Join Evaluation Schemata for Constraint Handling Rules (CHR)

Christian Holzbaur
University of Vienna
Department of Medical Cybernetics and Artificial Intelligence
Freyung 6, A-1010 Vienna, Austria
christian@ai.univie.ac.at

Thom Frühwirth
CWG at LMU[*]
Oettingenstrasse 67, D-80538 Munich, Germany
fruehwir@informatik.uni-muenchen.de

Abstract

CHR is a committed-choice language consisting of guarded rules that rewrite constraints into simpler ones until they are solved. CHR can define both simplification of and propagation over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints which are logically redundant but may cause further simplification. The constraints acted upon by CHR are represented in a constraint store, much like tuples in a conventional data base. The evaluation of the condition part of CHR consists of finding sets of tuples in the constraint store which match the heads of a rule, resembling the relational algebra join operation. In the binding environments of these tuple sets, the applicability of the rules is decided by evaluating the guards. Rule applications may add and remove constraints. The anticipated interaction under an immediate update view leads to a small number of prototypical join evaluation code templates. Their Prolog realization is discussed in this paper.

1 Introduction

CHR [Fru98] is a committed-choice language consisting of guarded rules with multiple head atoms. CHR define simplification of, and propagation over, constraints. Simplification rewrites constraints to simpler constraints while preserving logical equivalence (e.g. `X>Y,Y>X <=> false`). Propagation adds new constraints which are logically redundant but may cause further simplification (e.g. `X>Y,Y>Z ==> X>Z`). Repeatedly applying the rules incrementally solves

[*]Constraint Working Group at Ludwig-Maximilians-University

constraints (e.g. `A>B,B>C,C>A` leads to `false`). With multiple heads and propagation rules, CHR provide two features which are essential for non-trivial constraint handling.

In contrast to the family of the general-purpose concurrent logic programming languages [Sha89], concurrent constraint languages [Sar93] and the ALPS [Mah87] framework, CHR are a special-purpose language concerned with defining declarative objects, constraints, not procedures in their generality. In another sense, CHR are more general, since they allow for "multiple heads", i.e. conjunctions of constraints in the head of a rule. Multiple heads are a feature that is essential in solving conjunctions of constraints. With single-headed CHR alone, unsatisfiability of constraints could not always be detected (e.g `X<Y,Y<X`) and global constraint satisfaction could not be achieved. The probably most distinguishing functionality of CHR is that they act as a powerful iteration and retrieval mechanism over the constraint store, a data structure holding constraints.

CHR are typically realized as a library containing a compiler, runtime system and solvers written in CHR. The first implementation of CHR in 1991 was an interpreter written in ECL^{i}PS^e Prolog [Fru91], followed by several re-implementations and improvements, the last one being [FrBr95a]. Similar translations, i.e. compilation of committed-choice languages, have been investigated before, be it translating GHC [UeCh85], implementations of delay declarations [Nai85] or the efficient implementation of QD-Janus [Deb93]. Today, we benefit from more powerful programming constructs, in particular customizable suspension mechanisms provided by attributed variables [Neu90, Hui90, Hol90, Hol92, Hol93]. The most recent implementation of CHR exists in the form of a SICStus Prolog library module [HoFr98a].

Overview of this Paper

We quickly recapture syntax and semantics of CHR. An brief outline of the compilation process follows. Then we concentrate on the evaluation of the condition part of CHR and compare a small number of implementation alternatives, in particular we are concerned with efficient computation of joins of constraints and with the use of indices.

2 Syntax and Semantics

We assume some familiarity with (concurrent) constraint (logic) programming, e.g. [Sha89, VH91, Sar93, JaMa94]. As a special purpose language, CHR extend a host language with constraint solving capabilities. Auxiliary computations in CHR programs are executed as host language statements. For more formal and detailed syntax and semantics of CHR see [Fru98, FAM98].

2.1 Syntax

Syntax is given in EBNF grammar style.

Definition 2.1 *There are three kinds of CHR. A* simplification CHR *is of the form*[1]

```
[Name '@'] Head1,...,HeadN '<=>' [Guard '|'] Body.
```

where the rule has an optional `Name` *(a Prolog term), the multi-head* `Head1,..., HeadN` *is a conjunction of CHR constraints, which are Prolog atoms. The guard is optional; if present,* `Guard` *is a Prolog goal excluding CHR constraints; if not present, it has the same meaning as the guard* `'true |'`*. The body* `Body` *is a Prolog goal including CHR constraints*

A propagation CHR *is of the form*

```
[Name '@'] Head1,...,HeadN '==>' [Guard '|'] Body.
```

A simpagation CHR *is a combination of the above two kinds of rule, it is of the form*

```
[Name '@'] Head1,...'\'...,HeadN '==>' [Guard '|'] Body.
```

where the symbol `'\'` *separates the head constraints into two nonempty parts.*

2.2 Semantics

Declaratively[2], a rule relates heads and body, provided the guard is true. A simplification rule means that the heads are true if and only if the body is satisfied. A propagation rule means that the body is true if the heads are true.

A simpagation rule combines a simplification and a propagation rule. The rule `Heads1 \ Heads2 <=> Body` is equivalent to the simplification rule `Heads1, Heads2 <=> Body, Heads1`. However, the simpagation rule is more compact to write, more efficient to execute and has better termination behaviour than the corresponding simplification rule.

In this paper, we are interested in a particular aspect of the operational semantics of CHR in an actual implementation. A CHR constraint is implemented as both code (a Prolog predicate) and as data in the constraint store. Every time a CHR constraint is executed (called) or woken (reconsidered), it checks itself the applicability of its associated CHR. Such a constraint is called *(currently) active*, while the other constraints in the constraint store that are not executed at the moment are called *(currently) passive*. For each CHR, one of its heads is matched against the constraint[3]. Matching succeeds if the constraint is an instance of the head, i.e. the head serves as a pattern. If a CHR has more than one head, the constraint store is searched for *distinct partner* constraints that match the other heads. If the matching succeeds, the guard is executed. Otherwise the next rule is tried.

[1]For simplicity, we omit syntactic extensions like pragmas which are not relevant for this paper.

[2]Unlike general committed-choice programs, CHR programs can be given a declarative semantics since they are only concerned with defining constraints, not procedures in their generality.

[3]Variables in `Head` (but not those in `Constraint`) will be bound, so that `Head` becomes identical to `Constraint`

The guard either succeeds or fails. A guard succeeds if the execution succeeds without *touching* a variable that occurs also in the heads. A variable is *touched* if it is unified with a non-variable term or a variable appearing in a CHR constraint or if it is the cause of an instantiation error. If the guard succeeds, the rule applies. Otherwise the next rule is tried.

If the firing CHR is a simplification rule, the matched constraints are removed from the store and the body of the CHR is executed. Similarly for a firing simpagation rule, except that the constraints that matched the heads preceding '\' are kept. If the firing CHR is a propagation rule the body of the CHR is executed without removing any constraints. It is remembered that the propagation rule fired, so it will not fire again with the same constraints if the constraint is woken. Since the currently active constraint has not been removed, the next rule is tried.

If all rules have been tried and the active constraint has not been removed, it suspends (delays) until a variable occurring in the constraint is touched. Here suspension means that the constraint is inserted into the constraint store as data. When a constraint is woken, all its rules are tried again.

Example 2.1 (Cycle) *The following rule finds all cycles of length five in a graph encoded through a collection of directed edges.*

```
:- use_module( library(chr)).

handler cycle.

constraints edge/2, loop/1.

edge( A, B),
   edge( B, C),
      edge( C, D),
         edge( D, E),
            edge( E, A) ==> loop([A,B,C,D,E]).

%
% Given these edges,
%
edge(1,4), edge(1,9), edge(2,8), edge(3,10), edge(5,1),
edge(5,8), edge(7,4), edge(7,5), edge(7,10), edge(8,3),
edge(8,9), edge(9,3), edge(10,7).

%
% the rule adds the following constraints to the store:
%
loop([3,10,7,5,8])
loop([8,3,10,7,5])
loop([5,8,3,10,7])
loop([7,5,8,3,10])
loop([10,7,5,8,3])
```

3 The CHR Compilation Process

We compile CHR into Prolog predicates. The compilation process is global in the sense that we have to account for each constraint in each of the two possible roles (active,passive) in all rules where it occurs as a head. Roughly, the generated code threads the flow of control through the match-attempts, evaluates the guards, updates the constraint store depending on the rule type, and evaluates the rule bodies [HoFr98b].

3.1 Join Computation

The realization of the condition part of CHR consists of finding sets of tuples in the constraint store which match the heads of a rule, resembling the relational algebra join operation. In the binding environments of these tuple sets, the applicability of a rule is decided by evaluating its guard. The situation is quite similar to the matching phase in rule/production systems. The earlier predominant state-preserving RETE match algorithm [For82] was redeemed by the superior state-less TREAT algorithm [Mir87]. State preservation is even more of debatable utility in the presence of guards. Thus, the CHR compilation draws upon a state-less incremental matching mechanism.

In the terminological framework of [LiOk87] CHR operate under the "immediate update view": at the time a rule commits, the constraints to be removed as indicated by the rule type are gone. Constraints added by a rule's body are visible to the currently active rule(s) immediately. There are three prototypical cases:

1. The active constraint is removed by the rule. Independent from the number of partner constraints, the rule will apply at most once, and no further rules will have to be tried since the active constraint can no longer participate in a join.

2. The active constraint is kept, some partner constraints are removed. Since the active constraint is kept, one has to continue looking for applicable rules, even if the rule applies. However, since at least one partner constraint will have been removed, the same rule will only be applicable again with another constraint from the store that matches the same partner head.

3. The active constraint and all partner constraints are kept. This is the most expensive case. The full crossproduct of constraints as indicated by the rule heads has to be generated. Further rules will have to be tried since the active constraint is not removed.

Example 3.1 (SQL) *Here is the SQL select statement for the condition part of our rule from example 2.1. It would be executed against a relational data base holding the graph(s) in a relation/table* `edge`*. The virtual column* `rowid`[4] *uniquely identifies each tuple from the relation, which enables us to specify the distinctness of the partner constraints.*

[4]ORACLE

```
SELECT e1.*, e2.to, e3.to, e4.to
  FROM edge e1, edge e2, edge e3, edge e4, edge e5
 WHERE e1.to = e2.from
   and e2.to = e3.from
   and e3.to = e4.from
   and e4.to = e5.from
   and e5.to = e1.from
   and e1.rowid <> e2.rowid and e1.rowid <> e3.rowid
   and e1.rowid <> e4.rowid and e1.rowid <> e5.rowid
   and e2.rowid <> e3.rowid and e2.rowid <> e4.rowid
   and e2.rowid <> e5.rowid and e3.rowid <> e4.rowid
   and e3.rowid <> e5.rowid and e4.rowid <> e5.rowid
```

As the target language for CHR, within the scope of this paper, is Prolog[5], the select statement tells us *what* to compute, but it is not directly executable.

The most direct join computation code with a deterministic recursive loop per partner covers all three cases from above correctly, but for all but the last we can do better than that — at least with respect to the amount of code generated, which is an issue, as you can figure from output for the single propagation rule (case 3) from example 2.1, listed in the appendix. The code template employs one deterministic recursive predicate per partner constraint. A runtime predicate (`init_iteration/4`) provides data for these loops in the form os lists of constraints for a given `F/A`. Argument matching is performed inside the loops, and the environment for the guard and body evaluation is gradually accumulated and passed via predicate arguments to the innermost loop.

A second join computation scheme is applicable if at least one constraint gets removed by the rule (cases 1 and 2): Instead of deterministic recursion for each partner, we find individual partners nondeterministically within a single predicate. Once a tuple from the join is assembled, the rule's body is executed and we restart the nondeterministic search, until the generators are exhausted.

Example 3.2 (Cycle, modified) *Let us turn the propagation rule into a simpagation rule by adding a head to be removed (*`dummy`*):*

```
edge( A, B),
   edge( B, C),
      edge( C, D),
         edge( D, E),
            edge( E, A) \ dummy <=> loop([A,B,C,D,E]).
```

Example 3.3 (Generated Prolog code for 3.2) *Nondeterministic iteration for a constraint of given* `F/A` *is via* `init_iteration_nd/5`*. The code makes references to the use of indices, which are introduced in section 3.2.*

```
edge(A, B) :- 'chr_edge/2_1'(A, B, _).

'chr_edge/2_1'(A, B, C) :-                  % first head is active
        insert(C, user:edge(A,B)),          % put into constraint store
```

[5]allowing variables in the extensional data base, which has an impact on expressiveness from the language point of view, and a more technical one: see section 3.2

```
        'chr_edge/2_1_forall_1'(A, B, C),
        'chr_edge/2_2'(A, B, C).            % similar code for other
                                            % active heads
'chr_edge/2_1_forall_1'(A, B, C) :-
        via([B], D),                        % use index
        init_iteration_nd(D, edge, 2, E, F),
        F=edge(G,H),
        C\==E, B==G,
        via([H], I),                        % use index
        init_iteration_nd(I, edge, 2, J, K),
        K=edge(L,M),
        E\==J, C\==J, H==L,
        via([M], N),                        % use index
        init_iteration_nd(N, edge, 2, O, P),
        P=edge(Q,R),
        J\==O, E\==O, C\==O, M==Q,
        via([A,R], S),                      % use index
        init_iteration_nd(S, edge, 2, T, U),
        U=edge(V,W),
        O\==T, J\==T, E\==T, C\==T, R==V, A==W,
        via([], X),                         % no index applies
        init_iteration_nd(X, dummy, 0, Y, Z),
        Z=dummy,
        eval_guard(true, user), !,          % commit
        remove_constraint(Y),
        loop([A,B,H,M,R]),                  % rule body
        'chr_edge/2_1_forall_1'(A, B, C).   % restart
'chr_edge/2_1_forall_1'(_, _, _).
```

The nondeterministic formulation produces more compact code. Differences with respect to the WAM [Ait90] are that we trade choice point allocation against environment allocation. The relative speed of the two approaches depends on the particular Prolog system hosting CHR. In our case, a slight advantage of the recursive version was mostly overcompensated by the time required for garbage collection.

In figure 1 we compare the recursive and backtracking join computation. Random graphs with between 10 and 200 edges where fed through the rule from example 2.1. Each data point represents mean and standard deviation from 10 experiments. The recursive and the backtracking code operated on the same 10 random graphs each. The vertical axis represents runtime in seconds including garbage collection[6]. $n = 200$ edges sounds like a small graph, but to find all cycles of length five, we may have to look at $\binom{n}{5}$ edge combinations.

Case 1 of the three prototypical sorts of rules from above, where the current constraint is removed by the rule, does not even call for the restart of the nondeterministic search after one tuple from the join has been assembled, and is thus most compact to compile and cheapest to run.

[6]predicate statistics(walltime, _) in SICStus

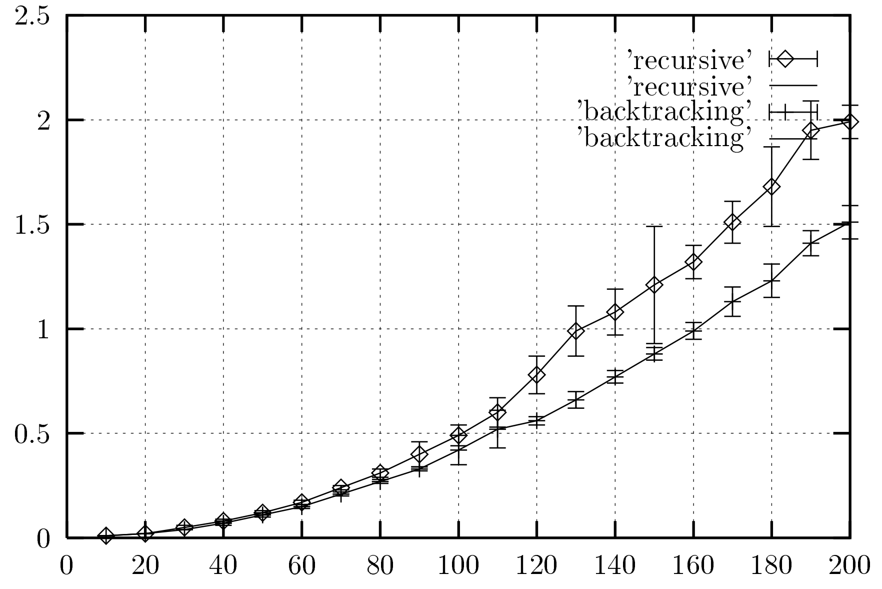

Figure 1: Recursive vs. backtracking join computation

3.2 The Use of Indices

A variable common to two heads of a rule considerably restricts the number of candidate constraints to be considered, because both partners must be suspending on the corresponding variable. Thus we access the constraint store by looking at only those constraints. The variables shared between partner constraints *index* the constraint store. Like with traditional data bases, the index may speed up join computations. References to this index are through the predicate `via/2` in the generated code (example 3.3). The first argument is the list of shared variables between the head for which the iteration is to be initiated and the heads matched so far. At runtime, this list will be instantiated by subsumed constraints. If it still contains a variable, it is the one we use to locate the candidates for the next match.

Example 3.4 (Graph, nonground) *We keep the rule from example 2.1 as it is, and change the graph representation. Instead of ground vertices, we use variables:*

```
edge(X1,X4), edge(X1,X9), edge(X2,X8), edge(X3,X10), edge(X5,X1),
edge(X5,X8), edge(X7,X4), edge(X7,X5), edge(X7,X10), edge(X8,X3),
edge(X8,X9), edge(X9,X3), edge(X10,X7).

%
% the rule outputs:
%
loop([X3,X10,X7,X5,X8])
loop([X8,X3,X10,X7,X5])
loop([X5,X8,X3,X10,X7])
loop([X7,X5,X8,X3,X10])
loop([X10,X7,X5,X8,X3])
```

In figure 2 we repeat the experiment from figure 1. The non-ground graph representation allows for the utilization of the mentioned index during the join computation. The difference in computation time is two orders of magnitude. The difference between the deterministic and nondeterministic versions is rather insignificant.

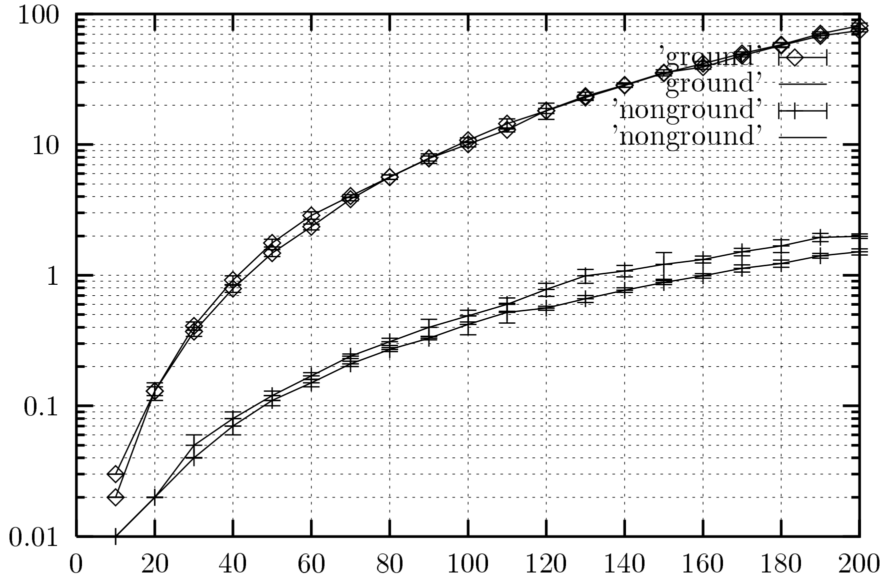

Figure 2: Join computation with and without indexing

4 Conclusions

Join computation schemata as part of the compilation of CHR to Prolog conceptually overlap with the realization of the matching phase in rule/production systems. The presence of guards and their established realization(s) in the target language corroborate the utilization of state-less matching mechanisms. Active treatment of the constraint update semantics allows for the generation of more compact code. Join evaluation is facilitated through an index mechanism specific to the Prolog embedding of CHR.

Acknowledgements

The Austrian Research Institute for Artificial Intelligence is supported by the Federal Ministry of Science and Transport.

References

[Ait90] Ait-Kaci H.: The WAM: A (Real) Tutorial, Digital Equipment, Paris, 1990.

[CoDi93] Diaz D., Codognet P, A Minimal Extension of the WAM for clp(FD), in Warren D.S.(Ed.), Proceedings of the Tenth International Conference on Logic Programming, The MIT Press, Budapest, Hungary, pp.774-790, 1993.

[Deb93] S. K. Debray, QD-Janus : A Sequential Implementation of Janus in Prolog, Software—Practice and Experience, Volume 23, Number 12, December 1993, pp. 1337-1360.

[For82] Forgy C.L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, Artificial Intelligence, 19(1)17-37, 1982.

[FrBr95a] T. Frühwirth and P. Brisset, High-Level Implementations of Constraint Handling Rules, Technical Report ECRC-95-20, ECRC Munich, Germany, June 1995.

[FAM98] T. Frühwirth, S. Abdennadher and H. Meuss, Confluence and Semantics of Constraint Simplification Rules, Constraint Journal, Kluwer Academic Publishers, to appear 1998.

[Fru91] T. Frühwirth, Introducing Simplification Rules, Technical Report ECRC-LP-63, ECRC Munich, Germany, October 1991.

[Fru98] T. Frühwirth, Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.), Journal of Logic Programming, Vol 37(1-3), pp 95-138, October 1998.

[Hol90] Holzbaur C, Specification of Constraint Based Inference Mechanisms through Extended Unification, Department of Medical Cybernetics and Artificial Intelligence, University of Vienna, Dissertation, 1990.

[Hol92] C. Holzbaur, Metastructures vs. Attributed Variables in the Context of Extensible Unification, In 1992 International Symposium on Programming Language Implementation and Logic Programming, pages 260–268. LNCS631, Springer Verlag, August 1992.

[Hol93] C. Holzbaur, Extensible Unification as Basis for the Implementation of CLP Languages, in Baader F., et al., *Proceedings of the Sixth International Workshop on Unification*, Boston University, MA, TR-93-004, pp.56-60, 1993.

[HoFr98a] Ch. Holzbaur and Th. Frühwirth, Constraint Handling Rules Reference Manual, for SICStus Prolog, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, Austria, TR-98-01, March 1998.

[HoFr98b] Ch. Holzbaur and Th. Frühwirth, Compiling Constraint Handling Rules, ERCIM/COMPULOG Workshop on Constraints, CWI, Amsterdam, The Netherlands, 1998.

[Hui90] Huitouze S.le, A new data structure for implementing extensions to Prolog, in Deransart P. and Maluszunski J.(Eds.), Programming Language Implementation and Logic Programming, Springer, Heidelberg, 136-150, 1990.

[JaMa94] J. Jaffar and M. J. Maher, Constraint Logic Programming: A Survey, Journal of Logic Programming, 1994:19,20:503-581.

[LiOk87] Lindholm T. O'Keefe R.A.: Efficient Implementation of a Defensible Semantics for Dynamic PROLOG Code, in Lassez J.L.(ed.), Proceedings of the Fourth International Conference on Logic Programming - Volume 1, MIT Press, Cambridge, MA, pp.21-39, 1987.

[Mah87] Maher M. J., Logic Semantics for a Class of Committed-Choice Programs, Fourth Intl Conf on Logic Programming, Melbourne, Australia, MIT Press, pp 858-876.

[Mir87] Miranker D.P.: TREAT: A Better Match Algorithm for AI Production Systems, in Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI- 87), Morgan Kaufmann, Los Altos, CA, pp.42-47, 1987.

[Nai85] L. Naish, Prolog control rules, Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, California, September 1985, pp. 720-722.

[Neu90] U. Neumerkel, Extensible unification by metastructures, In Proc. of Metaprogramming in Logic (META'90), Leuven, Belgium, 1990.

[Sar93] V. A. Saraswat, Concurrent Constraint Programming, MIT Press, Cambridge, 1993.

[Sha89] E. Shapiro, The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 21(3):413-510, September 1989.

[UeCh85] Ueda K., Chikayama T, Concurrent Prolog Compiler on Top of Prolog, in Symposium on Logic Programming, The Computer Society Press, pp.119-127, 1985.

[VH91] P. Van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, Vol 6:3, 1991, pp 151-194.

A Deterministic Code for Example 2.1

`init_iteration/4` deterministically returns a list of constraints for given `F/A`. Matching is performed in recursive loops. The environment for the guard and body evaluation is gradually accumulated and passed via arguments to the innermost loop.

```
edge(A, B) :-
        'chr_edge/2_1'(A, B, _).

'chr_edge/2_1'(A, B, C) :-              % first head is active
        insert(C, user:edge(A,B)),      % put into constraint store
        via([B], D),                    % use index
        init_iteration(D, edge, 2, E),
        'chr_edge/2_1_forall_1'(E, B, C, A),
        'chr_edge/2_2'(A, B, C).

'chr_edge/2_2'(A, B, C) :-
        %
        % similar code for further heads being
        % active
        %

'chr_edge/2_1_forall_1'([], _, _, _).
'chr_edge/2_1_forall_1'([A|B], C, D, E) :-
        state(A, F, G),
        G=edge(H,I),
        (   F==active,
            D\==A,
            C==H ->
            via([I], J),                % use index
            init_iteration(J, edge, 2, K),
            'chr_edge/2_1_forall_2'(K, C, D, E, A, I)
        ;   true
        ),
        'chr_edge/2_1_forall_1'(B, C, D, E).

'chr_edge/2_1_forall_2'([], _, _, _, _, _).
'chr_edge/2_1_forall_2'([A|B], C, D, E, F, G) :-
        state(A, H, I),
        I=edge(J,K),
        (   H==active,
            F\==A,
            D\==A,
            G==J ->
            via([K], L),                % use index
            init_iteration(L, edge, 2, M),
            'chr_edge/2_1_forall_3'(M, C, D, E, F, G, A, K)
        ;   true
        ),
        'chr_edge/2_1_forall_2'(B, C, D, E, F, G).

'chr_edge/2_1_forall_3'([], _, _, _, _, _, _, _).
'chr_edge/2_1_forall_3'([A|B], C, D, E, F, G, H, I) :-
        state(A, J, K),
        K=edge(L,M),
        (   J==active,
            H\==A,
            F\==A,
            D\==A,
            I==L ->
            via([E,M], N),              % use index
            init_iteration(N, edge, 2, O),
            'chr_edge/2_1_forall_4'(O, C, D, E, F, G, H, I, A, M)
        ;   true
        ),
        'chr_edge/2_1_forall_3'(B, C, D, E, F, G, H, I).

'chr_edge/2_1_forall_4'([], _, _, _, _, _, _, _, _, _).
'chr_edge/2_1_forall_4'([A|B], C, D, E, F, G, H, I, J, K) :-
        state(A, L, M),
        M=edge(N,O),
        (   L==active,
            J\==A,
            H\==A,
            F\==A,
            D\==A,
            K==N,
            E==O ->
            (   novel_production([D,F,H,J,A], 1, P),
                eval_guard(true, user) ->
                true,
                record_production(P),
        loop([E,C,G,I,K])
    ;   true
    )
;   true
),
'chr_edge/2_1_forall_4'(B, C, D, E, F, G, H, I, J, K).
```