ICOT Technical Report: TR-099

TR-099

Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog

Noriyoshi Ito, Hajime Shimizu
(ICOT)
Masasuke Kishi,
Eizi Kuno and Kazuaki Rokusawa
(Oki Electric Industry Co., Ltd.)

December, 1984

Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog

Noriyoshi ITO, Hajime SHIMIZU,

Institute for New Generation Computer Technology,
1-4-28 Mita, Minato-ku, Tokyo 108.

Masasuke KISHI, Eiji KUNO, and Kazuaki ROKUSAWA

Oki Electric Industry Co., Ltd.,
4-10-12 Shibaura, Minato-ku, Tokyo 108.

ABSTRACT

Study attempts to show that our machine architecture based on the data flow model is suitable for two types of logic programming languages with different aims: one is Parallel Prolog and the other is Concurrent Prolog. The data flow model can naturally implement parallel computation, and it has close similarity to these languages. Unification and nondeterministic control, two basic functions of these languages, are represented by data flow graphs and interpreted by the machine. Several representations of variables, that facilitate the development of parallel unification and nondeterministic control mechanisms for these languages, the unification and control primitives needed to execute these languages on this architecture are presented.

1. INTRODUCTION

Recently, a number of parallel inference machines have been proposed by various institutes and researchers to be used in the execution of Prolog, a logic programming language based on first-order predicate logic. Prolog is a simple language possessing strong descriptive capabilities and intrinsic potential for parallel processing [15]. Most of these parallel Prolog interpreters focus on OR parallelism or independent AND parallelism, even if AND-parallel execution is implemented. Their target is to solve all-solution problems in a highly-parallel execution environment. Such Prolog language is called Parallel Prolog.

Concurrent Prolog [18] [19] has been proposed as one of languages enabling interactive control of AND processes. Concurrent Prolog is a successor to the Relational Language by K. Clark and S. Gregory [6], who have further extended the Relational Language to PARLOG [7]. A common feature of these languages is that they facilitate interactive communication of partial bindings, or messages, among AND processes. Of these, Concurrent Prolog has been chosen for the parallel inference machine because it provides more flexible input/output annotation than other languages.

Both Parallel and Concurrent Prolog consist of Horn clauses based on first-order predicate logic; however their aims are somewhat different. The former's aim is to find all solutions, while the latter's prime aim is object-oriented programming.

The authors have been engaged in research on a parallel inference machine and have evaluated it for Parallel Prolog programs using a detailed software simulator [13] [14]. Simulation results indicate that machine performance can be significantly improved by exploiting parallelism. The machine is based on the data flow model, which is closely related to functional languages and is naturally well suited to parallel processing [2] [4] [11]. Programs in the data flow model is represented by data flow graphs, where nodes correspond to operators and directed arcs correspond to data paths along which operands are sent. An operator in the graphs is driven by arrivals of the operands from its input arcs and outputs the result operands to its output arcs without affecting the other operators executing in parallel. This functionality of the operators has close similarity to the functional languages.

The data flow model has also similarity to the logic programming languages described above. Execution of logic programs is performed in a goal driven manner: a clause in the programs is initiated when a goal is given, and returns the results (solutions) to the goal. If multiple clauses are given, their unification with the goal can be initiated in parallel, by representing these clauses by data flow graphs.

These logic programming languages both make use of the unification operation, which is one of their basic functions. Nondeterminism is another basic feature of these languages; the control of 'don't-know nondeterminism' is required for Parallel Prolog, while the control of 'don't-care nondeterminism' is required for Concurrent Prolog [7] [17]. Details of the execution mechanisms for both languages are described in this paper.

Parallel Prolog and Concurrent Prolog are outlined in Section 2. Section 3 describes an abstract machine architecture. Section 4 discusses variables in both languages and their representation on our machine. Section 5 and 6 describe the primitive operators required to implement unification and nondeterministic control for both languages.

2. PARALLEL PROLOG AND CONCURRENT PROLOG

The Parallel Prolog programs can be interpreted in a sequential manner by using backtracking. If problems to be solved are very complex, however, it is necessary to solve the problems by exploiting these parallelism instead of using the sequential interpreters, because they may need much more time than we can stand.

A Parallel Prolog program consists of a set of clauses as shown below:

```
H1 <- B1.
H2 <- B2.
  . . .
Hn <- Bn.
```

Here, Hi and Bi denote a head literal and a body, respectively (where 1 <= i <= n). The symbol '<-' represents an implication: if its right side (the body) is satisfied, the left side (the head) is also satisfied. The body can consist of an arbitrary number of body literals connected by ANDs. A literal has the form, p(t1,t2,...,tm), where p is a predicate, and ti (1 <= i <= m) is a term. A term may be a non-structured data item such as a variable, symbol, or numeric number, or a structured data item such as a list or compound term. A clause without body literals is called a unit clause.

Unification is initiated when a set of clauses and a goal statement (a clause consisting of an arbitrary number of body literals with no head) are given. One literal is selected from the goal statement; this is called a goal literal. A clause in which the predicate of the head literal is identical with that of selected goal literal is a candidate for unification. The subset of clauses that share the same head predicate is called the definition of the predicate.

When a goal literal, G, is given, the definition of G is invoked. A clause is then selected from the definition, and unification of G and the head literal of the clause, Hi, is attempted. Generally, when multiple clauses exist in the definition, unification of G and each Hi can be executed in parallel. This parallelism among clauses is called OR parallelism, and each process executing a clause is called an OR process. A unit clause that is successfully unified with G returns the result (solution). A non-unit clause initiates the next unification treating the body as a new goal statement.

If multiple literals exist in the goal statement, they are connected by logical ANDs. That is, the goal statement is satisfied (the solutions are found for the body) only when solutions are found for all the literals and there is no inconsistency between these solutions. The literals in a goal statement can be executed in parallel. The machine can exploit this parallelism efficiently in cases where the goal literals have no shared variables, or shared variables are bound to the ground instances before invocation of these literals, because consistency checking is easy or unnecessary.

In AND parallelism, however, the search space may be expanded in some cases, or the overhead for consistency checking between the solutions returned from the literals may be increased, if these goal literals have shared variables. Therefore, syntactical operators were introduced for specifying whether sequential or parallel execution is to be used for these AND-connected literals. In this paper, the symbol '&' represents an operator that specifies sequential execution of the literals on both sides of the operator in left-to-right order. The symbol '//' represents an operator that specifies parallel execution of the literals on both sides of the operator. Processes dedicated to solving goal literals are called AND processes.

A Concurrent Prolog program is given as a set of guarded clauses, as in the following:

```
H1 <- G1 | B1.
H2 <- G2 | B2.
  . . .
Hn <- Gn | Bn.
```

Here, Hi, Bi, and the symbol '<-' are the same as in Parallel Prolog, and Gi denotes a guard (where 1 <= i <= n). The guard can consist of an arbitrary number of literals as in the body. The symbol '|' is called a guard bar or commit operator, and is regarded as one of the sequential control operators that perform exclusive control, as in Dijkstra's guarded command [10].

In the above program, the head literal and the guard are executed as in Parallel Prolog. That is, OR-parallel and AND-parallel execution can be implemented. In the OR-parallel execution environment of Concurrent Prolog programs, only one clause for which head unification and guard invocation have been successfully completed is able to proceed to the execution of the body. The results derived from other clauses are discarded at that time. This exclusive control is implemented by the guard control mechanism, as described in Section 6.

Another role of the commit operator is to make available (i.e, export) the bindings of the variables of the goal literal, G, to other processes. That is, if processes share variables, the bindings for the variables are made available to these processes by the guard control primitives. This function enables AND processes to communicate bindings or messages interactively. On the other hand, in the all-solution searching program, AND processes may be executed independently: AND processes sharing unbound variables may generate the bindings for these variables, which are then used to check consistency of their bindings.

Another feature of Concurrent Prolog is read-only annotation, in which tags are appended to variables. If an attempt is made to unify a variable having a read-only tag with a non-variable term, the unification is suspended until this variable is bound to a non-variable term by another unification. This mechanism can be implemented by adding tag bits to the memory cells used for these variables, as described in the following section.

3. ABSTRACT MACHINE ARCHITECTURE

The machine can exploit OR and AND parallelism as described above, as well as parallelism in unification. When a goal literal and its definition are given, the goal literal is unified with a head literal of each clause in the definition. This operation is called head unification. In head unification, if both literals consist of multiple arguments, or if both arguments are structured data, the unification of these arguments or their substructures can be executed in parallel. The machine is constructed from multiple processing elements and multiple structure memories interconnected by networks as shown Fig.1 [12]; it is based on the data flow model and can exploit this type of low-level parallelism.

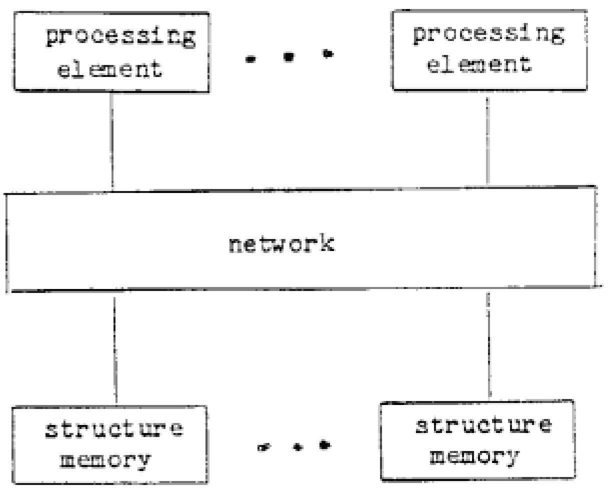
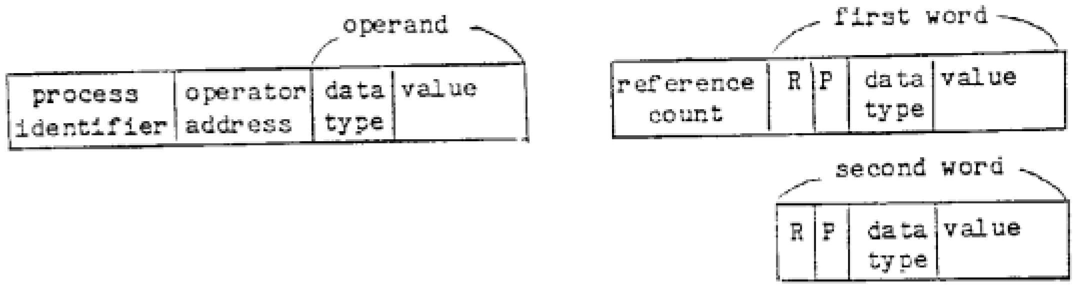


Fig.1 The abstract machine architecture

The parallel inference machine uses a tagging scheme, in which each operand has a value field and a tag field, which specifies the data type of the operand. If the operand is a structured data, the value field has a pointer to the structure memory, and the tag field is further divided into two subfields: a data type subfield, which specifies the data type of structure, such as a list or a vector; and an attribute subfield. The attribute subfield contains a non-ground flag, which indicates whether the structure has any simple variables. The attribute subfield also contains a shared flag, which indicates whether the structure has any shared-type variables (i.e., shared variables, global variables, or read-only variables). These variable types are described in detail at Section 4. The machine recognizes the tag field of the operand and transfers control to the appropriate firmware routine.

When a goal literal is given, unification is initiated by invoking the compiled clauses of its definition. The clauses are represented by data flow graphs, which correspond to the machine language of the parallel inference machine. Each node and each directed arc in a data flow graph respectively corresponds to an operator and a data path along which an operand is sent. A node in the graph can be executed when operands have arrived from all its input arcs. The machine is based on the unfolding interpreter [4], which provides procedure invocation primitives that allocate a unique identifier to each procedure instance and maintain the history of the procedure invocations. The identifier, as well as an operator address, is added to each token carrying arguments (operands), and is used to distinguish among activities (operators). All the operators, therefore, can be executed independently. The token format is shown in Fig.2 (a).



(a) Token Format

(b) Structure Memory Cell Format

Fig.2 The token and structure memory cell format

Structured data is stored in and distributed to the structure memories and is shared among processing elements instead of being locally copied to each processing element; the contents of the structured data can be referred to on demand. One advantage of this method is that it minimizes the overhead caused by copying of the structured data. The other is that it eliminates redundant storage for locally copied structured data. There may prove to be significant advantages in the manipulation of complexly structured data such as is required in natural language processing applications.

A problem in sharing structured data among processes is latency in accessing the structured data. Latency may be increased as the number of processing elements increases. In order to exploit parallelism, the processing element must issue multiple memory requests without waiting for responses. In such an environment, requests and responses must be managed by their identifiers since responses may not be returned to the processing element in the order in which the requests were issued. The data flow model implements this type of control in a natural way, and exploits low-level parallelism because it is assured of independence among operations or instructions [3] [5].

Structured data is stored in structure memory cells. Each cell consists of two words and each word has a data type tag field, a value field, and two flags: R (ready) flag and P (pending) flag. These flags are used for providing asynchronous communication between read and write operations to the memory word. R flag shows whether or not the contents of the data type tag and value fields are valid (i.e., whether or not a write operation to the memory cell is performed). P flag shows whether or not some read operations are linked to the value field. A read operation to a memory word, whose R tag is OFF, is suspended and linked to the memory word until a write operation to the memory word is issued. In order to implement a garbage collection of the structure memories, the reference count tags are appended to every memory cell [1] [8] [16], which also used for stream control as described in Section 5. Fig.2 (b) shows a structure memory cell format.

4. VARIABLES AND THEIR REPRESENTATION

A variable can be unified with, and bound to, any type of term. When a variable has multiple occurrences in a clause, however, the same term must be bound to each occurrence of the variable. If a variable is shared among unification processes being executed in parallel, it is called a shared variable. It is called a simple variable if it appears in a clause only once, or if unification affecting the variable is executed sequentially even if it appears more than once. Shared variables are distinguished from simple variables by their data type.

When a simple variable is unified with any term, the variable can be directly replaced by the term and no bindings are generated. However, when a shared variable is unified with a term other than a simple variable, the substitution information for the shared variable is output by unification. This substitution information is called the binding environment of the shared variable. The binding environment, represented as a list of shared variables and their instances, is used to check whether the bindings of the shared variables are consistent among the parallel unification operations. It is also used to substitute terms for shared variables.

Shared variables are created dynamically and represented by unique identifiers in Parallel Prolog. A shared variable in the unification of the head and guard of a guarded clause (i.e., a clause in Concurrent Prolog) is treated in the same manner as in Parallel Prolog. However, when a goal literal is given, only one clause can commit the binding information for the shared variables. Thus, shared variables can be represented by pointers to global memory cells. We call such shared variables global shared variables, or simply global variables.

The commit operator can be regarded as a write command to the memory cells pointed to by the global variables, and the read-only tag can be regarded as a read command to the memory cell. A variable with a read-only tag is called a read-only global variable, or simply a read-only variable. It is also represented as a pointer to the same memory cell to which the global variable points.

If the guard has multiple literals with global variables and these literals are executed in parallel, child processes invoked from these guard literals may communicate bindings via the global variables. Bindings are localized in this guard, i.e., information about these bindings is unavailable to the parent clause that invoked this guard until the guard is successfully terminated (i.e., the commit operation is issued). The guard acts like a mirror in which bindings are reflected from one child to another.

On the other hand, the children invoked from the body must "see" the bindings of their ancestors through the body. That is, the body acts like a window through which the bindings can be "viewed" by the children.

The relationship between a parent clause and its invoked children is illustrated in Fig.3. The two ovals in the figure correspond to the guard and body of the clause; the left oval represents a guard, the right one represents a body, and the junction of these ovals corresponds to a commit operator. The guard and body are executed from left to right. The bottom surface of the guard reflects bindings from its children, but the body allows the bindings to pass through.

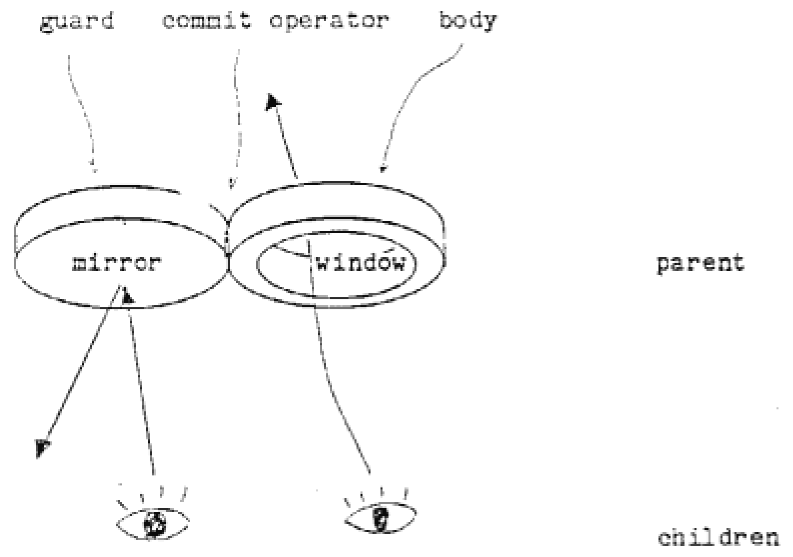


Fig.3 Relationship between a guarded clause and its children

Fig.4 depicts this relationship when the following clauses are given:

```
p0(X) <- q1(Y) // q2(Y) | ...
q1(Z) <- r1(Z) // r2(Z) | r3(Z) ...
r1(W) <- s1(W) | ...
r2(U) <- ...
```

In the above example, it is assumed that the clause with head predicate p0 (clause p0) is initially called. Because two parallel guard literals q1(Y) and q2(Y) of clause p0 share the variable Y, and Y is unbound before these literals are called, a new global variable, Yg, is created. Yg is sent as an argument of clauses q1 and q2, which are executed in parallel and share the global variable Yg. Clause q1 invokes clauses r1 and r2 from its guard; it also invokes clause r3 from its body if the invoked clauses r1 and r2 succeed. The binding for Yg is hidden from the clauses r1 and r2, and transparent to the clause r3.

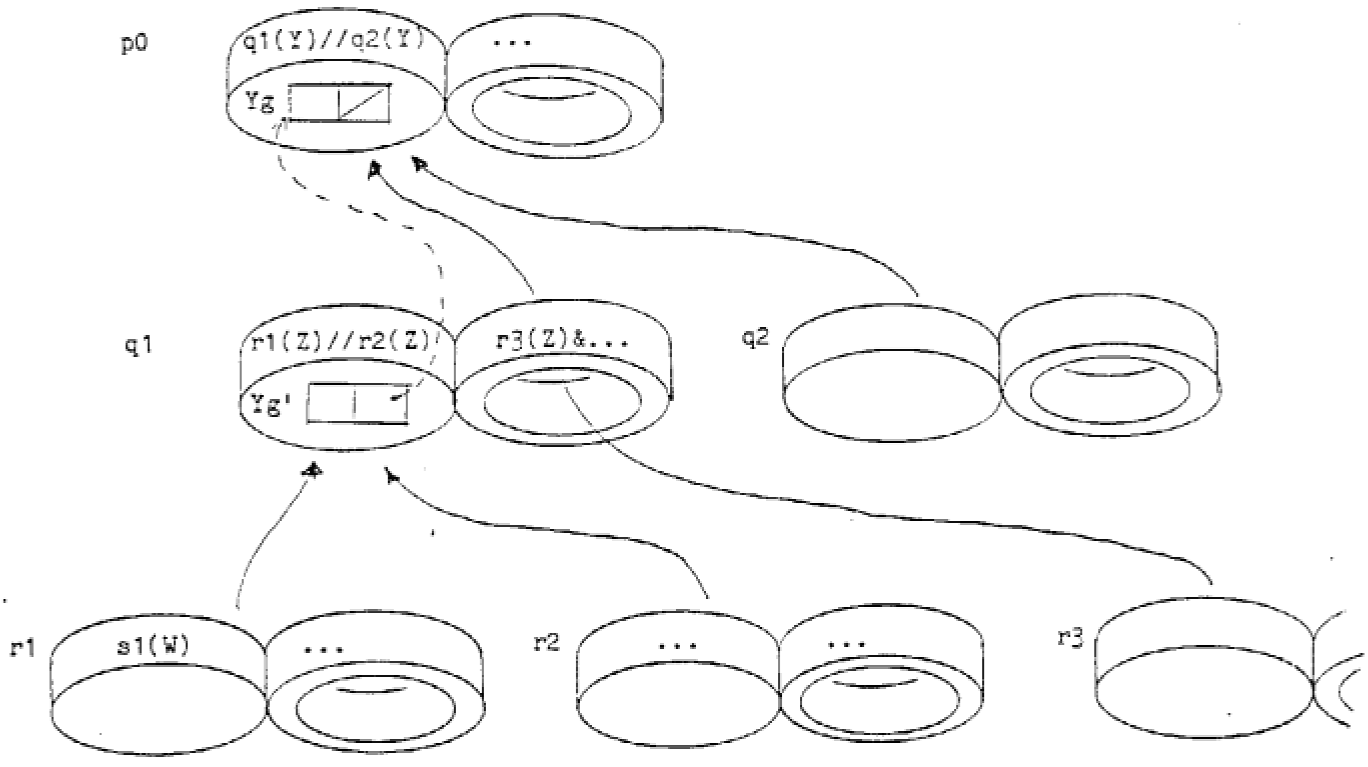


Fig.4 Relationships among guarded clauses

In order to hide the bindings of guard literals, local copies of global variables included in a guard must be created before the guard is invoked. We call this method local copying of shared variables. A local copy of Yg, which is represented by Yg', is created when clauses r1 and r2 are invoked from the guard of clause p1, as shown in Fig.4. The local copy can be represented as a memory cell having two words: (1) a slot for a local instance of the global variable and (2) a pointer to its parent guard, which is used to send the local instance to the original global variables in the parent when the commit operator is issued. The latter may be regarded as the binding environment for the locally copied variable.

However, if it is guaranteed that all the clauses invoked by a guard literal does not affect the arguments of the guard literal (i.e., if all the clauses does not bind any instance to the arguments), it is unnecessary to create local copies of global variables.

5. PRIMITIVES FOR PARALLEL PROLOG

This section describes the basic unification primitives, nondeterministic control primitives, and AND sequential/parallel control mechanisms for goal literals in Parallel Prolog.

5.1 Basic Unification Primitive Operators

The operators in the invoked clauses are initiated by passing goal arguments via the procedure invocation primitives; unification of the goal arguments is then attempted with the corresponding arguments in the head literals. Unification between an argument in the goal literal and the corresponding argument in the head literal is executed by a unify operator. If both literals have multiple arguments, multiple unify operators are executed in parallel. A unify operator has two input ports and two output ports: I (instance) port and E (environment) port. The I port is for an instance common to two input arguments and the E port is for the binding environment of shared variables.

Fig.5 shows a data flow graph representation of the unify primitive operator. In Fig.5, diamonds represent test operators for input operands, which generate boolean values. These in turn are sent to the right-hand ports of switch operators. A switch operator switches an input operand from its upper input port to one of its output ports by the boolean value: if the boolean value is "true", the input operands is sent to the T (true) port; if not, it is sent to the F (fail) port.

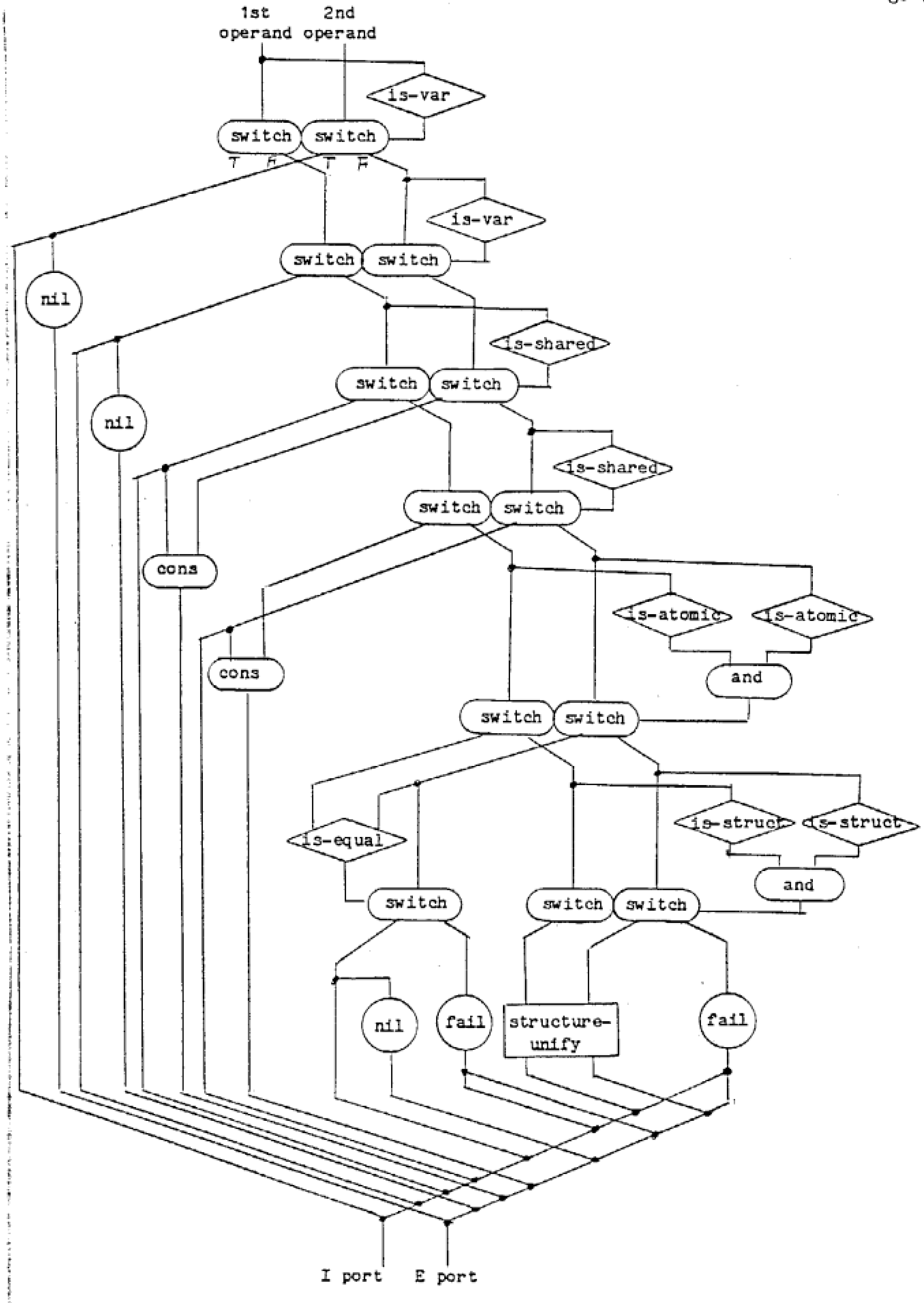


Fig.5 A data flow graph representation of a unify operator

The unify operator executes the following sequence:

(1) If an input operand is a simple variable, the operator outputs another operand to the I port and the special symbol 'nil' to the E port, which means there are no bindings for shared variables.

(2) If one is a shared variable and the other is a term other than a simple variable, the operator outputs the shared variable to the I port and outputs a newly constructed list consisting of the shared variable and its instance (i.e. the term) to the E port.

(3) If both operands are atomic terms (i.e., symbols or numeric numbers) and if their unification succeeds (i.e., if both terms are same), the operator outputs one of the terms to the I port and 'nil' to the E port; it outputs special 'fail' symbols to its two output ports, if the unification fails.

(4) If both operands are structured data, a structure-unify procedure, shown by the rectangular box in Fig.5, is invoked. This procedure decomposes two input structures into their substructures and recursively invokes unify operators for these substructures. In this procedure, all the outputs from the E ports of the unify operators are used to check consistency, and all the outputs from the I ports are used to reconstruct a new instance. In order to reduce overhead in reconstructing a new structured data, if one of the operands is a ground term (i.e., its non-ground flag and shared flag are OFF), the procedure outputs the ground term itself.

(5) In all other cases, the operator outputs 'fail' to both output ports.

The machine can interpret this graph directly. However, as the granularity of operators seems to be too fine, communication overhead between the operators is too great. In order to execute this primitive faster, most of the above graph is interpreted by hardware or firmware routines. As mentioned in Section 3, since structured data is shared among the processing elements and distributed to the structure memories, the structure access primitives must wait for the responses from the structure memories. The unify primitive, therefore, has additional output ports, represented by broken lines in Fig.6, which are used for token passing to the structure-unify procedure, only when both input operands are structured data.

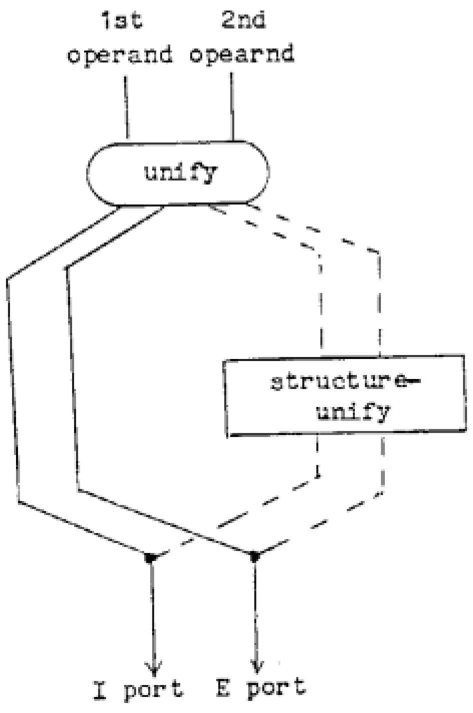


Fig.6 The actual implememtation of the unify operator

Fig.7 shows an example of a compiled code for the following clause:

p([Y,b],Z,c) <- ...

where a pair of square brackets denotes a list, and symbol b and c denotes atomic (or constant) values. [Y,b] represents a list whose left component is a variable Y and whose right component is a list [b]: it can be also represented as [Y|[b|[]]], where '|' denotes a list construct operator and '[]' denotes a nil list.

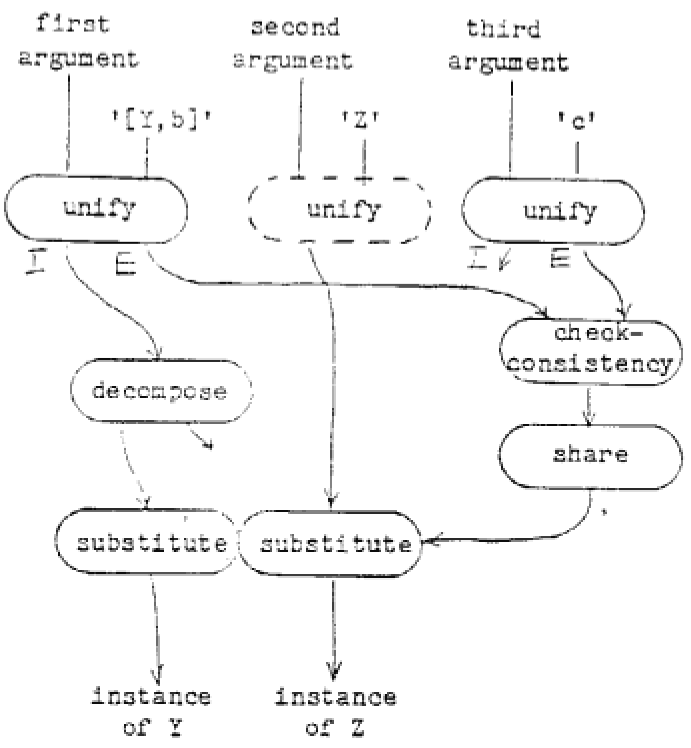


Fig.7 Data flow graph of clause p([Y,b],Z,c) <- ...

A unify operator is provided for each of the three arguments of head predicate p; these are executed in parallel. A variable can be unified with any term and the unify operator outputs the term itself for the common instance. Therefore, when the clause head argument is a variable, as in the second argument in the example, the unify operator can be omitted, as shown by the broken lines in the figure.

Two outputs from the E ports of the unify operators are sent to a check-consistency operator. The check-consistency operator checks for consistency among the binding environments of the shared variables. This operator generates a new binding environment for the shared variables if their bindings are consistent, and the symbol 'fail' if they are not. In this consistency checking, if substitution information for the shared variable is included in both its two input environments, the check-consistency operator calls a unify operator; the invoked unify operator tries to unify the two instances bound to the same shared variable. A new common instance obtained by this unify operator is output in the form of a binding environment, as described above. If simple variables are included in the new instance, they are changed to shared variables. This is done by a share operator.

In the above example, the head literal has two variables, Y and Z: Y is the left list component of the first argument and Z is the second argument. The instance of Y is obtained by the left output from the decompose-list operator, whose input is the I port output of the first unify operator. The instance of Z is directly obtained from the second goal argument. If these variables appear in the body of the clause, their instances are sent to the body; they are also used for constructing a solution to be returned to the goal, as described in Subsection 5.4. As in the third argument of the above example, if the head argument is a constant value, the I output of the unify operator is discarded.

If these instances have shared variables and if the check-consistency operator generates bindings for these shared variables, these instances are substituted for the shared variables. This substitution is executed by substitute operators. A substitute operator tests the binding environment; if it is 'fail' the operator outputs 'fail', and if 'nil', the operator outputs the instance directly; otherwise, the operator tests the shared flag of the instance and replaces the instance with binding environment, if the instance is a shared variable or its shared flag is ON (i.e., if the instance has any shared-type variables). The following subsection gives some examples of usage of these primitives.

5.2 Some Examples of Head Unification

If the following goal literal is given to the above data flow graph:

```
<- p([a|X],b,c).
```

all the binding environments from the unify operators are set to 'nil' or 'fail', since there is no shared variable in the goal and head literals. The check-consistency operator, thus outputs 'nil' or 'fail' to the substitute operators. The substitute operators then simply pass the common instances from the unify operators or outputs 'fail', according to the binding environment. These outputs constitute the result of the head unification of the clause.

The following is another example in which a goal literal including shared variable is given:

<- p([a|X],X,c) & ...

where variable X occurs twice in the literal. In this case, the compiled code of the goal invocation is as shown in Fig.8.

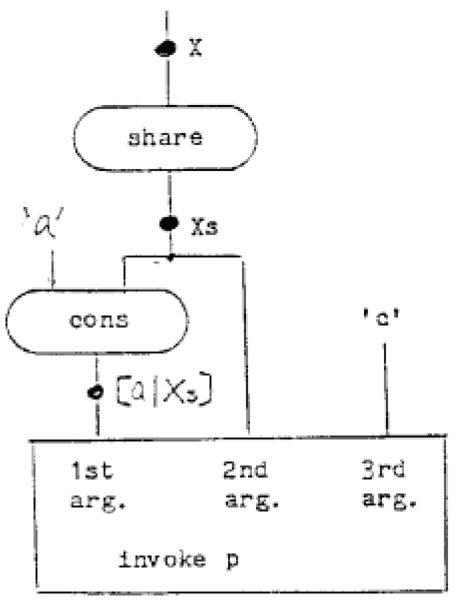


Fig.8 Data flow graph of goal <- p([a|X],X,c) & ...

The share operator tests the non-ground flag of the instance of X and, if the non-ground flag is ON, it changes all the simple variables in the instance into the shared variables; if not, it outputs the instance itself. If X is unbound before calling p([a|X],X,c) as in the above example, X is changed to a shared variable, Xs. The subscript 's' indicates that the variable is shared. Xs constitutes the first and second arguments of the goal literal. These arguments are sent to the head literals of the clauses in the definition. Assume that the same clause as shown in Fig.7 is given in the definition. In head unification, the unify operator of the first argument generates a binding environment [(Xs = [b])], which shows the instance of Xs is [b], and the unify operator of the last argument generates a binding environment 'nil'. These two environments are sent to the check-consistency operator, which outputs the final environment [(Xs = [b])]. The substitute operators replace the instances from the unify operators according to this final environment.

If another clause, such as the following, is given for the same goal literal:

```
        p([Y,b|Z],[b,c|W],c) <- ...
```

three unify operators are used for three head arguments. The binding environments from these unify operators become [(Xs = [b|Z])], [(Xs = [b,c|W])], and 'nil'. The result of consistency checking will output the final environment [(Xs = [b,c|W])], which is sent to the share operator and is then changed to [(Xs = [b,c|Ws])], as described above. The final instances of three arguments, therefore, become [a,b,c|Ws], [b,c|Ws], and c, where the first and second instances share the variable, Ws.

When a variable occurs two or more times in the head literal, the compiler can also generate codes described above: the codes in which share operators are executed before head unify operators are executed. However, in order to reduce the overhead of consistency checking, lazy execution of the share operator can be implemented as shown in the following example. Assume that a goal literal and a clause are given:

```
                  <- p([a|X],[W],c) & ...
        p([Y,Z],Z,c) <- ...
```

A compiled graph of the clause and tokens on the arcs are shown in Fig.9. In this case, instead of updating the variable Z appearing in the clause head to a shared variable Zs before executing the unify operators, a unify operator is executed to assure that the multiple occurrences of Z are bound to the same instance. That is, both instances of Z (the first one is obtained from the unify operator of the first head argument and the second one is directly obtained from the second goal argument) are unified again by another unify operator.

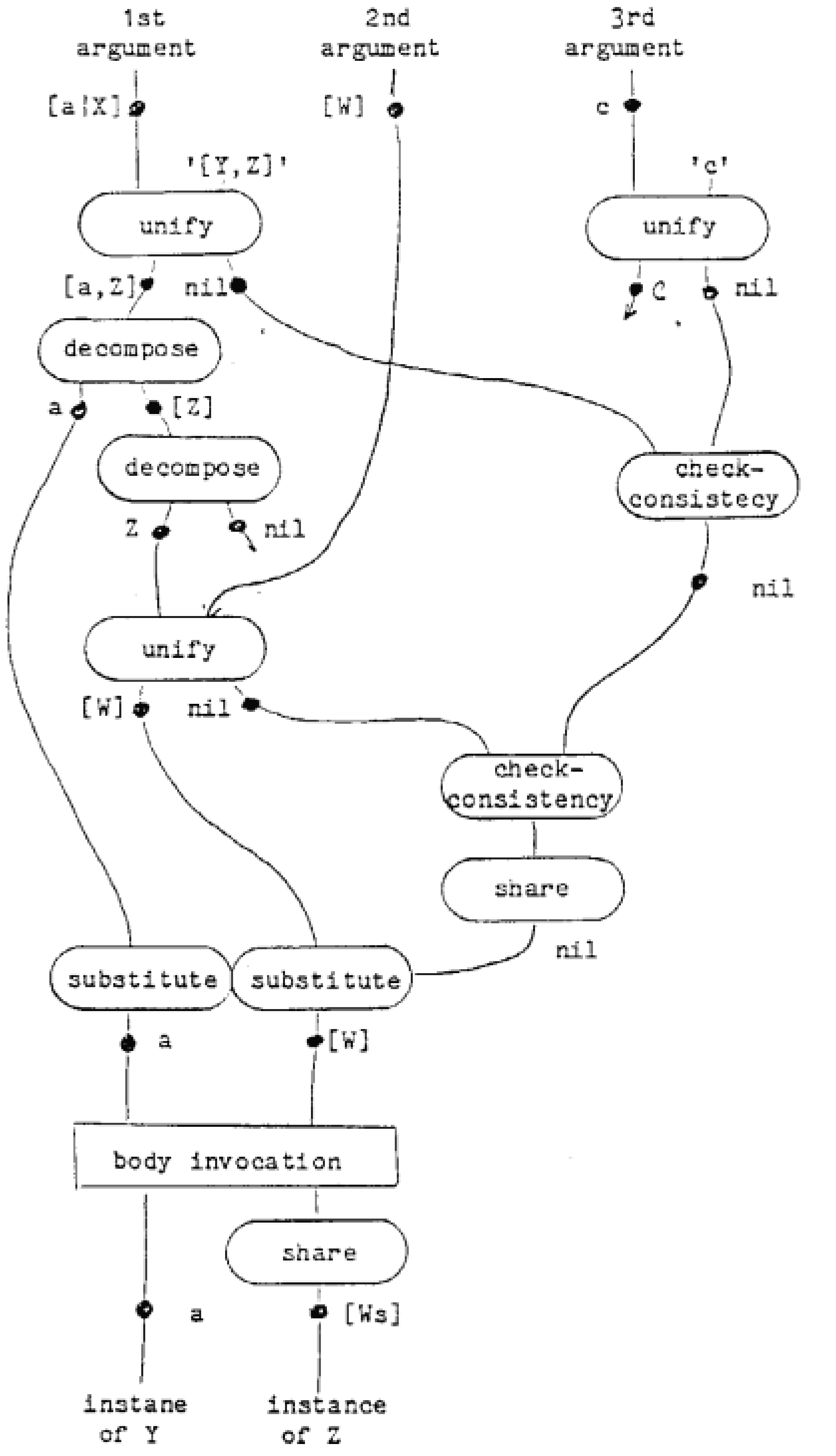


Fig.9 Data flow graph of clause p([Y,Z],Z,c) <- ... and its tokens

This final unify operator produces the list [W] as an instance of Z; the list [W] is then sent to the share operator and changed to [Ws], before it is returned to the goal. The goal literal will receive this shared structure as the instances of its variables X and W.

If there is a body in the clause and the instance of Z is used in the body literal, the variable W may be bound to another term by calling the body literal. If this share operator is executed after the body literal is invoked instead of just after head unification, the body unification treats the variable as a simple variable. Therefore, the unify operators in this unification do not produce the binding environment of Ws. This lazy execution of the share operator reduces the overhead of consistency checking.

5.3 Nondeterministic Merge Primitives

If unification succeeds and one of the solutions is obtained, it is returned to the goal statement. A solution is a list constructed from final instances of the head arguments followed by a binding environment, or 'fail'; a construct operator of the solution tests all of its operands, and if all of them are not 'fail' it generates a new list; otherwise, it returns 'fail'. In the OR-parallel environment, multiple solutions may be obtained in a nondeterministic manner. That is, solutions may be returned to the goal in the order in which they are obtained. This nondeterminism is called 'don't-know nondeterminism'. We introduced a non-strict data structure called a stream to implement this nondeterministic control. Solutions are merged into a stream by stream merge primitives, as shown in Fig.10 [4] [12].

When a goal literal calls its definition, an empty stream is created by a create-stream operator, which generates a stream descriptor as shown in Fig.11. A stream descriptor cell consists of two pointers: Stream Head Pointer (SHP) and a Stream Tail Pointer (STP). R (ready) and P (pending) flags of the cell words to be stored these pointers are initialized to OFF (empty). The pointer to the stream descriptor is returned immediately to the goal, the consumer of the stream, which in turn reads the contents of SHP by this pointer if the R flag of SHP is ON. The goal waits for a solution (i.e., sets its P flag ON and chains the read request to SHP) if the stream is still empty (i.e., if its R flag is OFF).

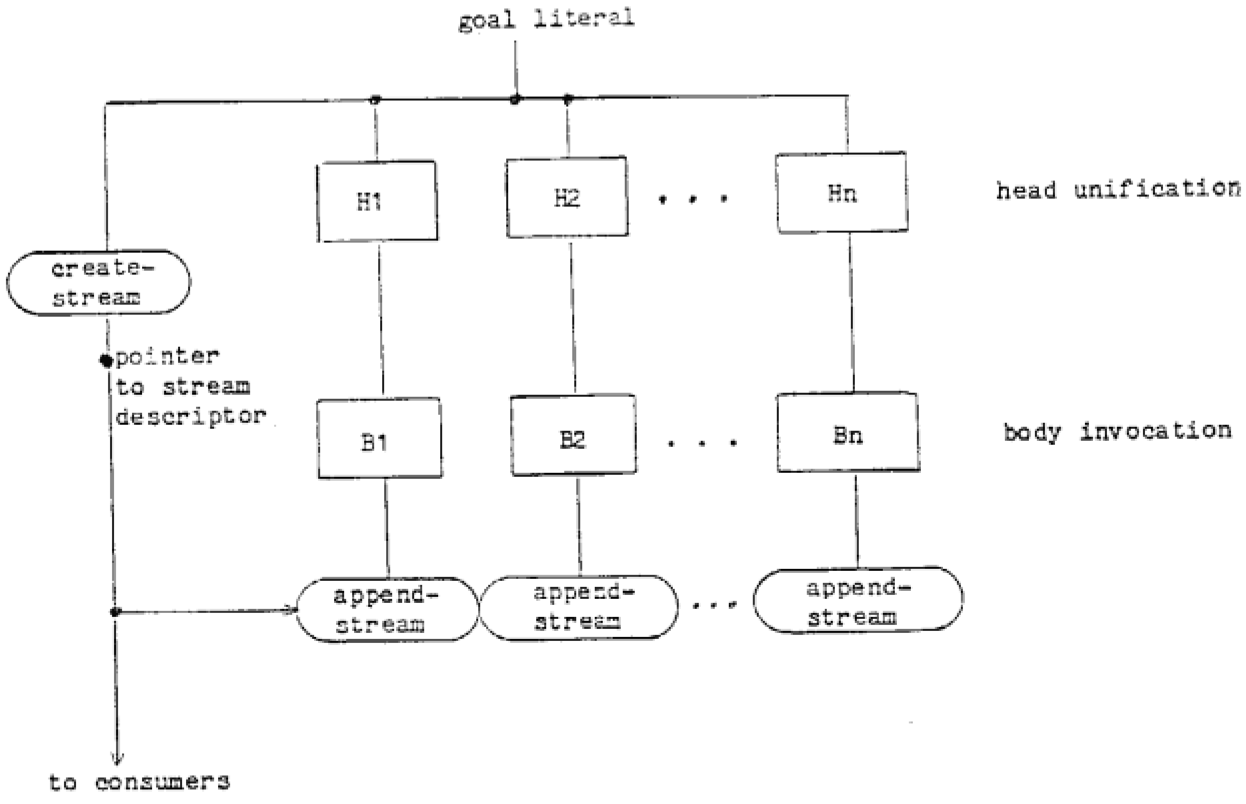


Fig.10 Stream merging primitives

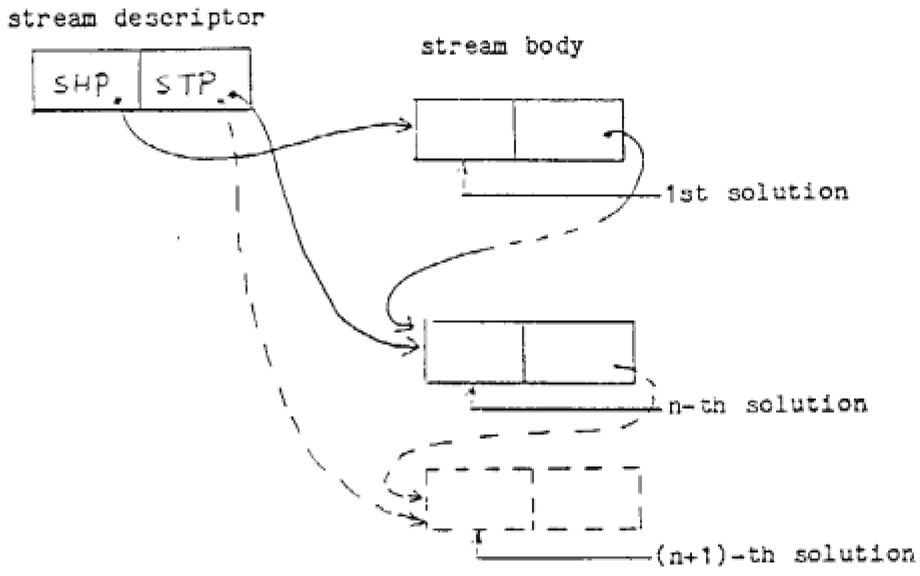


Fig.11 Representation of a stream

The pointer to the descriptor is also shared among OR processes, the producers of the stream. Each OR process appends a new solution by an append-stream operator. The append-stream operator allocates a new stream body cell, which is a new tail cell of the stream body, and writes the solution to the first word of the cell. The operator then updates the contents of STP to point to the new cell by testing its R flag: when it is OFF (i.e., when the solution to be appended to the stream is a first one), the operator writes the new cell address into STP and SHP, and sets their R flags ON (if the P flag of SHP is ON, the consumers' read requests linked to SHP are activated before this write operation); when the R flag of STP is already ON, the append-stream operator reads the contents of STP, which points to the current tail of the stream body, and updates it to point to the newly allocated stream body cell. In order to lock the stream descriptor from other append-stream operators while STP is being updated, the processing element executing append-stream operator sends an uninterruptable command, which contains the stream descriptor address and new tail cell address, to the structure memory pointed by the stream descriptor. The structure memory then performs this read-and-write cycles without interruption from other memory operations. Finally, the operator writes the new cell address into the second word of the current tail cell, previously pointed to by the STP, also by testing its P flag: if it is ON, the suspended read requests are activated. Thus, every second word of the stream body cells points to the rest of the stream. This update is indicated by the broken line in Fig.11.

A failed OR process does not affect the stream; the append-stream operator checks whether the solution is 'fail', before appending the solution.

In order to signal the goal statement that all the OR processes have terminated, the stream descriptor has a reference count of active OR processes. The reference count is initialized to the number of OR clauses invoked. It is decremented by one each time an append-stream operator is executed; it is incremented by the number of the newly created OR processes each time an AND process calls its body literal.

If the reference count reaches zero by decrementing, the append-stream operator writes 'fail', which signals the end-of-stream, into the second word of the cell pointed to by STP, and the descriptor becomes a garbage cell. If all the OR processes fail (i.e, if the stream is still empty when the reference count is zero), the SHP is set to 'fail'; otherwise, it is set to the pointer to the first word in the stream body. A waiting goal literal reads SHP, accesses the stream body cell, and decomposes it into a solution and the rest of the stream body. This stream reading operation can be executed recursively, until the rest of the stream becomes 'fail'.

5.4 Execution of AND Literals

As a solution consists of instances for goal literal arguments and a binding environment, AND-sequential and AND-parallel execution can be achieved. In this subsection, both execution mechanisms will be shown.

(1) AND-sequential execution

Assume the following clause is given in the definition of predicate p:

```
p([a|X],Y) <- q(X,Z) & r(Z,Y).
```

In this example, two body literals q(X,Z) and r(Z,Y) are executed sequentially. Connection paths between the instances of the variables are shown as in Fig.12.

In this figure, Eh is the created binding environment by head unification, Eq is the environment returned from the definition of q, Er is the environment returned from the definition of r, and Ep is the final environment of the clause p to be returned to the goal statement. The additional literal, apply-append([a|X],Y,Ep), represents a creation of the solution and execution of the append-stream operation described above. The solution to be returned is a list of final instances for the arguments in the head literal followed by Ep. The input/output modes for the literal arguments can be specified in the compiler. When these modes are specified, the arguments to be returned can be a subset of those contained in the clause head to improve performance.

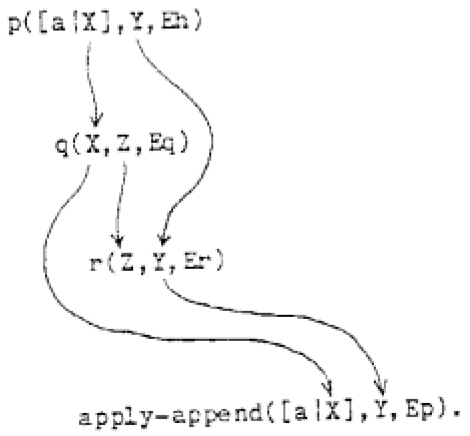


Fig.12 Connection paths of clause
p([a|X],Y,Eh) <- q(X,Z,Eq) & r(Z,Y,Er).

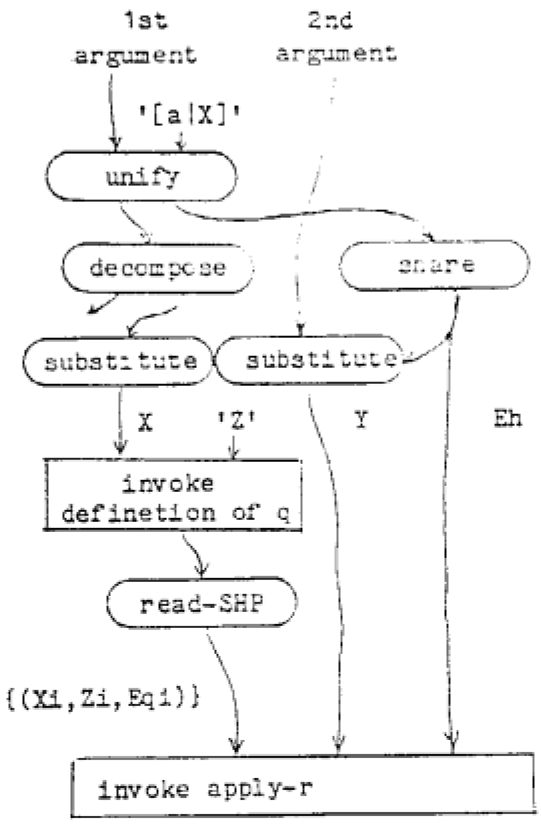
The compiler generates complete data flow graph procedures, which is shown as Fig.13 (a), (b), and (c), according to the connection paths shown in Fig.12. When a goal literal, whose predicate is p, is given, its arguments are passed to head unification of the clause shown by Fig.13 (a). The instances of variables X and Y are obtained if head unification succeeds. Succeeded head unification invokes the definition of the first body literal q(X,Z), whose first argument is the obtained instance X and second instance is an unbound variable Z. If head unification fails, the invocation of q(X,Z) is suppressed: the procedure invocation primitives do not invoke the procedure and decrement the reference count of the stream descriptor if its input argument is 'fail'. The instance of Y, which is not used in q(X,Z), is sent to the next stage by bypassing this literal.

The invocation of the literal q(X,Z) may return the stream of solutions {(Xi,Zi,Eqi)}, where Xi and Zi are i-th instances of X and Z, respectively, and Eqi is a i-th environment obtained. The recursive procedure apply-r shown as Fig.13 (b) is then invoked, reads the stream, and divides it into the first solution (X1,Z1,Eq1) and the rest of the stream {(Xi,Zi,Eqi)}, which is used as the argument of the recursive call of the apply-r (where, i = 2,3,...). If the stream is still empty, the read request is suspended as described in Subsection 5.3. When no successful solutions are obtained, or when the rest of stream is 'fail' (i.e., no more solutions are exist), the invocation of the procedure is suppressed.

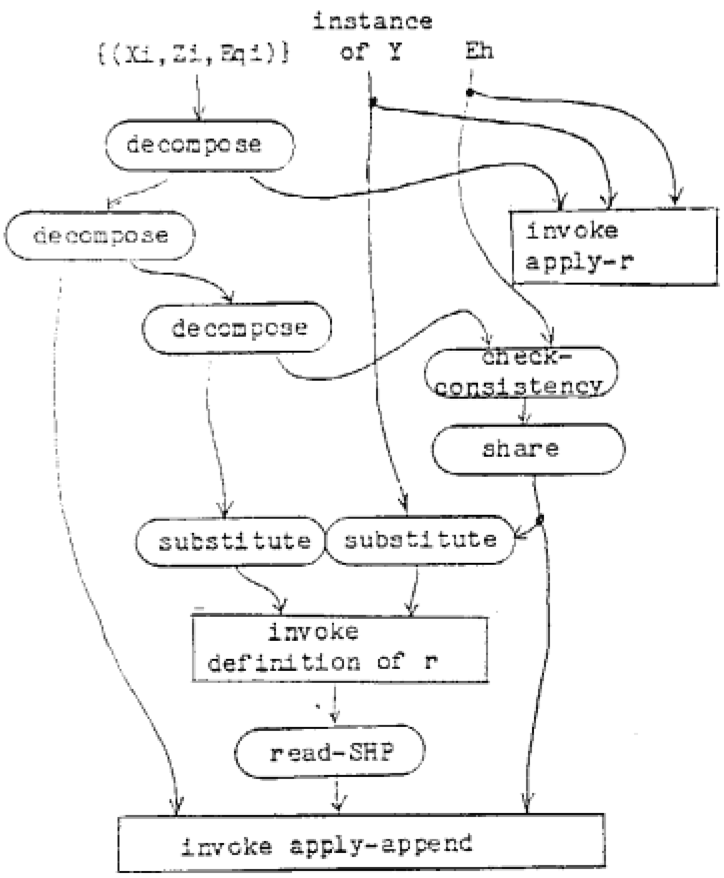
The body of apply-r further decomposes the solution into the instances of X and Z, and the environment Eq, which is used to check consistency with the environment of head unification Eh to produce a new environment Eq'. The bypassed instance of Y is replaced by this new environment (the result of check-consistency primitive).

In this procedure, when a goal literal has shared variables in its first and second arguments, the instances of X and Y, obtained by the substitute operators in Fig.13 (a), will share these variables. For example, if the given goal literal is p(Ws,Ws), the instances of X and Y are Xs and [a¦Xs], respectively, and share the variable Xs (where Ws = [a¦Xs]). If the execution of the first literal q(X,Z) succeeds and binds any term to the instance of X (the shared variable Xs), the same substitution for Xs must be applied to that of Y. The substitution information for Xs is given by Eq and will be used to produce the new environment Eq'. The shared variable Xs in the bypassed instance of Y is then replaced according to Eq' by the substitute operator in Fig.13 (b).

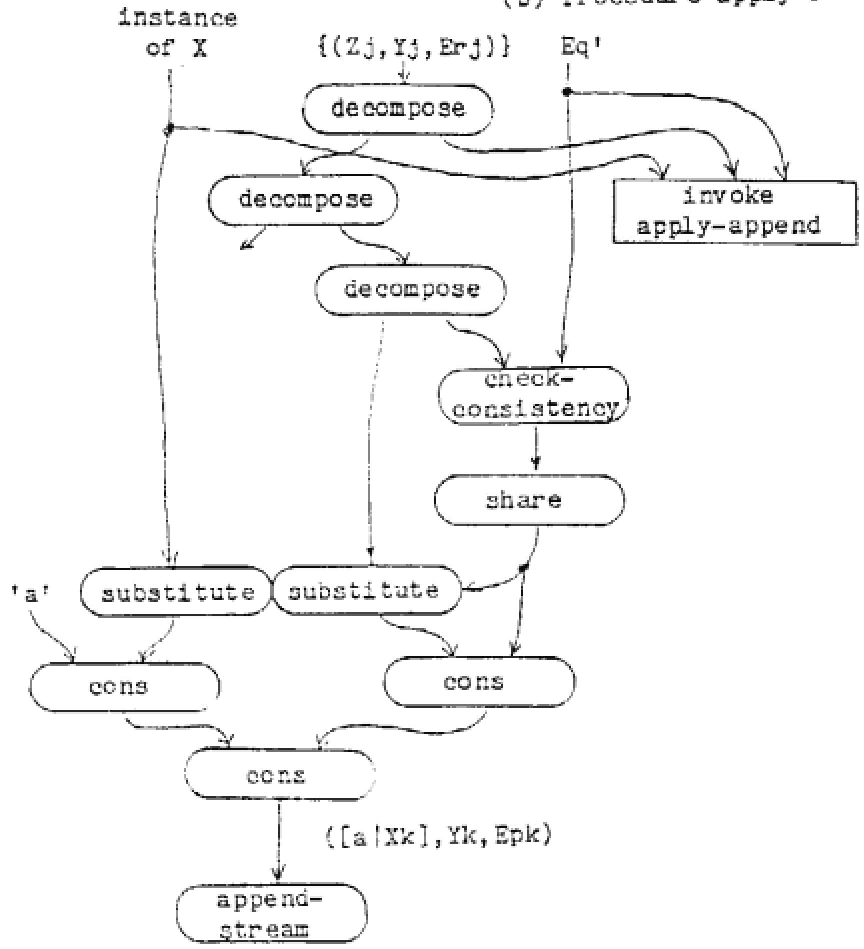
The consistent solution set, then, invokes the next body literal r(Z,Y), which again generates another stream {(Zj,Yj,Erj)}. The new stream is used as an argument of the next recursive procedure apply-append shown in Fig.13 (c). As the instance of X obtained from q(X,Z) is also transferred to apply-append([a¦X],Y,Ep) by bypassing r(Z,Y), the same consistency checking and substitution operations as in the case of the bypassed instance of Y must be executed.



(a) Head unification



(b) Procedure apply-r



(c) Procedure apply-append

Fig.13 The complete data flow graphs of clause p([a|X],Y) <- q(X,Z) & r(Z,Y).

(2) AND-parallel Execution

In the following example, AND-parallel execution is specified:

```
p([a|X],Y) <- q(X,Z) // r(Z,Y).
```

Here, both body literals q(X,Z) and r(Z,Y) are executed in parallel. Connection paths between instances of the variables are shown in Fig.14.

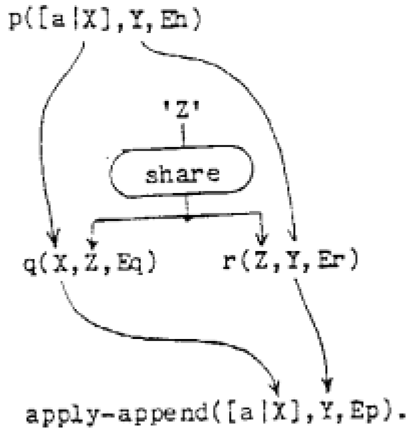


Fig.14 Connection paths of clause
p([a|X],Y,Eh) <- q(X,Z,Eq) // r(Z,Y,Er).

Since the variable Z is shared by two literals and is uninstantiated, it is changed to the shared variable Zs by the share operator before these literals are called. The two AND processes of the body obtain streams having as their i-th and j-th solutions (Xi,Zi,Eqi) and (Zj,Yj,Erj), respectively. This clause should obtain all combinations of these two solutions and check consistency between the two binding environments Eqi and Erj, the results of which are the final binding environments and are used for substitution of the instances of Xi and Yj. In this case, the procedure apply-append is defined as a duplicated recursive procedure, in order to divide two streams into their solutions.

6. Primitives for Concurrent Prolog

Processing of the clause head and the guard in Concurrent Prolog is almost the same as in Parallel Prolog, as described in Section 5. The major difference is that create-global-variable operators are used instead of share operators, and copy-global-variable operators are issued for every argument passed from the head to the guard before the guard consisting of multiple literals are called.

When the input operands of the create-global-variable operators are simple variables or structured data including simple variables, the operators allocate memory cells to all simple variables and change them into global variables. The copy-global-variable operators create a local copy for each global variable in their input operands. These outputs are passed to the clauses invoked by guard literals as their arguments.

Another difference from Parallel Prolog is the existence of a commit operator and read-only annotation.

6.1 Read-only Annotation

When a read-only tag is postfixed to a variable appearing as an argument in the goal literal, the set-read-only-tag operator is executed. This operator changes an instance of a variable to a read-only variable only when it is a global variable, as described in Section 4. When the input operand is other than a global variable, the operator outputs the input operand itself.

In the head unification of a clause, if the input argument from the goal literal is a read-only variable and if the corresponding head argument is a non-variable term, the unify operator tries to read the contents of the memory cell, which is pointed to by the read-only variable, before unification is performed. If the instance of the variable (contained in the memory cell) is not a non-variable term, the read request is suspended until the variable is bound to a non-variable term. The memory cell will be written by a guard operation mechanism, described below. In all other cases, the action of the unify operator is the same as in Parallel Prolog.

6.2 Guard operation mechanism

As described in Section 4, a commit operator has two functions: one for exclusive control and another for commitment of the binding environment for the global variables obtained by unification of the head and guard of the clause.

Exclusive control is nondeterministic, in that only one process which has executed the commit operator first can continue to a subsequent process. This nondeterminism is called 'don't-care nondeterminism'. To perform exclusive control, semaphore operators are provided. When a definition consisting of multiple guarded clauses is invoked, a create-guard 'operator is executed, as shown in Fig.15. This operator generates a semaphore flag to be shared between OR processes, initializes the flag to OFF, and sends its pointer to these OR processes.

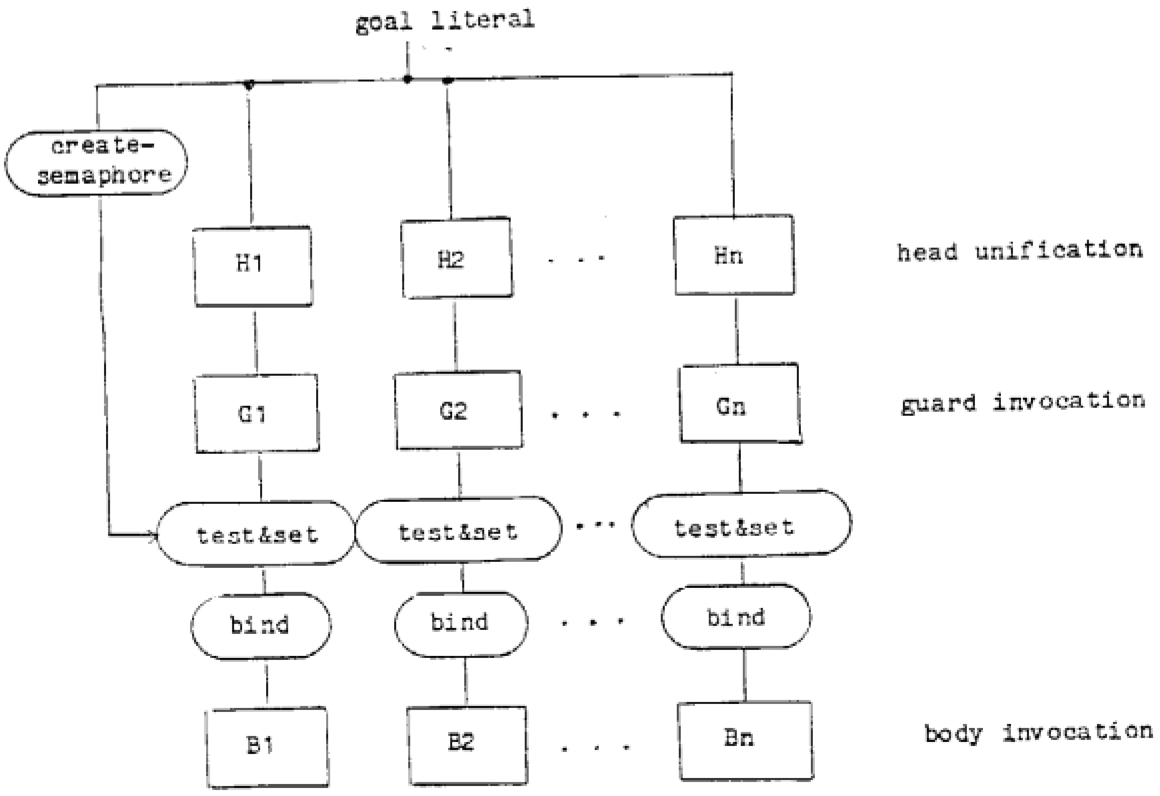


Fig.15 The guard control primitives

An OR process executing a guarded clause activates a test & set operator of the semaphore flag when unification of the clause head and the guard succeeds. The test & set operator reads and returns the contents of the semaphore flag, which shows whether or not it is the first OR process that passed the commit operator, and sets the semaphore flag to ON.

If the result of this test operation is OFF, the bind operator tests the binding environment for global variables; if it is not 'nil' or 'fail', the operator tries to unify an old instance previously written into the memory cell with a new instance of each global variable in the environment. That is, the bind operator obtains a pair consisting of a global variable and an instance from the binding environment, and executes the following sequence for all the pairs in the binding environment: it reads the contents of the memory cell pointed to by the global variable, attempts to unify the instance from the binding environment with the contents of the memory cell (i.e., the old instance for the global variable), then writes their common instance into the memory cell if the unification succeeds. While the memory cell is being written, it is locked to any other accesses. Finally, the written instances are made available to other processes and a trigger token is returned to the parent literal, that may initiate the next literal or the commit operator in the parent clause.

7. CONCLUSION

Execution mechanisms on a data flow machine for Parallel and Concurrent Prolog have been presented and primitive operators for supporting these two languages described. It has been shown that two types of logic programming languages with different aims can be supported on this machine.

There are two basic functions embedded in these languages: one is unification, and the other is nondeterminism. Several primitives for performing these functions are introduced and programs written by these languages are compiled into data flow graphs, which corresponds to the machine language. Thus, parallelism in the programs can be exploited naturally.

In order to exploit AND parallelism efficiently, unification primitives being executed in parallel generate bindings only for undefined shared variables. Check-consistency primitives of these bindings, therefore, are rather simplified and performance by exploiting this parallelism significantly increases.

Control of nondeterminism is related to OR parallelism: 'don't know nondeterminism' is necessary in Parallel Prolog, while 'don't care nondeterminism' is necessary in Concurrent Prolog. Stream merging primitives realize 'don't know nondeterminism', where OR processes executing independent clauses share a stream tail pointer and append new solutions to the tail of the stream, while stream consumer processes obtain the solutions by traversing the stream from its head pointer. Semaphore primitives and exporting mechanism of bindings for shared variables realize 'don't care nondeterminism', as in Dijkstra's guarded command. The guard, which succeeds first, makes its local bindings available to other processes sharing the variables.

Detailed designs for the machine are presently being developed; its simulation to Parallel Prolog programs indicates that performance can be significantly improved by exploiting parallelism [14]. Future efforts will involve the development of a Concurrent Prolog simulator and prototype hardware to serve as the basis for a highly-parallel inference machine.

Acknowledgement

The authors extend their thanks to Director Kazuhiro Fuchi in ICOT, who afforded them the opportunity to pursue this research. Also much appreciated are the valuable advice and comments of Dr. Kunio Murakami, Chief of the First Research Laboratory in ICOT, Mr. Rikio Onai, Senior Researcher, and other ICOT research members.

References

[1] Ackerman,W.B., " A Structured Processing Facility for Data Flow Computers", Proceeding of International Conference on Parallel Processing, 1978.

[2] Amamiya,M. and R.Hasegawa, "Data Flow Machine and Functional Language", AL81-84, PRL81-63, IECE of Japan, Dec., 1981 (in Japanese).

[3] Amamiya,M. R.Hasegawa, O.Nakamura, and H.Mikami, "A List-processing-oriented data flow architecture", Natinal Conputer Conference 1982, pp 143-151, June, 1982.

[4] Arvind, K.P.Gostelow, and W.E.Plouffe, "An Asynchronous Programming Language and Computing Machine", TR-114a, Dept. of ICS, University of California, Irvine, Dec., 1978.

[5] Arvind and R.A.Innucci, "A Critique of Multiprocessing von Neumann Style", Proceedings of 10th Internatinal Symposium on Computer Architecture, June, 1983.

[6] Clark,K. and S.Gregory, "A Relational Language for Parallel Programming", Research Report DOC 81/16, Imperial College of Science and Technology, July, 1981.

[7] Clark,K. and S.Gregory, "PARLOG: Parallel Programming in Prolog", Research Report DOC 84/4, Imperial College of Science and Technology, April, 1984.

[8] Cohen,J. "Garbage Collection of Linked Data Structures", Computing Surveys, Vol.13, No.3, Sept., 1981.

[9] Conery,J.S. and D.Kibler, "Parallel Interpretation of Logic Programming", Proceedings of Conference on Functional Programming and Computer Architecture, ACM, Oct. 1981.

[10] Dijkstra,E.M., "A Discipline of Programming", Prentece-Hall, 1976.

[11] Gurd,J.R. and I.Watson, "Data Driven System for High Speed Parallel Computing", Computer Design, July, 1980.

[12] Ito,N., K.Masuda, and H.Shimizu, " Parallel Prolog Machine Based on the Data Flow Model", TR-035, Institute for New Generation Computer Technology, Tokyo, Japan.

[13] Ito,N. and E.Kuno, " Simulation of a Parallel Prolog Machine", Proceedings of 28th National Conference of Information Processing Society of Japan, Tokyo, Japan, March, 1984 (in Japanese).

[14] Ito,N. and K.Masuda, "Parallel Inference Machine Based on the Data Flow Model", International Workshop on High Level Computer Architecture 84, Hyatt International Hotel, Los Angeles, California, May, 1984.

[15] Kowalski,R., "Predicate Logic as Programming Language", IFIP 74, North-Holland, 1974.

[16] Nakamura,O, R.Hasegawa, and M.Amamiya, "The Design and Evaluation of the Structure Memories for a List Processing Oriented Data Flow Machine", EC 81-32, IECE, Japan, 1981 (in Japanese).

[17] Onai,R. and M.Asou, "Control Mechanisms of the Guard and Read-Only Annotation in Parallel Inference Machine", Proceedings of 27th National Conference of Information Processing Society of Japan, Nagoya, Japan, Oct., 1983 (in Japanese).

[18] Shapiro,E.Y., "A Subset of Concurrent Prolog and its Interpreter", TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, Jan., 1983.

[19] Shapiro,E.Y., "System Programming in Concurrent Prolog", TR-034, Institute for New Generation Computer Technology, Tokyo, Japan, Nov., 1983.