TR-241

Parallel Programming with Layered Streams

by

A. Okumura and U. Matsumoto

March, 1987

PARALLEL PROGRAMMING WITH LAYERED STREAMS

Akira Okumura and Yuji Matsumoto

ICOT Research Center
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108 Japan

Abstract

We propose a parallel programming paradigm for solving search problems in committed-choice languages, and introduce a layered stream, which is a recursively defined data structure with a similar property to a stream. An element in a layered stream may include other layered stream(s). The main advantage of the data structure is that each layer is independently accessible from different processes. A stream is regarded as a set of data in the sense that a data structure that includes streams is interpreted as a set of data that can be obtained when all the elements in the streams are expanded. This representation enables different processes to access any part in the data structure, so the potential of parallelism becomes inherently high. Another advantage is that the process for each layer is programmed independently. If the problem has a recursive definition, as most of the search problems have, a clear and declarative way of writing parallel programs is possible.

1. Introduction

It is said that it is difficult to write search programs in committed-choice languages, which do not support the backtracking mechanism. To obtain all the solutions, users must write a program which gathers all the solutions as a set. This paper proposes a straightforward way of writing such a program. In this framework, we are not connected with how a solution is constructed but with what conditions each element in a solution must satisfy. This enables a declarative way of writing parallel programs.

Another way of solving search problems in parallel is to compile restricted Prolog programs into a committed-choice language. Approaches of this sort have already been introduced by some researchers.

Ueda's approach is the continuation-based method [1]. After the translation of a Prolog program into a committed-choice language, conjunctive goals are processed from left to right although alternatives for a goal in the original program are searched for in parallel. This is because a goal that follows another is passed as a continuation, and when the execution of the preceding goal is completed, the continuation invokes the execution of the next goal. In this approach, conjunctive goals are executed sequentially, making it difficult to extract enough parallelism.

Tamaki's approach is the stream-based method [2]. Conjunctive goals which share common variables are expressed as processes connected by a stream to pass solutions of one goal to another. Since all conjunctive goals are expanded, numerous stream interface processes such as composers, decomposers, and multipliers are generated. Processes corresponding to the original goals communicate through these interface processes. Since a goal sends its complete solutions to its output stream, the goal that receives the stream is suspended until its preceding goal produces at least one solution. These facts reduce the degree of parallelism.

The approach in this paper is similar to the stream-based method, but is more efficient. We use a special stream, called a layered stream, instead of a regular stream. Elements in a layered stream are not necessarily complete solutions of a goal, but can be partially determined solutions. They are constructed from their top elements, and can be referred to by other processes while their remaining parts are still under construction. Therefore, the parallelism as a whole increases.

This paper shows that most search problems are easily programmed in a uniform style that utilizes layered streams. This programming style is called layered stream programming. Several examples are given, and the performance of the programs compared with other styles of programs is illustrated. Although programs are written in GHC [3] throughout this paper, the mothod is applicable to any other committed-choice languages, such as PARLOG [4] and Concurrent Prolog [5].

2. Basic Concept of Layered Streams

A Prolog program often produces its solution as a list. Generally, the producer of a solution first determines the head of the list, then invokes another producer which generates the tail. This process runs recursively. A solution is formed from a head and a tail. However, since there can be more than one possible tail for one head, the Prolog program selects the tails one by one using a backtracking mechanism.

This paper proposes that such a list can be expressed by a pair consisting of a head and a set of all possible tails, such as

```
H*Ts
```

where H is the head, and Ts is a set of tails paired with H. The normal list expression is used to represent the set of tails. Such a list must be regarded as a set, and is treated as a stream in the parallel programs. The operator * represents the constructor to make up the solution of the problem. Lists may be used to express both. However, these two kinds of data structure were chosen to avoid confusion. Ts can be a null list (i.e. an empty set). When there is more than one possible head, the whole set of possible candidates of the solution is represented as the following list:

```
[H1*Ts1,H2*Ts2,...]
```

where H-i is the i-th possible head and Ts-i is a list of tails to be paired with H-i. This type of structure is called a layered stream. Ts-i also has the same structure. More generally, both sides of * may be layered streams. For example, the set of permutations of list [1,2,3] can be expressed as

```
[ 1*[2*[3*begin],3*[2*begin]],
     2*[1*[3*begin],3*[1*begin]],
     3*[1*[2*begin],2*[1*begin]] ]
```

where 'begin' marks the deepest end of each solution. As stated above, an element in a layered stream also includes other layered streams. This is the reason it is called layered. A layered stream represents a tree-like structure and 'begin', denoting the deepest layer, specifies a leaf of the tree.

There are several advantages of layered streams in parallel programs. A set of structures that have the same heads shares the head. This saves memory and reduces execution time. An element in a layered stream represents a set of data that share the same property (i.e., have the same head). This means that a set of data can be produced or tested in a single process. Most importantly, the tail of an element in a layered stream is again a layered stream. As will be seen in the sample programs shown in the next section, a process can generate and send an element through a layered stream while another process is still constructing the tail of the element. This is the key feature in increasing parallelism.

In the continuation-based method, invoking a consumer is signaled by the continuation when the producer completes a structure, giving a lower degree of parallelism. In the stream-based method, the producer and consumer are invoked simultaneously, but the stream transfers only the completed solution from the producer. The consumer is suspended until it receives at least one solution, causing a loss of parallelism.

3. Layered Stream Programming

This section shows how to obtain practical parallel programs using layered streams. First, a programming style based on this method is proposed, then examples are given. A brief discussion on solution forms follows.

3.1 Style of Layered stream Programming for Search Problems

Generally, a search problem is to build some structure of elements satisfying certain constraints. We pay attention to each element in the solution. Roughly speaking, a process is allocated to each position which might take part in a solution. The process generates the possible elements as the heads of lists, and also produces filtering processes which eliminate all the contradicting elements from the tail represented as a layered stream.

Programs are divided into three parts. The first part describes the configuration of the problem representing how the processes are connected. It designates the structure of solutions. The second part defines the processes used in the first part. A process generates all the possible elements for a certain position in a solution and makes each of them the head of the operator *. Their tails are created by other processes and are represented as layered streams. A filter is created for each layered stream and eliminates every element in it that cannot be the tail of the head element. Head elements can be passed to other processes even when their tails are not yet completed. The third part defines the filters. A filter describes the constraints that the head element of the operator * gives to the elements in its tail. Tails that are incompatible with the head are discarded from the layered stream.

3.2 N Queens Problem

The N queens problem is one of the most popular search problems. Normally it is solved by placing queens one by one so that the latest placed queen does not attack any queens that are already on the board. Sequentiality is unavoidable in this part when it is implemented by the continuation-based or stream-based method. In the layered stream method, any queen may be placed even if the preceding queens have not passed any information.

Figure 1 shows the GHC program for solving the 4 queens problem using layered streams.

```
fourQueens(Q4) :- true |
     q(begin,Q1),
     q(Q1,Q2),
     q(Q2,Q3),
     q(Q3,Q4).
q(In,Out) :- true |
     filter(In,1,1,Out1),
     filter(In,2,1,Out2),
     filter(In,3,1,Out3),
     filter(In,4,1,Out4),
     Out = [1*Out1,2*Out2,3*Out3,4*Out4].
filter(begin,_,_,Out) :- true |
     Out = begin.
filter([],_,_,Out) :- true | Out = [].
filter([I*_|Ins],I,D,Out) :- true |
     filter(Ins,I,D,Out).
filter([J*_|Ins],I,D,Out) :- D =:= I-J |
     filter(Ins,I,D,Out).
filter([J*_|Ins],I,D,Out) :- D =:= J-I |
     filter(Ins,I,D,Out).
filter([J*In1|Ins],I,D,Out) :-
          J \= I, D =\= I-J, D =\= J-I |
     D1 := D+1, filter(In1,I,D1,Out1),
     filter(Ins,I,D,Outs),
     Out = [J*Out1|Outs].
```

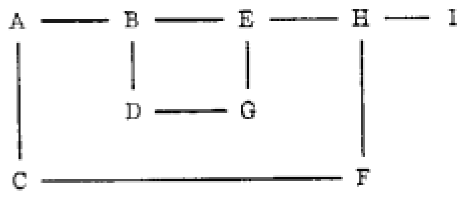
Figure 1 Four queens problem using layered streams

A queen is allocated to each column on the 4×4 chess board. Queens are defined as processes and are connected by streams in a row. This is the configuration of the problem, and corresponds to the first part of the program. This part is defined by fourQueens/1. fourQueens/1 produces four of q/2 processes, each of which corresponds to a queen. The second part of the program generates the possible positions of each queen and produces filters. q/2 corresponds to this part and generates the output bindings for four positions. (Note: this program is easily extended to N queens by redefining fourQueens/1 and q/2 appropriately.) The tail of the operator * produced by q/2 is the output of filter/4 and might not have been determined when the data was passed. filter/4 discards all the elements in the input stream that are incompatible with the head element, I. The definition of filter/4 is now easy to understand. The third clause of the definition discards the partial solution that puts a queen on the same row as the process that produced the filter. The fourth and fifth clauses discard the solutions that put queens in diagonal positions. The first clause is for the case where all the preceding queens are checked, that is, the filter comes at the deepest layer. The second clause is for the case where all the elements in the layered stream are checked. The last clause defines the successful case. In this case, the remaining elements in the layered stream must continue to be filtered, and another filter is produced to check the deeper layers.

The clause of q/2 and the last clause of filter/4 produce output bindings containing variables such as Out1, which is determined by other processes. Thus, the output is prepared before the internal stream is completed. Even when Out1 is still uninstantiated, the receiver of the stream becomes active and starts examining the layered stream.

3.3 Good Path Problem

The good path problem is to find acyclic paths between two specified nodes in a given graph. Consider the following graph. Arcs are assumed to be bidirectional.



The first part of the program generates a process for each node in the graph. Layered streams are set up along the arcs. Because arcs are bidirectional, streams are also provided in both directions. The second part produces the primary output bindings for each node. Since the only possible element for a node is its name, the second part only puts the node name in the output stream. The third part filters out anything from the input stream that includes the specified node name to eliminate loops. Figure 2 shows the program.

goodPath/3 forms the first part and defines the configuration of the graph. The second part defines node/6. The first and second arguments of node/6 specify the starting and goal nodes of the path. The third argument is the name of the node. The fourth argument shows the set of paths that come to the node. This set forms a layered stream. The fifth argument is eventually instantiated to the set of good paths terminating at the node. The last argument is for obtaining the solutions, and is instantiated by the goal node.

The first clause of node/6 is for the starting node. It specifies that there is no node preceding the starting node by putting the special symbol begin. The second and third clauses put the node name on top of the layered stream which is the output of the filter for that

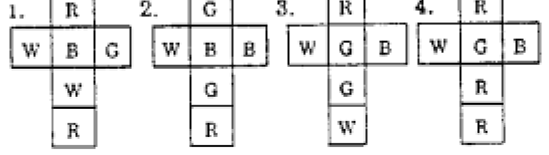
```
goodPath(Start,Goal,Path) :- true |
     node(Start,Goal,a,[B,C],A,Path),
     node(Start,Goal,b,[A,E,D],B,Path),
     node(Start,Goal,c,[A,F],C,Path),
     node(Start,Goal,d,[B,G],D,Path),
     node(Start,Goal,e,[B,G,H],E,Path),
     node(Start,Goal,f,[C,J],F,Path),
     node(Start,Goal,g,[D,E],G,Path),
     node(Start,Goal,h,[E,I,J],H,Path),
     node(Start,Goal,i,[H],I,Path),
     node(Start,Goal,j,[F,H],J,Path).

node(S,_,S,_,Out,_) :- true |
     Out = S*begin.
node(_,G,G,In,Out,Path) :- true |
     Out = nil, Path = G*In.
node(S,G,N,In,Out,_) :- S \= N, G \= N |
     Out = N*In1, filter(In,N,In1).

filter(begin,_,Out) :- true |
     Out = begin.
filter([],_,Out) :- true | Out = [].
filter([nil|In],Node,Out) :- true |
     filter(In,Node,Out).
filter([Node*_|In],Node,Out) :- true |
     filter(In,Node,Out).
filter([N*Ns|In],Node,Out) :- Node \= N |
     Out = [N*Ns1|Out1],
     filter(Ns,Node,Ns1),
     filter(In,Node,Out1).
```

Figure 2 Good path problem using layered streams

node name. The only difference between these clauses is that the second clause is for the goal node. It determines the value of the variable, Path, which is the answer of the problem. The definition of the filter is quite simple. It receives a layered stream and a node name, and filters out anything from the input layered stream that includes the node name.

3.4 Colored Cubes problem

Consider four cubes whose six sides are colored red, green, white or blue, as follows.



R = Red G = Green
W = White B = Blue

The problem is to find possible settings of the four cubes into a column so that each of the four sides of the column has all of the four colors. A cube can rotate in three dimensions, and has 24 possible placements. A program for this problem must generate every possible placement, and is rather long compared with the previous examples. Figure 3 shows the program of the layered stream method.

The program is again divided into three parts. The first part provides the sequence of four cubes, and invokes four processes of the second part. Those processes are connected by layered streams, and give a configuration that each solution of this problem should take. Each cube is expressed by a triple of pairs of opposite sides.

The second part generates every possible placement of a cube. The four sides of the column can be considered as two pair of opposite sides, such as left and right, and front and back. `set/3` generates three possibilities, in which two pairs of a cube serve as the sides of the column. `rotate1/5` decides which pair is on the left and the right, and which is at the front and the back. `rotate2/5` shows which side of the first pair is on the left and which is on the right, and `rotate3/6` shows which side of the second pair is at the front and which is at the back.

The third part is the filter, which is quite similar to the previous examples. This problem requires all four colors on each side of the column, so each color appears exactly once on a side. `filter/6` ensures that no cube shares the same color as other cubes.

3.5 Solution Forms

The execution result of the program of Figure 1 (4 queens) is as follows.

```
| ?- ghc fourQueens(Q).
52 msec.

Q = [     1*[3*[],4*[2*[]]],
          2*[4*[1*[3*begin]]],
          3*[1*[4*[2*begin]]],
          4*[1*[3*[]],2*[]]]

yes
```

Variable Q is instantiated to a layered stream, which contains a set of solutions of the 4 queens problem. Every sequence leading from the surface of Q to the symbol `begin` represent a solution. Some null lists (`[]`) are found in Q. They are generated when all the elements in a layered stream are filtered out, and do not contribute to a solution. Thus, Q is equivalent to `[[2,4,1,3],[3,1,4,2]]` in a regular stream.

```
% part 1 %

cubes(S4) :- true |
     cube(1,Q1), set(Q1,begin,S1),
     cube(2,Q2), set(Q2,S1,S2),
     cube(3,Q3), set(Q3,S2,S3),
     cube(4,Q4), set(Q4,S3,S4).

cube(1,Out) :- true |
     Out = q(p(w,g),p(r,w),p(b,r)).
cube(2,Out) :- true |
     Out = q(p(w,b),p(g,g),p(b,r)).
cube(3,Out) :- true |
     Out = q(p(w,b),p(r,g),p(g,w)).
cube(4,Out) :- true |
     Out = q(p(w,b),p(r,r),p(g,r)).

% part 2 %

set(q(P1,P2,P3),In,Out) :- true |
     rotate1(P1,P2,In,Out,Out1),
     rotate1(P1,P3,In,Out1,Out2),
     rotate1(P2,P3,In,Out2,[]).

rotate1(P1,P2,In,S,T) :- true |
     rotate2(P1,P2,In,S,SS),
     rotate2(P2,P1,In,SS,T).

rotate2(p(C1,C2),P2,In,S,T) :- true |
     rotate3(C1,C2,P2,In,S,SS),
     rotate3(C2,C1,P2,In,SS,T).

rotate3(C1,C2,p(C3,C4),In,Out,T) :-
          true |
     filter(C1,C2,C3,C4,In,Out1),
     filter(C1,C2,C4,C3,In,Out2),
     Out=[q(C1,C2,C3,C4)*Out1,
          q(C1,C2,C4,C3)*Out2|T].

% part 3 %

filter(_,_,_,_,[],O) :- true | O = [].
filter(_,_,_,_,begin,O) :- true |
     O = begin.
filter(C1,C2,C3,C4,[q(X,_,_,_)*_|I],O) :-
     C1 = X | filter(C1,C2,C3,C4,I,O).
filter(C1,C2,C3,C4,[q(_,X,_,_)*_|I],O) :-
     C2 = X | filter(C1,C2,C3,C4,I,O).
filter(C1,C2,C3,C4,[q(_,_,X,_)*_|I],O) :-
     C3 = X | filter(C1,C2,C3,C4,I,O).
filter(C1,C2,C3,C4,[q(_,_,_,X)*_|I],O) :-
     C4 = X | filter(C1,C2,C3,C4,I,O).
filter(C1,C2,C3,C4,[q(P,Q,R,S)*J|I],O) :-
          C1\=P, C2\=Q, C3\=R, C4\=S |
     filter(C1,C2,C3,C4,J,O1),
     filter(C1,C2,C3,C4,I,Os),
     O=[Q*O1|Os].
```

Figure 3 Colored cubes problem using layered streams

Of course, there can be a situation where solutions in the form of a regular stream are required. Although the transformation program from layered stream expression to normal stream expression is easy to construct, the program must retrieve solutions from a layered stream, and is a search program.

```
lastQ(In,Out) :- true |
     lastFilter([1],In,1,1,Out,Out1),
     lastFilter([2],In,2,1,Out1,Out2),
     lastFilter([3],In,3,1,Out2,Out3),
     lastFilter([4],In,4,1,Out3,[]).

lastFilter(Stack,begin,_,_,S,T) :- true |
     S = [Stack|T].
lastFilter(Stack,[],_,_,S,T) :- true |
     S = T.
lastFilter(Stack,[I*_|Ins],I,D,S,T) :-
     true |
     lastFilter(Stack,Ins,I,D,S,T).
lastFilter(Stack,[J*_|Ins],I,D,S,T) :-
     D =:= I-J |
     lastFilter(Stack,Ins,I,D,S,T).
lastFilter(Stack,[J*_|Ins],I,D,S,T) :-
     D =:= J-I |
     lastFilter(Stack,Ins,I,D,S,T).
lastFilter(Stack,[J*In|Ins],I,D,S,T) :-
     J \= I, D =\= I-J, D =\= J-I |
     D1 := D+1,
     lastFilter([J|Stack],In,I,D1,S,SS),
     lastFilter(Stack,Ins,I,D,SS,T).
```

Figure 4 Specialized filter

Such a search program is not required if a special definition of the filter is provided, the input of which is a layered stream and the output of which is a regular stream. Figure 4 shows the specialized filter and the definition of the fourth queen that utilizes it. If `lastQ/2` is used in place of the last `q/2` of `fourQueens/1` in Figure 1, a set of solutions in the form of a regular stream can be obtained.

The new predicate, `lastFilter/6`, has the last two arguments for output working as difference lists. It is the same as `filter/4` except that the first argument keeps the elements of a solution. Therefore, little extra computation is required.

4. Experiments

This section shows the statistical simulation results of N queens, good path, and colored cubes simulation. Table 1 shows the statistics of the execution of layered stream programs and those of the continuation-based and stream-based programs. A breadth first execution model of programs, where all reducible goals are reduced simultaneously, is assumed. Simultaneous reduction of goals is counted as one cycle. In Table 1, reductions and suspensions stand for the total number of goals reduced and suspended during the computation. A suspension of a goal is counted as one even if it continues to be suspended during two or more cycles. The number of cycles more or less indicates the time complexity in the ideal parallel environment where reducible goals are reduced within a certain amount of time.

As shown in the table, continuation-based programs make no process suspension, because they are executed completely deterministically. Since conjunctive goals are executed sequentially, and all processes are invoked with the necessary data, no process need wait for binding by other processes. However, this method does not extract AND parallelism.

Stream-based programs are, in most cases, less efficient than continuation-based programs. The total number of suspensions is the largest. Conjunctive goals are expanded to parallel processes. Streams are set up according to the input-output causal relation. However, the producer of a stream never makes output binding until the first solution is found, and the consumer of the stream suspends during that period of time.

Layered stream programs are the most efficient. The total number of reductions for N queens is less than half of that of continuation-based programs and one third of that of stream-based programs. Although the good path does not show apparent differences in the

Table 1 Exeecution analysis of each method

	Reductions	Suspensions	Cycles
= <6 queens> =			
Continuation	2932	0	80
Stream	3161	1566	69
Layered Stream	1340	124	25
= <8 queens> =			
Continuation	48543	0	133
Stream	53824	25033	112
Layered Stream	19418	2470	38
= <Good path> =			
Continuation	205	0	48
Stream	312	92	62
Layered Stream	181	26	13
= <Colored cubes> =			
Continuation	47383	0	33
Stream	62601	15810	121
Layered Stream	8085	2033	44

number of reductions, the colored cubes gives differences of six to eight times. This is because layered streams reflect the form of solutions and no processes are required to construct solutions.

Layered stream programs also have the least number of cycles except that of colored cubes. A stream transfers the output binding even when some of the stream elements are under construction. In the second part of layered stream programs, processes generate the output binding at the same time that they create the filters. Therefore, most of the filters are active, since output bindings and filters are created simultaneously.

As to colored cubes, the layered stream method gives a larger number of cycles than the continuation-based. This is because each cube in this problem has a large number of possible placements. The possibility is reflected by the size of each layer of the layered streams. Each layer is currently implemented by a list structure and its elements are handled in sequence. This makes the number of cycles higher. However, in substance, the contents of a layer can be examined simultaneously. Therefore, more efficient implementation will be possible.

5. Detailed Comparison with Other Approaches

The preceding section compared our method with the continuation-based method and the stream-based method by simulating examples. This section compares these methods in more detail.

5.1 Comparison with Continuation-based Method

The continuation-based method is intended to implement a deterministic search. A Prolog program transformed according to this method is almost as efficient as the original program, since the original Prolog program's backtracking search is implemented as a continuation and the amount of computation is almost the same. A transformed program runs in Prolog almost as fast as its source program which makes a backtracking search without collecting solutions.

It has some disadvantages in parallel execution because of its inherent sequentiality. Since conjunctive relations are executed sequentially, the total number of cycles is relatively large. Consider the following Prolog clause.

```
p(In,[X|Y]) :- q(In,In1,X),p(In1,Y).   (1)
```

Suppose that the first arguments of both of p/2 and q/3 are input, and the other arguments are output. Clauses of this type often appear in practical programs. To translate this clause into a continuation-based program, the output binding of the second argument of the head goal must be done after the termination of the body. Therefore, the clause must be understood as:

```
p(In,Out) :- q(In,In1,X),p(In1,Y),
             Out = [X|Y].           (2)
```

Such output bindings are stacked as an argument of continuation in the transformed program, and are not executed until the execution of the original body goals finishes. This increases the sequentiality and the total number of cycles.

In the layered stream method, the processes corresponding to the body goals of a clause are invoked simultaneously. The stream that connects these processes transfers partial solutions immediately after the processes are created, increasing the potential of parallel execution. The difference of the two methods appears as the difference of the numbers of cycles in Table 1.

5.2 Comparison with Stream-based Method

The layered stream method is basically a stream-based method. However, the stream contents are different. While a regular stream transfers complete solutions of a goal in a conjunctive relation, a layered stream transfers structures which are not necessarily completed at the time of sending. This difference is reflected in the difference of efficiency.

The stream-based method requires extra processes that have no equivalents in the original Prolog program. Because processes in a conjunctive relation communicate through streams, the two kinds of extra computation, decomposition and composition of streams, are necessary. Consider clause (1) in Section 5.1. In the program translated by this method, the solutions of the first goal in the body of the clause are transferred to the second goal through a stream. In the second goal, the stream is decomposed and its solutions are computed for each element. Then a new stream is composed for transferring the total solutions. These decomposition and composition processes are required in every step in the recursion.

There is another disadvantage in the stream-based method. Although the processes corresponding to conjunctive goals are invoked simultaneously, the streams among them transfer only completed solutions of the goals. Goals are suspended until the first solution of their input stream is received. For this reason, not only the number of cycles but also the total number of suspensions is large.

In the layered stream method, there is no overhead of decomposition and composition of streams. To implement the output binding corresponding to (2), a single unification is enough:

```
Out = X*Ys
```

Ys is a layered stream including the solutions of p/2 in the body. Since this unification can be executed in parallel with other goals, the processes which require output binding can access this structure eagerly. Thus, the degree of parallelism is higher.

6. Evaluation of Layered Stream Method

This section evaluates the layered stream method in four aspects: overhead, parallelism, space efficiency, and ease of programming.

6.1 Overhead

In the second part of the layered stream programs, a process generates the head of a layered stream element regardless of the value of the tail. In some cases, the tail might be an empty stream. This means that there is no tail that can be paired with the head, and having sent the head is useless. This is where the layered stream method may have its disadvantage compared with other methods. The overhead caused by useless generation of the partial data is probably not serious. The program for the N queens problem is quite efficient even when it is executed sequentially.

A simple, but effective countermeasure is what we call a *non-nil check*. In the second part of the program, the process is redefined so as to output an element into a stream after confirming its tail is not empty. If it is empty, no output is produced. Thus, no redundant computation is made, although sequentiality increases because the non-nil check delays the output until that confirmation. We consider that this modification is useful when the search space is extremely large.

6.2 Parallelism

The layered stream method provides a large degree of parallelism. All the processes corresponding to possible elements in the problem domain become active almost simultaneously. This is the great advantage of the layered stream method since the other two methods cannot extract such parallelism. This style of programming is enabled by the layered data structure. This programming style is quite suitable for committed-choice parallel programming languages since the only control mechanism is the suspension.

6.3 Space Efficiency

The layered stream method also provides good space efficiency. The contents of a layered stream consist of shared structures. For example, the set of all sequences of length two consisting of 0 and 1 is expressed by a regular stream:

```
[[0,0],[0,1],[1,0],[1,1]]
```

Twelve cells are used in this structure. The same set is expressed by a layered stream:

```
[0*[0*begin,1*begin],1*[0*begin,1*begin]]
```

The internal stream can be shared, so it requires only eight cells. In general, a set of sequences of length L consisting of N distinct elements is expressed by a regular stream with $(L+1)\times N^L$ cells. The same set is expressed by a layered stream with only $2\times N\times L$ cells. Although the shared structures are copied according to the computation in a layered stream program, it never requires more memory space than stream-based programs.

6.4 Declarative Description

As mentioned in the section on programming style, a layered stream program reflects the nature of the source problem. The first part of the program represents the structure of the solutions, the second part describes the possible elements that construct solutions, and the third part defines the constraints that an element must satisfy with other elements in the same solution. The concept of layered stream programming offers a fairly declarative way of writing parallel programs for search problems.

Once the programmer notices what the primary elements of solutions are, it is straightforward to write the layered stream program.

7. Future research

This paper introduced a parallel programming paradigm using layered streams. A layered stream provides a high degree of parallelism and a declarative way of writing parallel programs. Most search problems can be easily and directly programmed in this paradigm. Current research aims to clarify the class of problems for which the layered stream method is effective.

Another issue is to define a description language for the layered stream programming. Prolog is probably not the answer.

The examples shown in this paper represent their solutions in a linear structure. The form of the solution is essentially either a sequence of elements or a set of elements. It is not yet understood what consideration is necessary when the form of the solution is more complex. Taking these problems into consideration will help to design a good description language.

Acknowledgment

We would like to thank Koichi Furukawa, the deputy director of ICOT Research Center for his comments and discussion. Thanks are also due to our colleagues in ICOT, especially Kazunori Ueda and Toshihiko Miyazaki, for their comments and suggestions.

References

[1] Ueda, K.: "Making Exhaustive Search Programs Deterministic", Proc. Third International Conference on Logic Programming, (Lecture Notes in Computer Science, Vol.225), Springer-Verlag, pp.270-282, 1986.

[2] Tamaki, H.: "Stream-based Compilation of Ground I/O Prolog into Committed-choice Languages", Proc. the Fourth International Conference on Logic Programming, J-L. Lassez (ed.), The MIT Press, pp.376-393, 1987.

[3] Ueda, K.: "Guarded Horn Clauses", in Proc. Logic Programming '85, E. Wada (ed.), Lecture Notes in Computer Science, 221, Springer-Verlag, pp.168-179, 1986.

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

[5] Shapiro, E., "A Subset of Concurrent Prolog and its Interpreter," ICOT Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983.

Appendix 1 Four queens program by continuation-based compilation

```
q(B) :- true |
    'sweeper$q1'([1,2,3,4],[],'L1',B,[]).

'sweeper$q1'([H|T],R,Cont,Rs0,Rs1) :-
        true |
    'sweeper$sel'([H|T],'L2'(Cont,R),
        'L2',Rs0,Rs1).
'sweeper$q1'([],R,Cont,Rs0,Rs1) :- true |
    Rs0 = [R|Rs1].

'sweeper$sel'(HT,Cont,Conts,Rs0,Rs2) :-
        true |
    'sel/3#1'(HT,Cont,Conts,Rs0,Rs1),
    'sel/3#2'(HT,Cont,Conts,Rs1,Rs2).

'sel/3#1'([A|L],'L2'(Cont,R),Conts,
        Rs0,Rs1) :- true |
    'sweeper$check1'(R,A,1,
        'L2b'(Cont,R,A,L,Conts),Rs0,Rs1).
'sel/3#1'([],Cont,Conts,Rs0,Rs1) :- true |
    Rs0=Rs1.

'sel/3#2'([H|T],Cont,Conts,Rs0,Rs1) :- '
        true |
    'sweeper$sel'(T,Cont,'L5'(Conts,H),
        Rs0,Rs1).
'sel/3#2'([],Cont,Conts,Rs0,Rs1) :- true |
    Rs0=Rs1.

'sweeper$check1'([H|T],U,N,Cont,Rs0,
        Rs1) :- H=\=U+N, H=\=U-N, N1:=N+1 |
    'sweeper$check1'(T,U,N1,Cont,Rs0,Rs1).
'sweeper$check1'([H|T],U,N,Cont,Rs0,
        Rs1) :- H =:= U+N |
    Rs0=Rs1.
'sweeper$check1'([H|T],U,N,Cont,Rs0,
        Rs1) :- H =:= U-N |
    Rs0=Rs1.
'sweeper$check1'([],U,N,
        'L2b'(Cont,R,A,L,Conts),Rs0,Rs1) :-
         true |
    b(Conts,'L3'(Cont,R,A),L,Rs0,Rs1).

b('L5'(Conts,A),Cont,T,Rs0,Rs1) :- true |
    b(Conts,Cont,[A|T],Rs0,Rs1).
b('L2','L3'(Cont,R,A),L,Rs0,Rs1) :- true |
    'sweeper$q1'(L,[A|R],Cont,Rs0,Rs1).
```

Appendix 2 Four queens program by stream-based compilation

```
queens(Q) :- true |
    'Qq'([1,2,3,4],[],Q,[]).

'Qq'([],Y,Z0,Z1) :- true | Z0 = [Y|Z1].
'Qq'(X,Y,Z0,Z1) :- X \= [] |
    'Qsel'(X,UVs,[]), 'Iq21'(Y,UVs,Z0,Z1).

'Qsel'([],Z0,Z1) :- true | Z0 = Z1.
'Qsel'([X|Y],Z0,Z2) :- true |
    Z0 = [(X,Y)|Z1],
    'Qsel'(Y,UVs,[]),
    'Isel21'(X,UVs,Z1,Z2).

'Iq21'(Y,[(U,V)|UVs],Z0,Z2) :- true |
    'Qcheck'(Y,U,1,YY),
    'Iq22'(V,[U|Y],YY,Z0,Z1),
    'Iq21'(Y,UVs,Z1,Z2).
'Iq21'(_,[],Z0,Z1) :- true | Z0 = Z1.

'Iq22'(V,List,ok,Z0,Z1) :- true |
    'Qq'(V,List,Z0,Z1).
'Iq22'(_,_,ng,Z0,Z1) :- true | Z0 = Z1.

'Isel21'(X,[(U,V)|UVs],Z0,Z2) :- true |
    Z0 = [(U,[X|V])|Z1],
    'Isel21'(X,UVs,Z1,Z2).
'Isel21'(_,[],Z0,Z1) :- true | Z0 = Z1.

'Qcheck'([Q|R],P,N,Res) :-
        Q =\= P+N, Q =\= P-N |
    M := N+1,
    'Qcheck'(R,P,M,Res).
'Qcheck'([Q|R],P,N,Res) :- Q := P+N |
    Res = ng.
'Qcheck'([Q|R],P,N,Res) :- Q := P-N |
    Res = ng.
'Qcheck'([],_,_,Res) :- true | Res = ok.
```