion in the number of activated processes may result. Therefore, in a practical case such as where available resources are restricted, how to control process activation becomes an important problem. To solve this problem, we can use the lazy evaluation mechanism. What this means is that the process activation for solution of OR-connectives is delayed until solutions are actually required.

AND-parallel processing whereby all atoms in a clause body are executed in parallel creates a problem when variables to be dealt with are shared by several body atoms. In such cases, it would be impractical to check the consistency of all combinations of solutions obtained from each body atom. Rather, one should first determine the producer/consumer relationship among the body atoms and let the

consumer check the value output from the producer. Therefore, when variables are shared by body atoms, their execution should be serialized. However, pipeline processing among body atoms is possible by using eager evaluation. This is due to the fact that solutions output from one atom can be successively passed to the next atom processing.

Parallel substitution for all terms in an atom presents the same problem as in AND-parallel processing. Therefore, this processing should also be serialized.

The main processing of logic programs includes structure data manipulation. Lenient cons makes it possible to fully exploit parallelism in the manipulation of structure data.

3.2 Dataflow Based Execution Model

From the viewpoint of dataflow implementation, two types of execution models can be considered: a procedural model and a proof tree model. The former is based on procedural semantics and the latter on resolutional semantics. In the procedural model, clauses are executed the same as in functions, based on a procedural interpretation [18]. When the language aspect is considered, clauses differ from functions in that:

(1) input/output relationships for arguments are decided during execution, and
(2) clause invocation is non-deterministic (not unique).

A possible implementation scheme with dataflow machines is to directly translate clauses into corresponding dataflow graphs and then execute them. In this case, item (1) above corresponds to the bi-directional flow of tokens in the graph, and item (2) to several candidate dataflow graphs which are selected upon clause invocation. This scheme offers an opportunity for high speed execution of logic programs. However, it induces an overhead because of dynamic token-flow control and requires a complicated facility which supports non-deterministic clause invocation [12].

Taking all this into consideration, we have adopted an interpretive execution scheme based on the proof tree model.

In the proof tree model, the logic program execution is regarded as a reduction process. In the course of execution, new goals are successively deduced from the given goal.

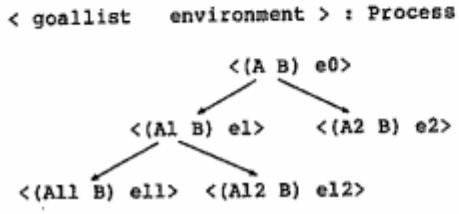

Fig. 1 Reduction-based Execution

Thus, the new goals which are generated expand the proof tree (called an AND/OR tree). A typical execution control method for a proof tree model can be outlined as follows.

A process is created to solve some given goals. In general, the process maintains a goal list to be solved, in addition to maintaining some binding information. Consider the execution process shown in Fig. 1. Given a goal list (A B) and an environment e0, the process first attempts to solve goal A. It then creates two independent processes to solve new goals A1 and A2 both deduced from A. At that time, goal B which remains to be solved is copied into each process. This reduction process will terminate either when a goal fails or when the goal list becomes empty. All solutions can be found in the leaf processes of the proof tree.

To implement this reduction process, one must:

(1) control the non-deterministic branching of a proof tree, i.e., control the initiation and termination of a process,
(2) effectively manage variable bindings, and
(3) gather solutions obtained in leaf processes.

In order to solve these problems, a dataflow-based parallel execution model is adopted. In this model, it is assumed that an interpreter written in the functional language Valid, which simulates the above reduction process, runs on the dataflow machine.

The basic idea of this execution model is as follows:

(1) Reduction-based computation can be effectively realized within the framework of functional execution using the dataflow machine. Due to the concurrent execution of functions, a high degree of parallelism can be exploited. Moreover, eager and lazy evaluation mechanisms make it easy to control process initiation and termination.
(2) It is necessary to maintain a hierarchical structure in the AND/OR tree, in order to effectively manage binding information and activated processes. To achieve this, the environments obtained by solving OR-connectives are constructed into a tree, and then passed to the next goal. This results in the environment tree as shown in Fig. 2. This environment management scheme makes it easy to gather solutions obtained separately from each process.

Note that the execution process shown in

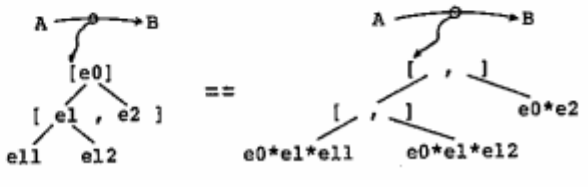

Fig. 2 Execution with environment tree

Figure 2 yields the same results as the execution process shown in Figure 1, in which solutions are scattered in a leaf processes. This is because the distributive law holds for substitution as follows:

```
[A1 B,..,An B] = [A1,..,An] B
[e1*eb,..,en*eb] = [e1,...,en]*eb
ei (i=1,..,n) : resultant environment of Ai
                deduced from A
eb            : resultant environment of B
[ .. ] represents a disjunction.
ei*ej  represents the composition of
       ei and ej.
```

Here, ei and eb represent the results of a substitution, and A1,..,An are goals deduced from A. Given a goal list (A B) and the initial environment (e0), the results obtained by solving (A B) are:

```
[e0*e1, .. , e0*en]*eb
= [e0*e1*eb, .. ,e0*en*eb].
```

3.3 Outline of The Execution Model

An outline of the above-mentioned execution model is presented in this section. In our model, it is assumed that goal lists and binding information are given in the form of list structures and then stored into structure memories for use in the list-processing-oriented dataflow machine.

For a goal atom in a clause body, a process is created to solve the OR-connectives for the goal. This process is called an OR process. For a clause head, a process is created to solve the body atoms. This process is called an AND process.

Lenient cons mechanism is used to manage binding information for variables. Here, the following terms are used to represent binding information : variable cell (V), leaf environment (LE) and environment tree (ET). Syntactically , these are defined as

```
ET := [ET, .. , ET] | LE
LE := [V, .. ,V].
```

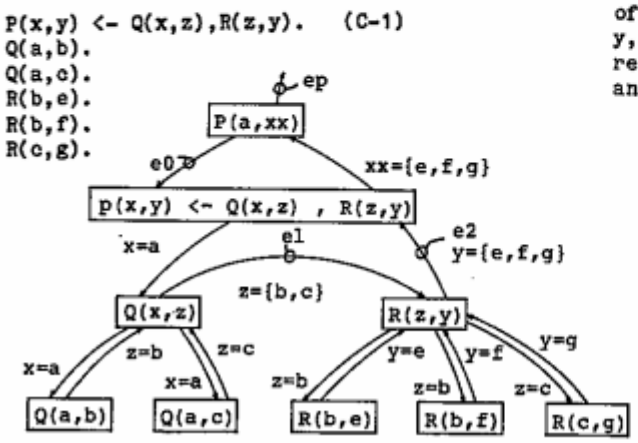

Fig. 3 AND/OR processing

A variable cell of the form of [variablename : valuefield] is used to retain a binding for a variable. The variable cell is called a defined cell if it contains a value; otherwise, it is called an undefined cell. Thus, variable cell V is either :

```
V = [x:a]  if variable x has value "a",
           otherwise
V = [x:undef].
```

A leaf environment is created each time a goal atom is unified with the head of a clause. If the unification fails, a failure environment of the form ['failure] is created. Leaf environment LE contains variable cells such as:

```
LE = [[x:a],[y:b],[z:undef]].
```

An environment tree is generated during an atom processing.

In the AND process, it is assumed that the order of execution of the body atoms is determined prior to beginning the execution of the clause body. Basically, the execution of body atoms proceeds from left to right. If a variable appears in the body atoms, the first atom in which it appears is the value producer, and the others are the value consumers. For variables appearing in the head of a clause, the defined and undefined variables are determined during unification, and corresponding defined and undefined cells are created in the leaf environment.

When an AND or OR process invokes its descendent processes, cells are created to retain the resulting environments returned from these descendent processes. That is, the resultant environments are formed into a list structure. Thus, an environment tree is created as the result of an atom processing. The environment tree represents the alternative solutions obtained by the current atom processing. The result of the final goal atom in a clause body is passed back to the parent (the caller) process.

For example, consider the program shown in Fig. 3. If P(a,xx) is given as a goal, P(a,xx) matches clause (C-1) in Figure 3 and value "a" is passed into variable x of the head P(x,y). Then Q(x,z) and R(z,y) produce a set of values {b,c} and {e,f,g} for variables z and y, respectively. Let e0, e1 and e2 be the resulting environment trees of P(x,y), Q(x,z) and R(z,y), respectively. Thus, we have

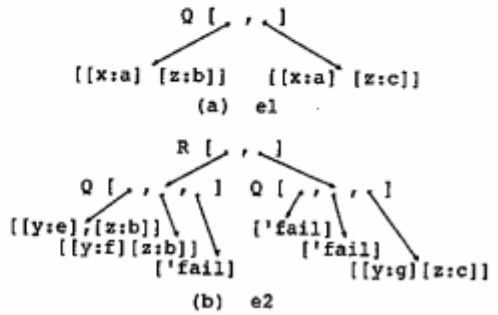

Fig. 4 Example of environment tree

```
e0 = [[x:a],[y:undef]]
e1 = [ [[x:a],[z:b]], [[x:a],[z:c]] ]
e2 = [ [ [[y:e],[z:b]], [[y:f],[z:b]] ],
       [ [[y:g],[z:c]] ] ]
   (failure environments are omitted).
```

The final environment tree e2 created in the processing of R(z,y) is returned as a new environment created by the clause body processing (Figure 4). As the result of P(a,xx), the environment tree

```
ep = [ [[xx:e], [xx:f]], [[xx:g]] ]
```

is obtained.

As mentioned above, tree-structured environments and cons operation are used instead of streams and merge operations, to retain solutions output from descendent processes. This eliminates the problems involved in non-determinate stream merging. In addition, by using a lenient cons mechanism and a divide-and-conquer algorithm for creating and traversing the environment tree, parallel and pipeline processing can be achieved.

4 IMPLEMENTATION

This section presents an implementation scheme of OR-parallel and AND-pipelined execution using the dataflow machine.

4.1 Environment Management

When a goal is unified with the head of a clause, a new leaf environment is created. This new environment consists of a binding-environment and return-environment.

The binding-environment contains the defined variable cells and is used locally in the clause. The return-environment contains the return information in the form of [uvar: rval]. Here, "uvar" is an undefined variable in the goal atom and "rval" is a term in the clause head. The return information indicates that the value of "rval" is returned as the value of "uvar".

For the current goal, the binding-environment and return-environment are called "curenv" and "currtn", respectively. For the clause head unified with the goal, they are called "newenv" and "newrtn", respectively.

In OR-parallel execution, the current environment is copied for each OR process because a variable may have several bindings. However, it is not necessary to make copies of the entire environment from the current to the root process. Only the undefined variables in the current environment should be duplicated. The defined variable cells, on the other hand, can be shared with the descendent OR processes.

Suppose we have the following goal statement and clause definition.

```
<- .. p( .. x ..) ..   goal
p( .. y .. ) <- ...    head
```

The unification rule for variables x and y is given in Table 1. The unification is serially executed from left to right, using

Table 1 Unification rule

x / y	x:a (curenv)	x:u (newrtn)	x:
y:b (newenv)	a=b ? check	u/b ** (newenv)	x/b (newrtn)
y:	y/a (newenv)	y/u (newenv)	x/y (newrtn)

Note. x, y are variables occurring respectively in a goal and a clause head.
a, b are non-variable terms (constants).
u is a variable in the clause head.
** If u is defined, check if b is equal to the value of u.

curenv, newenv and newrtn. The meaning of the notation used here is as follows:

- v:a — v is a defined variable having the value "a".
- v: — v is an undefined variable or v is the first occurrence.
- v/u — v is bound to u.
- (env) — a variable cell is to be registered in the environment "env" in parentheses.

For example, when x has a non-variable term "a" as its value and y is undefined, y is bound to the value "a" (that is, the value "a" is passed into y). If both x and y are undefined, the return information [x: y] is created in newrtn for the clause head.

Example 1.

```
<- ap([1,2,3],[4, 5],*x).
ap([*a . *b],*c,[*a . *d]) <- ap(*b,*c,*d).
```

Here, a variable is represented by an identifier beginning with "*". Let *x be an undefined variable in curenv. After unification, a newenv and newrtn are created such as:

```
curenv == [ [] ]
newenv == [ [*a: 1], [*b: [2,3]]
            [*c: [4, 5]] ]
newrtn == [ [*x: [*a . *d]] ].
```

In this case, newrtn indicates that the result of cons(1,*d) is to be returned as the value of *x.

Example 2.

```
<- p([*a . *b], *c).
p(*u,[1,2]) <- r(*u).
```

If a term is a structure containing undefined variables, the structure is copied into newenv. At that time, internal variables are generated. The environments generated after the unification are given bellow. Here, *r001 and *r002 are newly generated internal variables.

```
curenv == [ [*c: [1,2]] ]
newenv == [ [*u: [*r001 . *r002]]
newrtn == [ [*a: *r001], [*b: *r002] ]
```

In the unification, non-variable terms, whether they are atoms or structures, are passed to the descendent process. The undefined variables, on the other hand, are retained in the newrtn as return information. For undefined variables, their bindings are returned to the caller (curenv) in accordance with the return information. Therefore, there is no need for an environment pointer

(identifier) indicating where the variable bindings were introduced, and no circular environment list is created. This makes it easy and efficient to copy an environment in OR-parallel execution.

4.2 OR-parallel Interpreter

This section presents a parallel interpreter for the execution model. The kernel of the interpreter written in Valid-E [10], an extended version of Valid for symbol manipulation, is shown in Program 1. This Program is an eager evaluation version using lenient cons. To help understand the programs, a brief description of Valid-E is given in the Appendix.

4.2.1 OR Process

When an environment tree and a goal atom are given, the "ORprocess" solves the goal atom under the given environment tree. An outline of the OR process execution is shown in Fig. 5.

(1) The function "ORprocess" searches a database to find clauses with the same relational name as a goal atom, and calls the function "Activateclause". The "Activateclause" invokes the "OR1process" to solve each clause. At that time, new cells are created to retain the results obtained from each "OR1process" (F-1). The "OR1process" results are written into the cells as soon as they are obtained.

(2) The "OR1process" picks up a leaf environment out of the given environment tree and invokes the "ANDprocess" to solve the goal atom under each given environment. At that time, new cells are created to retain the "ANDprocess" results (F-2). These cells are returned before the "ANDprocess" is completed, and the "ANDprocess" results are then written into the cells as soon as they are obtained. If the environment is a failure environment of the form ['failure], the "OR1process" filters it out.

(3) As mentioned above, the results obtained by the "ORprocess" and "ANDprocess" are joined using lenient cons to generate a new environment tree. Lenient cons enables any environment to be returned whenever it is created by the "ANDprocess".

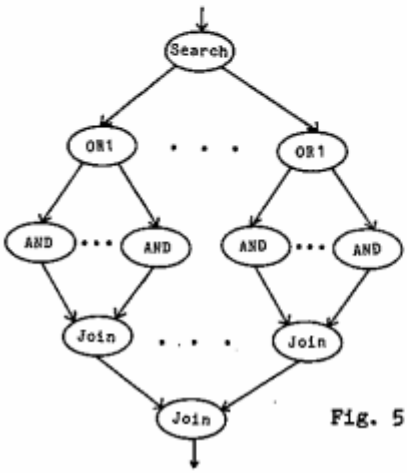

Fig. 5 OR process

4.2.2 AND Process

Given a clause head and an environment, the function "ANDprocess" solves each goal atom of the clause body in a predetermined order. An outline of AND process execution is shown in Fig. 6.

(1) The "ANDprocess" first initiates the function "Substitute". This function calls function "unify" (not shown in Program 1) to perform unification. As a result, a new binding-environment and return-environment are created. If the unification fails, a failure environment ['failure] is returned as the result of the "ANDprocess".

(2) The order of execution of atoms in a clause body is predetermined by the function "Schedule", according to the information regarding the variable bindings in the binding-environment. Several scheduling strategies must be considered. For example, it is sometimes useful to adopt a strategy where an atom which has an increased number of defined variables takes precedence. However, to support practical applications, a more heuristic scheduling method should be incorporated.

(3) When the function "AND1process" accepts the scheduled body atoms, it executes each atom of the body in a specified order. The "AND1process" activates the "ORprocess" and passes a pair consisting of the current environment tree and a goal atom to the activated "ORprocess". The next atom is then executed using the new environment tree returned from the "ORprocess", and so on.

(4) The environment tree obtained by solving the final goal in the body is sent to the "RETURNprocess". The "RETURNprocess" picks up individual environments and passes them to the "RETURN1process". For each environment, the "RETURN1process" sends variable bindings back to its caller process in accordance with the return information in the return-environment.

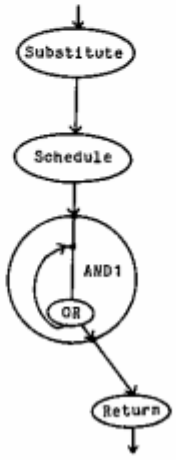

Fig. 6 AND process

Program 1

```
--  OR parallel interpreter (eager evaluation)
--  SUBSTITUTION process is omitted here.

-- ** OR process **
ORprocess(database,goal,env)
= Activateclause(clauses,goal,env)
  where{
    clauses=SearchDB(database,relname(goal)) }.

Activateclause([clause.restclauses],goal,env)
= [e1.e2]    -- (F-1)
  where{
    e1=OR1process(clause,goal,env),
    e2=Activateclause(restclauses,goal,env) }.
Activateclause([],goal,env)= [].

OR1process(clause,goal,env)
= if leafenvp(env) then
      ANDprocess(clause,goal,env)
  else [OR1process(clause,goal,e1).
        OR1process(clause,goal,e2)]    -- (F-2)
       where{ [e1.e2]=env }.
OR1process(clause,goal,[])= [].
OR1process(clause,goal,['failure])= [].

-- ** AND process **
ANDprocess([clausehead.clausebody],goal,env)
= if success then
      Returnprocess(curenv,env1,newrtn)
  else ['failure]
   where{ (success,curenv,newenv,newrtn)
            =Substitute(clausehead,goal,env),
           goals=Schedule(clausebody,newenv),
           env1=AND1process(goals,newenv) }.

AND1process([goal.restgoals],env)
= AND1process(restgoals,env1)
 where{ env1=ORprocess(DataBase,goal,env) }.
AND1process([],env)= env.

-- ** RETURN process **
Returnprocess(curenv,env,newrtn)
= if leafenvp(env) then
      Returnprocess1(curenv,env,newrtn)
  else [Returnprocess(curenv,e1,newrtn).
        Returnprocess(curenv,e2,newrtn)]
       where{ [e1.e2]=env }.
Returnprocess(curenv,[],newrtn)= [].
Returnprocess(curenv,['failure],newrtn)
= ['failure].

Returnprocess1
  (curenv,newenv,[rtn1.restrtns])
= Returnprocess1(curenv1,newenv,restrtns)
 where{ [uvar,rval]=rtn1,
        curenv1=entry(uvar,getval(rval,newenv),
                      curenv) }.
Returnprocess1(curenv,newenv,[])= curenv.

-- ** GATHER process **
Gather([e1.e2])
= if leafenvp(e1) then [e1.Gather(e2)]
  else append(Gather(e1),Gather(e2)).
Gather([['failure].e])= Gather(e).
Gather([])= [].
```

4.2.3 GATHER Process

The function "Gather" collects alternative solutions for a goal atom on the top level. It traverses the environment tree generated by the goal atom processing in order to find leaf environments. It then gathers variable bindings from each environment and outputs variable values in the goal atom. By using lenient cons, the "ORprocess", "ANDprocess" and "Gather" can be executed concurrently, thereby achieving a high degree of parallelism in logic programs.

4.3 Execution Example with Eager Evaluation

Consider the following program which defines the "append" relation:

```
append([*a . *b],*c,[*a . *d])
                  <- append(*b,*c,*d).
append([],*z,*z) <- .
```

Figure 7 is a snapshot taken during the execution of the above program using the eager evaluation mechanism. This figure exhibits the behavior of an activated "ANDprocess" for solving a find-all type of query "append(*x,*y,[1,2,3])" as a goal.

Process c1 forks into two parallel processes c2 and c3. Process c3 then forks into parallel processes c4 and c5, and so on. When a process forks into its descendent processes, new cells are created to retain the results. Due to lenient cons, these cells are returned to the parent process before the results of descendent processes are obtained. For instance, the result of c2 (*x=[],*y=[1,2,3]) can be returned immediately when it is obtained. The same may be said for process c3. Thus, the solutions can be immediately returned to "Gather" every time the solutions are obtained.

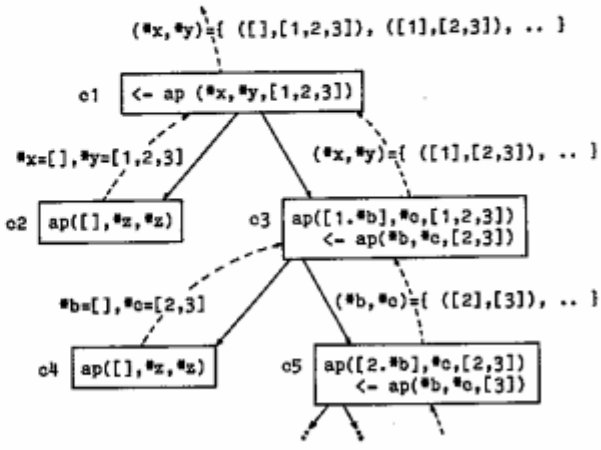

Fig. 7 Execution of "append" with eager evaluation mechanism

Program 2

```
--  OR parallel interpreter (lazy evaluation)
--  Only essential part of bounded parallel
--  control is shown here.

-- ** GATHER process **
Gather([e1.e2],count)
= if leafenvp(e1) then ([e1.s],r)
       where{ (s,r)=Gather(e2,count-1) }
  else (append(s1,s2),restenv)
    where{ restenv=case{ null(r1)->r2,
                         null(r2)->r1,
                         others->[r1.r2] },
           (s1,r1)=Gather(e1,count-1),
           (s2,r2)=Gather(e2,count) }.
Gather(['delayenv,denv],count)
= Gather(force denv,count-1).
Gather([['failure].env],count)
= Gather(env,count-1).
Gather([],count)= ([],[]).
Gather(env,0)= ([],env).

-- ** Activation control **
Activateclause
  ([clause.restclauses],goal,env,count)
= if count=0 then
    ['delayenv, delay Activateclause(
          restclauses,goal,env,Resetcount())]
  else
    [OR1process(clause,goal,env,count).
     Activateclause(
          restclauses,goal,env,count-1)].
Activateclause([],goal,env,count)= [].

-- ** OR1 process ** --
OR1process(clause,goal,env,count)
= if leafenvp(env) then
     ANDprocess(clause,goal,env,count)
  else [OR1process(clause,goal,e1,count).
        OR1process(clause,goal,e2,count)]
       where{ [e1.e2]=env }.
OR1process(clause,goal,[],count)= [].
OR1process(clause,goal,['failure],count)= [].
OR1process(clause,goal,['delayenv,denv],count)
= ['delayenv,denv].
```

5 ACTIVATION CONTROL

As mentioned before, eager evaluation contains an "explosive activation" problem. This problem derives from the fact that the function "Activateclause" in Program 1 activates all OR candidates. This activation, however, can be suppressed through the lazy evaluation mechanism which involves a counter. The essential part of an interpreter employing this mechanism is shown in Program 2. Here, the function "Activateclause" limits the number of "OR1Process" activations to a number specified by "count".

When the counter value reaches 0, evaluation of "Activateclause" is delayed and the counter is initialized to the value specified by the function "Resetcount". The unforked "OR1process" is invoked when a demand is sent to the delayed "Activateclause" from the "Gather" function at the top level (the demand is issued when the required number of solutions has not yet been obtained).

The above-mentioned execution scheme leads to realization of the bounded parallel execution, and enables a sophisticated combination of depth-first and breadth-first search executions. In practice, this means that by changing the counting scheme in a dynamic manner, we can properly use each specific search strategy according to the resource utilization circumstances.

Figure 8 shows an example of an (OR) environment tree created in the execution process, where the number of activations is limited to n. The example for n=3 is shown in Fig. 9 and the remaining unevaluated environment tree after the "Gather" has extracted several solutions is shown in Fig. 10. Figure 10 is for a case where four solutions have been obtained and a demand is about to be sent to the rest of the environment tree to enables the ascertaining of another solution.

Note that the activation count = 1 case corresponds to a depth-first search execution. If the else part of the function "Activateclause" in Program 2 is changed to

```
[OR1process(clause,goal,env,count-1).
 Activateclause(
        restclauses,goal,env,count)], (F-3)
```

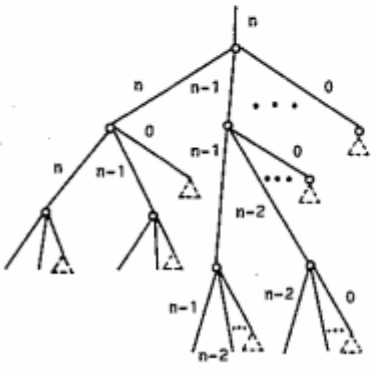

Fig. 8 Example of bounded parallel

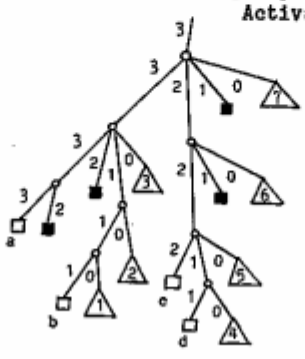

Fig. 9 Example for n=3

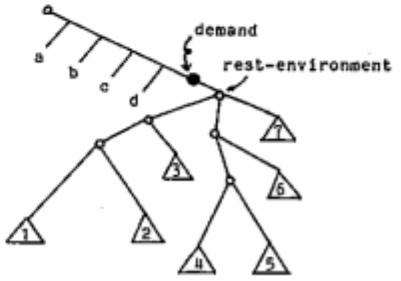

Fig. 10 Remaining OR environment tree after GATHER process

breadth-first search execution can then be achieved.

5.1 Example of Activation Control with Lazy Evaluation

Consider the following program which defines "on" and "above" relationships.

```
above(*x,*z) <- above(*x,*y),on(*y,*z).
above(*a,*b) <- on(*a,*b).
on(a,b).
on(b,c).
on(c,d).
```

For a program with a left-recursive structure, the same problem arises as in Prolog execution. For instance, the solution for the goal "above(*x,d)" cannot be obtained if the clause is sequentially executed from left to right and in a depth-first search manner as in Prolog.

On the other hand, with OR-parallel execution, solutions can be obtained because solutions for the goal "on(*a,*b)" are immediately found. If we use eager evaluation to execute this program, solutions will be found much faster. However, the execution never terminates since the process which solves the goal "above(*x,*z)" is activated infinitely. This problem can be solved using the lazy evaluation mechanism.

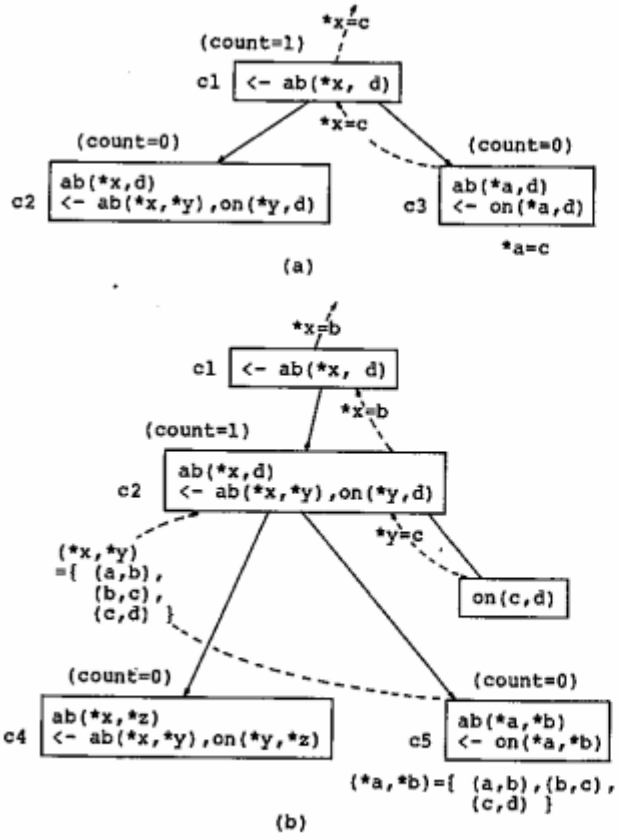

Fig. 11 Examples of activation control with lazy evaluation mechanism

Figure 11 shows some snapshots taken during execution of the above mentioned program using the lazy evaluation mechanism. The modified "Activateclause" given in (F-3) is used here to support the breadth-first search strategy and the activation count is set to 1 by the function "Resetcount".

Figure 11 (a) shows the activated "ANDprocess" for solving the goal "above(*x,d)". Processes c2 and c3 are delayed since the activation count reaches 0. Note that the count is decreased by one at the "OR1process" activation.

When a demand is sent from the "Gather" function, c2 and c3 are forced (at that time, the activation count is reset to 1), and the first *x=c solution obtained from the result of c3 is then returned to the GATHER process.

The next activation result for c2 is shown in Fig.11 (b). Another demand from the "Gather" triggers activation of processes c4 and c5, and then the second *x=b solution is obtained from c5.

6 CONCLUSION

A new execution model for parallel processing of logic programs has been presented that was based on a dataflow concept. The execution model, which embodies an implementation of a proof tree model, is well suited to dataflow implementation. The eager and lazy evaluation presented here were shown to be effective in realizing OR-parallel and AND-pipeline processing. Eager evaluation was used to enhance OR-parallel forking, while lazy evaluation was used to prevent combinatorial explosion of OR parallelism.

An activation control method has also been presented. It uses a lazy evaluation mechanism and a counter to control the number of OR process activations in accordance with resource utilization circumstances. By dynamically changing the counting scheme, it is possible to properly use both depth-first and breadth-first search strategies. However, many problems remain to be solved before this method can be effectively utilized for practical applications, because the optimal solution for deciding when which strategy should be used depends on the manner of query.

Work is now moving forward involving the study of heuristic search methods for the proof tree and pruning mechanism, where an OR-parallel and AND-pipelined interpreter are employed. The interpreter written in Valid is now running on the Valid language system Valisp [19], which is a Valid-to-Lisp translator developed on a conventional machine. This interpreter will be implemented on the dataflow machine prototype DFM now under development at the Musashino Electrical Communication Laboratory, NTT.

ACKNOWLEDGEMENT

The authors would like to thank Dr. Noriyoshi Kuroyanagi, Director of the Communication Principles Research Division at Musashino Electrical Communication Laboratory, for his guidance and constant encouragement. They also wish to thank Dr. Yasushi Kiyoki for his thoughtful comments, and to express their gratitude to the members of the Dataflow Architecture Group in the Second Research Section for discussions.

APPENDIX

A subset of Valid-E is introduced below that deals with pattern matching and lazy evaluation facilities.

(1) Patterns for lists a, b, c are:
[] == nil, [a.b] == cons(a,b),
[a,b] == [a.[b.[]]], [a,b.c] == [a.[b.c]].
Patterns may be written any place at which a value is defined.

(2) Value definition
LeftPatterns=RightPatterns
This provides for data structure selection through pattern matching. Both Leftpatterns and Rightpatterns may be a tuple of patterns. If the left and right patterns do not match, the value definition is illegal and the execution is aborted.

```
[x.y]=['a.'b]  -->  x='a, y='b,
(['a.x],['b.y],z)=(['a.'b],['b,'c],['d,'a])
      -->  x='b, y=['c], z=['d,'a],
['a,'b,x]=[y,'b,'c]
      -->  if y='a then x='c else Error.
```

(3) Function definition
FuncName(Pattern, .. ,Pattern)=Expression
Function activation is pattern-directed. Only the function with a head that succeeds in matching all argument patterns is activated.

(4) Expression
Exp where {ValueDef, .. ,ValueDef}
Value names used locally in primitive expression Exp can be defined in a "where { ... }" block.

(5) Lazy evaluation
A demand for evaluation of the delayed expression can be specified by either using a force operator as force Exp, or using implicit forcing for lazy cons. In the case of implicit forcing, primitive "head" and "tail" operators respectively trigger the evaluation of the car and cdr part of the cons cell.

REFERENCES

[1] J.B.Dennis, "Data Flow Supercomputers," IEEE Computer vol.13, no.11, 1980, pp.48-56.
[2] Arvind and V.Kathail, "A Multi-Processor Dataflow Machine That Supports Generalized Procedures," 8th Ann. Symp. Computer Architecture, 1981, pp.291-302.
[3] P.C.Treleaven, D.R.Brownbridge and R.P.Hopkins, "Data Driven and Demand Driven Computer Architecture," ACM Computing Survey, vol.14, no.1, 1982, pp.93-143.
[4] R.M.Keller, G.Lindstrom and S.S.Patil, "A Loosely-Coupled Applicative Multi-processing System," Proc. of the 1979 National Computer Conference, AFIPS, 1979, pp.613-622.
[5] J.Gurd and I.Watson, "Data Driven System for High Speed Computing, Part 1" Computer Design, June, 1980, pp.91-100.
[6] M.Amamiya, R.Hasegawa, O.Nakamura and H.Mikami, "A List-Processing-Oriented Data Flow Machine Architecture," Proc. of the 1982 National Computer Conference, AFIPS, 1982, pp.143-151.
[7] M.Amamiya, R.Hasegawa and H.Mikami, "List Processing with Data Flow Machine," Lecture Notes in Computer Science, No.147, Springer-Verlag, 1983, pp.165-190.
[8] R.Hasegawa and M.Amamiya, "Parallel List Processing using Data Flow Machine," Trans. of IECE (D), J66-D,12,pp.1400-1407, Japan 1983.
[9] M.Amamiya, R.Hasegawa and Y.Kiyoki, "Eager and Lazy Evaluation Mechanism in Data Flow Architecture and Its Application to Parallel Inference Machine," Proc. of Work Meeting for Computers, IECE, Japan, Nov., 1983.
[10] M.Amamiya and R.Hasegawa, "Dataflow Computing and Eager and Lazy Evaluation," New Generation Computing, vol. 2, No. 2, 1984.
[11] J.S.Conery and D.F.Kibler, "Parallel Interpretation of Logic Programs," Proc. of the 1981 Conference on Functional Programming Languages and Computer Architecture," pp.163-170, 1981.
[12] M.Amamiya, R.Hasegawa and M.Hashizume, "An Execution Scheme for Parallel Inference Based on a Dataflow Concept," Proc. of Work Meeting for Computers, IECE, Japan, June, 1982.
[13] M.Amamiya and R.Hasegawa, "Execution Mechanisms of Logic Programs using Data Flow Control," Proc. of The Logic Programming Conference, Sponsored by ICOT, Mar., 1983.
[14] S.Umeyama and K.Tamura, "A Parallel Execution Model of Logic Programs," The 10th Ann. Int. Symp. on Computer Architecture, pp.349-355, June, 1983.
[15] A.Ciepielewski and S.Haridi, "An Or-Parallel Token Machine," Logic Programming Workshop 83, Algarve/Portugal, also TRITA-CS-8303, Royal Institute of Technology, Stockholm, 1983.
[16] R.Hasegawa and M.Amamiya, "On the Implementation of Lazy Evaluation with Data Flow Machine," Poc. Ann. Conf. IPSJ, Japan, 1982.
[17] M.Amamiya, R.Hasegawa and Y.Ono, "Valid, A High-Level Functional Programming Language for Data Flow Machines," To appear in Review of the E.C.L, Vol.32, No.5, 1984.
[18] R.A.Kowalski, "Predicate Logic as a Programming Language," IFIP Congress, 1974, pp.569-574.
[19] R.Hasegawa and M.Amamiya, "VALID Language System: Lazy Evaluation Mechanism and Its Implementation," Proc. Meeting on Software Engineering, WGSE-84,IPSJ, Mar., 1984.