An Exhaustive-Search Method using Layered Streams Obtained through a Meta-Interpreter for Chain Programs

David A. Rosenblueth

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Universidad Nacional Autónoma de México
Apdo. 20-726, 01000 Mexico D.F.
drosenbl@servidor.unam.mx

1 Introduction

Okumura and Matsumoto have published [2] examples of logic programs using a data structure they call "layered stream," especially suited for completely traversing a search space in a deterministic manner [3, 5]. A layered stream is a representation of a list of lists, in which common heads of adjacent sublists have been factored. This factorization allows the pruning of several branches of the search space in constant time and is also a source of parallelism [4, p. 147]. The published examples include the N-queen problem and "instant insanity" (a precursor of Rubik's cube). Each such deterministic, layered-stream program supposedly performs exhaustive search over the space generated by some nondeterministic, naive program. However, a method converting a nondeterministic program into a corresponding exhaustive-search, layered-stream version has not yet been proposed, as far as we know [1]. Layered streams have thus remained difficult to use [4, p. 408], since the programmer must be concerned both about factorization of heads and exhaustive search. We build upon the work of Okumura and Matsumoto by showing how to translate nondeterministic programs into exhaustive-search, layered-stream form. Our method is restricted to generate-and-test programs fitting a certain schema.

We start by considering syntactically restricted programs that we call "chain" programs, based on clauses of the form:

$$p_0(X_0, X_n) \leftarrow p_1(X_0, X_1),\ p_2(X_1, X_2),\ \ldots,\ p_n(X_{n-1}, X_n)$$

Reasoning about exhaustive search for such programs can be conveniently done in terms of union and composition of binary relations. We then exploit this property and write an exhaustive-search meta-interpreter for chain programs. Next, we incorporate layered streams and observe that the interpretation overhead can be eliminated with partial deduction.

2 Layered streams

We devote this section to some basic definitions of layered streams and give an example using this data structure.

Syntax. A layered stream is recursively defined as follows.

- *begin* and $[\,]$ are *layered streams.*
- $[X * Xs|\,Ys]$ is a *layered stream* if $X * Xs$ is a $*$-pair and Ys is a layered stream.

Now we define $*$-pairs in terms of layered streams.

- $X * Xs$ is a $*$-*pair* if Xs is a layered stream.

Semantics. Let $\overline{Xs}$ denote the list of lists represented by the layered stream or $*$-pair Xs. Then

$$\begin{aligned} \overline{begin} &= [[\,]] \\ \overline{[\,]} &= [\,] \\ \overline{[X * Xs|\,Ys]} &= \overline{X * Xs} \bullet \overline{Ys} \end{aligned}$$

where $\bullet$ denotes list concatenation. A $*$-pair $X * Xs$ represents the list obtained by inserting X in front of every list in $\overline{Xs}$.

$$\overline{X * Xs} = X \ \diamond \ \overline{Xs}$$

where

$$\begin{aligned} X \ \diamond \ [\,] &= [\,] \\ X \ \diamond \ [Y|\,Ys] &= [[X|Y]|X \ \diamond \ Ys] \end{aligned}$$

Note that a layered stream having only $*$-pairs which represent the empty list will also represent the empty list. This allows us to include "garbage" at any level. For instance the layered stream:

$$[\ 1*[3*[\,], 4*[2*[\,]]],\ 2*[4*[1*[3*begin]]],\ 3*[1*[4*[2*begin]]],\ 4*[1*[3*[\,]], 2*[\,]]\ ] \qquad (1)$$

represents $[[2,4,1,3],\ [3,1,4,2]]$.

As an example of an exhaustive-search program using layered streams, we reproduce in Figure 1 a program from [2] (with slight changes) for obtaining all answers for the four-queen problem. The answer to the goal $\leftarrow fourQueen(begin, Z)$ is the layered stream given above (1).

Perhaps *begin* was originally so called because some layered-stream programs built the intended lists in reverse. Garbage can be eliminated to a large extent by using variants of the predicate with symbol *filter* [2, 4]. For simplicity, we will not be concerned here about such elimination. We also omit mentioning the merits of layered streams, for which we refer to [2, 4].

3 An exhaustive-search meta-interpreter for chain programs

3.1 Chain programs

We define a *chain program* as a definite program such that all its binary predicates are defined only by clauses of the form:

$$\begin{array}{ll} & p_0(X_0, X_n) \leftarrow p_1(X_0, X_1),\ p_2(X_1, X_2),\ \ldots,\ p_n(X_{n-1}, X_n) \qquad n \geq 0 \\ \text{or} & p(\langle t, Xs\rangle, [t'|Xs]) \leftarrow test(\langle u, Xs\rangle) \end{array}$$

$$
\begin{aligned}
&\mathit{fourQueen}(Xs_0, Xs_4) \leftarrow qs(Xs_0, Xs_1),\ qs(Xs_1, Xs_2),\ qs(Xs_2, Xs_3),\ qs(Xs_3, Xs_4)\\
&qs(In, [1{*}Out_1, 2{*}Out_2, 3{*}Out_3, 4{*}Out_4]) \leftarrow \mathit{filter}(In, 1, 1, Out_1),\ \mathit{filter}(In, 2, 1, Out_2),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{filter}(In, 3, 1, Out_3),\ \mathit{filter}(In, 4, 1, Out_4)\\
&\mathit{filter}(begin, \_, \_, begin) \leftarrow\\
&\mathit{filter}([\,], \_, \_, [\,]) \leftarrow\\
&\mathit{filter}([I{*}\_|Ins], I, D, Out) \leftarrow \mathit{filter}(Ins, I, D, Out)\\
&\mathit{filter}([J{*}\_|Ins], I, D, Out) \leftarrow D =:= I - J,\ \mathit{filter}(Ins, I, D, Out)\\
&\mathit{filter}([J{*}\_|Ins], I, D, Out) \leftarrow D =:= J - I,\ \mathit{filter}(Ins, I, D, Out)\\
&\mathit{filter}([J{*}In'|Ins], I, D, [J{*}Out'|Outs]) \leftarrow J =\backslash= I,\ D =\backslash= I - J,\ D =\backslash= J - I,\ D'\ \mathit{is}\ D + 1,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{filter}(In', I, D', Out'),\ \mathit{filter}(Ins, I, D, Outs)
\end{aligned}
$$

Figure 1: *A layered-stream program for the four-queen problem.*

where the X_i's are distinct variables, $\text{var}(u) \subseteq \text{var}(t)$, and $\text{var}(t') \subseteq \text{var}(t)$. Here, and throughout the sequel, we use $\text{var}(t)$ to denote the set of variables in term t. We use angled brackets $\langle\ \rangle$ as a function symbol of a term grouping inputs and outputs.

A chain program for the four-queen problem, computing one solution at a time, would be:

$$
\begin{aligned}
&f(X_0, X_4) \leftarrow q(X_0, X_1),\ q(X_1, X_2),\ q(X_2, X_3),\ q(X_3, X_4)\\
&q(Is, [1|Is]) \leftarrow \mathit{safe}(\langle 1, 1, Is\rangle)\\
&q(Is, [2|Is]) \leftarrow \mathit{safe}(\langle 2, 1, Is\rangle)\\
&q(Is, [3|Is]) \leftarrow \mathit{safe}(\langle 3, 1, Is\rangle)\\
&q(Is, [4|Is]) \leftarrow \mathit{safe}(\langle 4, 1, Is\rangle)\\
&\mathit{safe}(\langle \_, \_, [\,]\rangle) \leftarrow\\
&\mathit{safe}(\langle I, D, [J|Is]\rangle) \leftarrow J =\backslash= I,\ D =\backslash= I - J,\ D =\backslash= J - I,\ D'\ \mathit{is}\ D + 1,\\
&\qquad\qquad\qquad\qquad \mathit{safe}(\langle I, D', Is\rangle)
\end{aligned} \tag{2}
$$

This program can be viewed as denoting the system of relational expressions:

$$
\begin{aligned}
F &\supseteq Q\ ;\ Q\ ;\ Q\ ;\ Q\\
Q &\supseteq Q_1 \cup Q_2 \cup Q_3 \cup Q_4
\end{aligned}
$$

where ";" represents relational composition, F and Q are the names of the relations denoted by f and q, respectively, and each Q_i $(i = 1, \ldots, 4)$ is a "primitive" relation denoted by the clause $q(Is, [i|Is]) \leftarrow \mathit{safe}(\langle Is, i, 1\rangle)$. In addition, sets are represented by lists. We are interested in the least solution of this system.

3.2 An exhaustive-search meta-interpreter

We illustrate our object-program representation with the p program:

$$
\mathit{cls}(f, [f_1]) \leftarrow \qquad\qquad \text{\% only one clause defines } f
$$

$$
\begin{array}{llll}
\mathit{cls}(q, [q_1, q_2, q_3, q_4]) \leftarrow & \text{\% four clauses define } q & & \\
\mathit{cl\_comp}(f_1, [q, q, q, q]) \leftarrow & \text{\% clause (2)} & & \\
\multicolumn{4}{l}{\mathit{cl\_test}(q_1, \mathit{Is}, [1|\mathit{Is}]) \leftarrow \mathit{safe}(\langle 1, 1, \mathit{Is}\rangle)} \\
\multicolumn{4}{l}{\mathit{cl\_test}(q_2, \mathit{Is}, [2|\mathit{Is}]) \leftarrow \mathit{safe}(\langle 2, 1, \mathit{Is}\rangle)} \\
\multicolumn{4}{l}{\mathit{cl\_test}(q_3, \mathit{Is}, [3|\mathit{Is}]) \leftarrow \mathit{safe}(\langle 3, 1, \mathit{Is}\rangle)} \\
\multicolumn{4}{l}{\mathit{cl\_test}(q_4, \mathit{Is}, [4|\mathit{Is}]) \leftarrow \mathit{safe}(\langle 4, 1, \mathit{Is}\rangle)} \\
\mathit{is\_test}(q_1) \leftarrow & \mathit{is\_test}(q_2) \leftarrow & \mathit{is\_test}(q_3) \leftarrow & \mathit{is\_test}(q_4) \leftarrow
\end{array}
$$

Obtaining all answers for a chain program and a goal $\leftarrow p(\mathbf{x}_0, Z)$ translates to evaluating the expression $\{(\mathbf{x}_0, \mathbf{x}_0)\} ; P$. During this evaluation, we will have to evaluate relational expressions of the form:

$$\{(\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_d, \mathbf{x}_d)\} ; P^1 ; P^2 ; \ldots ; P^m \tag{3}$$

$$\{(\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_k, \mathbf{x}_k)\} ; (P_1 \cup P_2 \cup \ldots \cup P_n) \tag{4}$$

We wish to achieve as much factorization as possible of partial solutions. This suggests using a breadth-first evaluation of both compositions and unions of relations. Hence, we will use a left-to-right evaluation of (3) and (4). (Using other strategies leads to meta-interpreters having essentially the same behavior as either the continuation-based [5] or the stream-based [3] exhaustive-search methods.)

The top-level predicate $\mathit{list\_solve\_subgoal}(P, [\mathbf{x}_1, \ldots, \mathbf{x}_d], \mathit{Zs})$ is intended to hold iff

$$\mathit{Zs} = \pi_2(\{(\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_d, \mathbf{x}_d)\} ; P).$$

Here, and thoughout the sequel π_2 denotes projection on the second coordinate. A definition of this predicate would be:

$$
\begin{array}{l}
\mathit{list\_solve\_subgoal}(P, [\,], [\,]) \leftarrow \\
\mathit{list\_solve\_subgoal}(P, [X|\mathit{Xs}], \mathit{Zs}) \leftarrow \mathit{cls}(P, \mathit{Pjs}),\ \mathit{list\_solve\_clauses}(\mathit{Pjs}, [X|\mathit{Xs}], \mathit{Zs})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{list\_solve\_clauses}([\,], \mathit{Xs}, [\,]) \leftarrow & \text{\% union} \\
\mathit{list\_solve\_clauses}([\mathit{Pj}|\mathit{Pjs}], \mathit{Xs}, \mathit{YsZs}) \leftarrow & \mathit{list\_solve\_clause}(\mathit{Pj}, \mathit{Xs}, \mathit{Ys}), \\
& \mathit{list\_solve\_clauses}(\mathit{Pjs}, \mathit{Xs}, \mathit{Zs}), \\
& \mathit{append}(\mathit{Ys}, \mathit{Zs}, \mathit{YsZs})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{list\_solve\_clause}(\mathit{Pj}, \mathit{Xs}, \mathit{Ys}) \leftarrow & \mathit{is\_test}(\mathit{Pj}),\ \mathit{list\_cl\_test}(\mathit{Pj}, \mathit{Xs}, \mathit{Ys}) \\
\mathit{list\_solve\_clause}(\mathit{Pj}, \mathit{Xs}, \mathit{Zs}) \leftarrow & \mathit{cl\_comp}(\mathit{Pj}, \mathit{Qs}), \\
& \mathit{list\_solve\_subgoals}(\mathit{Qs}, \mathit{Xs}, \mathit{Zs})
\end{array} \tag{5}
$$

$$
\begin{array}{ll}
\mathit{list\_solve\_subgoals}([\,], \mathit{Xs}, \mathit{Xs}) \leftarrow & \text{\% composition} \\
\mathit{list\_solve\_subgoals}([P|\mathit{Ps}], \mathit{Xs}, \mathit{Zs}) \leftarrow & \mathit{list\_solve\_subgoal}(P, \mathit{Xs}, \mathit{Ys}), \\
& \mathit{list\_solve\_subgoals}(\mathit{Ps}, \mathit{Ys}, \mathit{Zs})
\end{array}
$$

The predicate $list\_cl\_test(Pj, Xs, Zs)$ (clause (5)) could be defined in terms of $cl\_test(Pj, X, Y)$:

$$list\_cl\_test(Pj, [\,], [\,]) \leftarrow$$
$$list\_cl\_test(Pj, [X\,|Xs], [Y\,|\,Ys]) \leftarrow cl\_test(Pj, X, Y),\ list\_cl\_test(Pj, Xs, Ys)$$
$$list\_cl\_test(Pj, [X\,|Xs], Ys) \leftarrow not(cl\_test(Pj, X, \_)),\ list\_cl\_test(Pj, Xs, Ys)$$

However, observe that we are now in a position to create a layered stream representing all answers to $\leftarrow Pj(Xs, Ys)$, by giving an alternative definition.

3.3 Layered streams for example program

In our example, the object predicate symbol for *list_cl_test* is q_i $(i = 1, \ldots, 4)$. Then instead of representing each partial answer explicitly, we will now get as input a layered stream *Ins*, and will produce as output the layered stream $[i{*}Outs]$, where $\overline{Outs}$ is $\overline{Ins}$ minus the lists that fail the predicate with symbol *safe*. The new definition of the predicate with symbol *list_cl_test* when the first argument is q_1 is:

$$list\_cl\_test(q_1, Ins, [1{*}Outs]) \leftarrow filter(Ins, 1, 1, Outs)$$

and similarly for q_2, q_3, and q_4. The definition of *filter* appears in Figure 1, and can be proved correct from the definition of *safe* and the meaning of a layered stream.

The intended meaning of $list\_cl\_test(Pj, Xs, Ys)$ has now changed as follows: $list\_cl\_test(Pj, Xs, Ys)$ holds iff Xs and Ys are layered streams such that $\overline{Ys} = \pi_2(\{(\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_d, \mathbf{x}_d)\}\ ;\ Pj)$, where $\overline{Xs} = [\mathbf{x}_1, \ldots, \mathbf{x}_d]$. And similarly with the rest of the meta-interpreter predicates.

Finally, *append* must also be updated to its new, layered-stream meaning. Now $append(Xs, Ys, Zs)$ is intended to hold iff $\overline{Zs}$ is the concatenation of $\overline{Xs}$ and $\overline{Ys}$, where Xs, Ys, and Zs are layered streams. Because some layered streams are lists, the standard definition of *append* is sound (though not complete) with respect to the new meaning. However, a careful analysis reveals that instead of writing a complete definition for this new meaning we can restrict ourselves to:

$$\begin{aligned} &append(Xs, [\,], Xs) \leftarrow \\ &append([A{*}Xs], Ys, [A{*}Xs|\,Ys]) \leftarrow \end{aligned} \tag{6}$$

Clause (6) is a property which can be proved using induction on the length of the first argument.

A goal to the meta-interpreter is: $\leftarrow list\_solve\_subgoal(p, begin, Z)$, which produces the same *intermediate* layered streams as those of the program in Figure 1 with $\leftarrow fourQueen(begin, Z)$. In fact, with unfolding it is possible to obtain the program in Figure 1. An advantage over Figure 1 is that the programmer can provide a single-answer version of the program, and need not think in terms of exhaustive search or layered streams.

Writing a layered-stream version of the N-queen problem could perhaps be found to be relatively straightforward. Writing layered-stream programs for other problems may require considerably more cunning, as shown in [4]. Tick [4, pp. 305–310] uses 38 clauses and 12 predicates for labeling polyhedra scenes. A disadvantage of our approach at this point, however, is that we are limited to chain programs. Hence, we now turn our attention to more general kinds of program.

4 Beyond chain form

So far, we have indicated how to perform exhaustive search with layered streams in programs with clauses:

$$\begin{array}{ll} & p(X_0, X_n) \leftarrow p^1(X_0, X_1),\ p^2(X_1, X_2),\ \ldots,\ p^n(X_{n-1}, X_n) \\ \text{or} & p(\langle t, Xs\rangle, [t'|Xs]) \leftarrow \mathit{test}(\langle u, Xs\rangle) \end{array}$$

It is easy to handle programs having a slightly more general form, which we will call *quasi-chain form*, where binary predicates are defined only by clauses:

$$\begin{array}{ll} & p(\langle t_0, X_0\rangle, X_n) \leftarrow p^1(\langle t_1, X_0\rangle, X_1),\ p^2(\langle t_2, X_1\rangle, X_2),\ \ldots,\ p^n(\langle t_n, X_{n-1}\rangle, X_n) \\ \text{or} & p(\langle t, Xs\rangle, [t'|Xs]) \leftarrow \mathit{test}(\langle u, Xs\rangle) \end{array}$$

where $\mathrm{var}(t_{i+1}) \subseteq \mathrm{var}(t_i)$, $\mathrm{var}(t') \subseteq \mathrm{var}(t)$, and $\mathrm{var}(u) \subseteq \mathrm{var}(t)$.

If we fix the problem size, then all four layered-stream examples in [4] can readily be written in quasi-chain form: the N-queen problem [4, p. 117], instant insanity, [4, p. 241], the "turtle problem" [4, p. 266], and Waltz's polyhedra-scene labeling [4, p. 300]. Lack of space prevents us from deriving here through partial deduction the following layered-stream version (which uses DCG notation for difference lists, and where negations have been removed by adding cuts) of a (nondeterministic) program in quasi-chain form:

Let p be a predicate symbol of a program P in quasi-chain form, and P' be the associated layered-stream program.

1. If the ith clause defining p is:

$$p(\langle t_0, X_0\rangle, X_n) \leftarrow p^1(\langle t_1, X_0\rangle, X_1),\ p^2(\langle t_2, X_1\rangle, X_2),\ \ldots,\ p^n(\langle t_n, X_{n-1}\rangle, X_n)$$

   Then P' has the clauses

$$\begin{array}{ll} \mathit{all\_p}_i(\langle t_0, \mathit{Xls}_0\rangle, \mathit{Xls}_n, \mathit{Zls}_n) \leftarrow\ !, & \%\ \text{composition} \\ & \mathit{all\_p}^1(\langle t_1, \mathit{Xls}_0\rangle, \mathit{Xls}_1, [\,]), \\ & \mathit{all\_p}^2(\langle t_2, \mathit{Xls}_1\rangle, \mathit{Xls}_2, [\,]), \\ & \ldots, \\ & \mathit{all\_p}^n(\langle t_n, \mathit{Xls}_{n-1}\rangle, \mathit{Xls}_n, \mathit{Zls}_n) \\ \mathit{all\_p}_i(\_) \rightarrow [\,] & \end{array}$$

2. If the ith clause defining p is:

$$p(\langle t, Xs\rangle, [t'|Xs]) \leftarrow \mathit{test}(\langle u, Xs\rangle)$$

   Then P' has the clauses

$$\begin{array}{ll} \mathit{all\_p}_i(\langle t, \mathit{Xls}\rangle) \rightarrow\ !,\ [t' {*} \mathit{Xls}'],\ \{\mathit{ls\_test}(\langle u, \mathit{Xls}\rangle, \mathit{Xls}')\} & \%\ \text{layered stream} \\ \mathit{all\_p}_i(\_) \rightarrow [\,] & \end{array}$$

3. Let m be the number of clauses defining p. Then P' has the clause

$$\mathit{all\_p}(X) \rightarrow \mathit{all\_p}_1(X),\ \mathit{all\_p}_2(X),\ \ldots,\ \mathit{all\_p}_m(X) \qquad \%\ \text{union}$$

Yet another generalization allows us to handle variable-size problems: see appendix A.1.

5 Concluding remarks

When writing layered-stream programs, the programmer not only faces the problem of dealing with exhaustive search, but with factorization of heads. Our approach incorporated both these problems into a meta-interpreter, relieving the programmer from being concerned about them. Applying partial evaluation to such an interpreter yields a transformation converting a non-deterministic program that computes one answer at a time into an exhaustive-search program that uses layered streams.

A disadvantage of our transformation is that it is restricted to programs in quasi-chain form (actually a slight generalization allowing variable-size problems illustrated in appendix A.1). However, we were able to easily write in this form all four layered-stream examples of [4]. Out of these four examples, two of the programs produced by our method, namely the programs for the N-queen problem and instant insanity, *construct exactly the same intermediate layered streams as those of the programs crafted manually* (up to variable renaming). This suggests that the virtues (such as the amount of parallelism) of the original programs are preserved in ours. The programs for the other two problems solved with layered streams in [4] (the turtle problem and Waltz's polyhedra-scene labeling) are incomparable with ours in this sense because of using different representations of the problem.

We have demonstrated that chain programs, together with meta-interpretation and partial deduction are useful for reasoning about exhaustive search. Here we have discovered a transformation into layered-stream form. By writing different meta-interpreters we can also show how to reconstruct the existing continuation-based and stream-based exhaustive-search methods.

Acknowledgment

We gratefully acknowledge IIMAS, UNAM for support.

A Examples

A.1 N Queens

$$
\begin{aligned}
& p(\langle 0, \mathit{Values}, \mathit{Placed}\rangle, \mathit{Placed}) \leftarrow \\
& p(\langle N, \mathit{Values}, \mathit{Placed}\rangle, \mathit{AllPlaced}) \leftarrow N > 0,\ N' \mathrel{is} N-1, \\
& \qquad\qquad q(\langle \mathit{Values}, \mathit{Placed}\rangle, \mathit{XPlaced}), \\
& \qquad\qquad p(\langle N', \mathit{Values}, \mathit{XPlaced}\rangle, \mathit{AllPlaced}) \\
& q(\langle [X|Xs], \mathit{Placed}\rangle, [X|\mathit{Placed}]) \leftarrow \mathit{safe}(\langle X, 1, \mathit{Placed}\rangle) \\
& q(\langle [X|Xs], \mathit{Placed}\rangle, \mathit{XPlaced}) \leftarrow q(\langle Xs, \mathit{Placed}\rangle, \mathit{XPlaced})
\end{aligned}
$$

Example goal: $\leftarrow p(\langle 5, [1,2,3,4,5], [\,]\rangle, P)$.

A.2 Cubes

$$c(X_0, X_4) \leftarrow \mathit{set}(\langle q(p(w,g), p(r,w), p(b,r)), X_0\rangle, X_1),$$

$$
\begin{aligned}
&\qquad\qquad set(\langle q(p(w,b),p(g,g),p(b,r)),X_1\rangle,X_2),\\
&\qquad\qquad set(\langle q(p(w,b),p(r,g),p(g,w)),X_2\rangle,X_3),\\
&\qquad\qquad set(\langle q(p(w,b),p(r,r),p(g,r)),X_3\rangle,X_4)\\
&set(\langle q(P_1,P_2,P_3),Qs\rangle,Qs') \leftarrow rotate(\langle P_1,P_2,Qs\rangle,Qs')\\
&set(\langle q(P_1,P_2,P_3),Qs\rangle,Qs') \leftarrow rotate(\langle P_2,P_1,Qs\rangle,Qs')\\
&set(\langle q(P_1,P_2,P_3),Qs\rangle,Qs') \leftarrow rotate(\langle P_1,P_3,Qs\rangle,Qs')\\
&set(\langle q(P_1,P_2,P_3),Qs\rangle,Qs') \leftarrow rotate(\langle P_3,P_1,Qs\rangle,Qs')\\
&set(\langle q(P_1,P_2,P_3),Qs\rangle,Qs') \leftarrow rotate(\langle P_2,P_3,Qs\rangle,Qs')\\
&set(\langle q(P_1,P_2,P_3),Qs\rangle,Qs') \leftarrow rotate(\langle P_3,P_2,Qs\rangle,Qs')\\
&rotate(\langle p(C_1,C_2),p(C_3,C_4),Qs\rangle,[q(C_1,C_2,C_3,C_4)|Qs]) \leftarrow check(\langle q(C_1,C_2,C_3,C_4),Qs\rangle)\\
&rotate(\langle p(C_1,C_2),p(C_3,C_4),Qs\rangle,[q(C_1,C_2,C_4,C_3)|Qs]) \leftarrow check(\langle q(C_1,C_2,C_4,C_3),Qs\rangle)\\
&rotate(\langle p(C_1,C_2),p(C_3,C_4),Qs\rangle,[q(C_2,C_1,C_3,C_4)|Qs]) \leftarrow check(\langle q(C_2,C_1,C_3,C_4),Qs\rangle)\\
&rotate(\langle p(C_1,C_2),p(C_3,C_4),Qs\rangle,[q(C_2,C_1,C_4,C_3)|Qs]) \leftarrow check(\langle q(C_2,C_1,C_4,C_3),Qs\rangle)\\
&check(\langle \_,[\,]\rangle) \leftarrow\\
&check(\langle q(A_1,B_1,C_1,D_1),[q(A_2,B_2,C_2,D_2)|Qs]\rangle) \leftarrow A_1 \backslash{==}\ A_2,\ B_1 \backslash{==}\ B_2,\\
&\qquad\qquad\qquad\qquad\qquad\qquad C_1 \backslash{==}\ C_2,\ D_1 \backslash{==}\ D_2,\\
&\qquad\qquad\qquad\qquad\qquad\qquad check(\langle q(A_1,B_1,C_1,D_1),Qs\rangle)
\end{aligned}
$$

Example goal: $\leftarrow c([\,], Qs)$.

References

[1] André Mariën and Bart Demoen. Findall without findall/3. In David S. Warren, editor, *Proceeding of the Tenth International Conference on Logic Programming*, pages 408–423. MIT Press, 1993.

[2] Akira Okumura and Yuji Matsumoto. Parallel programming with layered streams. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 224–231, San Francisco, California, U.S.A., 1987. Lecture Notes in Computer Science 348.

[3] H. Tamaki. Stream-based compilation of ground I/O Prolog into committed-choice languages. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 376–393, Melbourne, Australia, 1987.

[4] Evan Tick. *Parallel Logic Programming*. MIT Press, 1991.

[5] Kazunori Ueda. Making exhaustive search programs deterministic. *New Generation Computing*, 5:29–44, 1987.