A Prolog Constraint Handling Rules Compiler and Runtime System

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
Ludwig-Maximilians-University
Department of Computer Science
Oettingenstrasse 67, D-80538 Munich, Germany
fruehwir@informatik.uni-muenchen.de

September 15, 1999

Abstract

We introduce the most recent and advanced implementation of constraint handling rules (CHR) in a logic programming language. The Prolog implementation consists of a runtime system and a compiler. The runtime system utilizes attributed variables for the realization of the constraint store with efficient retrieval and update mechanisms. Rules describing the interactions between constraints are compiled into Prolog clauses by a compiler, the core of which comprises a small number of compact code generating templates in the form of definite clause grammar rules.

1 Introduction

In the beginning of constraint logic programming (CLP), constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language. While efficient, this so-called "black-box" approach makes it hard to modify a solver or build a solver over a new domain, let alone debug, reason about and analyze it. This is a problem, since one lesson learned from practical applications is that constraints are often heterogeneous and application-specific. Consequently, several proposals have been made to allow for more flexibility and customization of constraint systems ("glass-box" or even "no-box" approaches):

- Demons, forward rules and conditionals in CHIP [Di*88] allow the definition of propagation of constraints in a limited way.
- Constraint combinators in cc(FD) [vHSD92] allow to build more complex constraints from simpler constraints.

[*]Part of the work was performed while visiting CWG at LMU with financial support from DFG.

- Constraints connected to a Boolean variable in BNR-Prolog [BeOl92] and "nested constraints" [Sid93] allow to express any logical formula over primitive constraints.
- Indexicals in clp(FD) [CoDi93] allow to implement constraints over finite domains at a medium level of abstraction.
- Meta- and attributed variables [Neu90, Hui90, Hol92] allow to attach constraints to variables at a low level of abstraction.

It should be noted that all the approaches but the last can only extend a solver over a given, specific constraint domain, typically finite domains. The expressive power to realize other (application-specific) constraint domains is only provided by the last approach.

Attributed variables [Hol92] provide direct access storage locations for properties associated with variables. When such variables are unified, their attributes have to be manipulated. Thus attributed variables make unification user-definable [Hol90, Hol93]. Attributed variables require roughly the same implementation effort as hard-wired delay (suspension) and coroutining mechanisms found in earlier Prolog implementations, while being more general. And indeed, attributed variables nowadays serve as the primary low-level construct for implementing suspension (delay) mechanisms and constraint solver extensions in many constraint logic programming languages, e.g. SICStus [CaWi95] and ECL^{i}PS^e [Br*98] Prolog. However writing constraints this way is tedious, a kind of "constraint assembler" programming.

If there already is a powerful constraint assembler, one may wonder what an associated high-level language could look like. Our proposal is a declarative language designed for writing constraint solvers, called constraint handling rules (CHR) [FrBr95b, Fru98, HoFr98a, FAM99]. With CHR, one can introduce user-defined constraints into a given high level host language, be it Prolog or Lisp. CHR have been used in dozens of projects worldwide to encode constraint handlers (solvers), including new domains such as terminological and temporal reasoning [Fru98].

CHR is essentially 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. CHR can be seen as a generalization of the various CHIP [Di*88] constructs for user-defined constraints.

In contrast to the family of the general-purpose concurrent logic programming languages [Sha89], concurrent constraint languages [Sar93] and the ALPS [Mah87] framework, CHR allow for *multiple heads*, i.e. conjunctions of constraints in the head of a rule, and *propagation rules*. Multiple heads are a feature that is essential in solving conjunctions of constraints. With single-headed CHR alone, unsatisfiability of constraints cannot 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, retrieval, and update mechanism over the so-called *constraint store*, the data structure holding constraints.

Besides defining the behaviour of constraints, CHR can be and have been used as

- general purpose concurrent constraint language with ask and tell,
- as fairly efficient production rule system,
- as a special kind of theorem prover,

- in general as system combining forward and backward chaining.

The first implementations of CHR were interpreters: In 1991 in ECL^{i}PS^e Prolog, in 1993 in Common LISP [Her93]. In 1994, the first compiler was written as a library of ECL^{i}PS^e [FrBr95a, FrBr95b]. An interpreter was written in the concurrent logical object-oriented constraint language OZ [SmTr94] in 1996. Independent of our work, a new experimental prototype of CHR has been implemented in ECL^{i}PS^e 4.0 [She98] in 1998.

Nowadays, CHR are typically realized as a library containing a compiler, runtime system and dozens of constraint solvers written in CHR. Rules compile into Prolog clauses which inspect and update the constraint store at runtime. Such a type of compilation of committed-choice languages into Prolog has 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. CHR specific topics are multiple heads (multi-heads) and propagation rules. The new idea is to realize the CHR constraint store through attributed variables. So CHR can also be understood as a powerful means to manipulate the attributes of variables in a declarative high-level fashion.

In this paper we describe the most recent and advanced implementation of CHR in SICStus Prolog [HoFr98a], which improves both on the previous implementation [FrBr95b] in terms of completeness, flexibility and efficiency and on the principles that should guide such an implementation [FrBr95a]. For the user, the new release of CHR improves over older versions in the following aspects:

- The number of heads in a rule is no longer limited to two.
- Guards now with Ask and Tell as in concurrent constraint languages.
- For more control, rules are compiled in textual order.
- Improved set of built-in predicates for advanced CHR users.
- New options and pragmas for powerful compiler optimizations.
- Compilation is now transparent to the user, on-the-fly when loading.
- Constant time access to constraints.
- Code runs generally about twice as fast as in older versions.
- The runtime system includes a stepper for Prolog-like debugging.

Two examples will guide us through the paper. Even though they do not define typical constraints, we chose them for didactic reasons. They are small but can still illustrate various considerations and stages of our compilation scheme. We use Prolog syntax in this paper.

Example 1.1 (Primes) *We implement the sieve of Eratosthenes to compute primes in a way reminiscent of the "chemical abstract machine" [BCL88]: The constraint* `candidates(N)` *generates candidates for prime numbers,* `prime(M)`*, where* `M` *is between* `1` *and* `N`*. The candidates react with each other such that each number absorbs multiples of itself. In the end, only prime numbers remain.*

```
candidates(1) <=> true.
generate @ candidates(N) <=> N>1 | M is N-1, prime(N), candidates(M).

sieve @ prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

The first rule says that the number `1` *is not a good candidate for a prime,* `candidates(1)` *is thus rewritten into* `true`*, a constraint that is always satisfied and therefore it has no effect. Note that head matching is used in CHR so the first rule will only apply to* `candidates(1)`*. A constraint for* `candidates` *with a free variable, like* `candidates(X)`*, will suspend (delay).*

The `generate` *rule generates a candidate* `prime(N)` *and proceeds recursively with the next smaller number, provided the guard (precondition, test)* `N>1` *is satisfied.*

The third, multi-headed rule named `sieve` *reads as follows: If there is a constraint* `prime(I)` *and some other constraint* `prime(J)` *such that* `J mod I =:= 0` *holds, i.e.* `J` *is a multiple of* `I`*, then keep* `prime(I)` *but remove* `prime(J)` *and execute the body of the rule,* `true`*.*

Example 1.2 (Cycle) *The following rule finds all cycles of length five in a graph encoded through a collection of directed edges.*

```
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]).
```

Overview of the Paper

We quickly recapture syntax and semantics for CHR. Then we describe the three phases of the new compilation scheme and the runtime system for CHR. We conclude with a comparison with the previous implementation. This paper is a revised version of [HoFr99a, HoFr98b].

2 Syntax and Semantics

We assume some familiarity with (concurrent) constraint (logic) programming, e.g. [Sha89, vHSD92, Sar93, JaMa94, FrAb97, MaSt98]. As a special purpose language, CHR extend a host language with (extended) constraint solving capabilities. Auxiliary computations in CHR programs are executed as host language statements. Here the host language is (SICStus) Prolog. For more formal and detailed syntax and semantics of constraint handling rules see [Fru98, FAM99].

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 `Name`*, which is a Prolog term, and 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*

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

```
[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.*

A simpagation rule combines simplification and propagation in one rule. The rule `HeadsK \ HeadsR <=> Body` is equivalent to the simplification rule `HeadsK, HeadsR <=> HeadsK, Body`, i.e. `HeadsK` is kept while `HeadsR` is removed. However, the simpagation rule is more compact to write, more efficient to execute and has better termination behaviour than the corresponding simplification rule.

2.2 Semantics

In this paper, we are interested in the operational semantics of CHR in actual implementations. A CHR constraint is implemented as both *code* (a Prolog predicate) and *data* (a Prolog term in the constraint store). Every time a CHR constraint is posted (executed) or woken (reconsidered, re-executed), it checks itself the applicability of the rules it appears in. 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.*

Heads. For each rule, one of its heads is matched against the constraint. Matching succeeds if the constraint is an instance of the head, i.e. the head serves as a pattern. If matching succeeded and a rule has more than one head, the constraint store is searched for *partner* constraints that match the other heads. If the matching succeeds, the guard is executed. Otherwise the next rule is tried.

Guard. A guard is a precondition on the applicability of a rule. The guard either succeeds or fails. A guard succeeds if the execution succeeds without causing an instantiation error and without *touching* a variable from the heads. A variable is *touched* if gets more constrained by a built-in constraint. If the guard succeeds, the rule applies, one commits to it and it fires. Otherwise it fails and the next rule is tried.

Body. 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. When the currently active constraint has not been removed, the next rule is tried.

(Re-)Suspension. 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.

3 The Compiler

The compiler is written in (SICStus) Prolog [HoFr98a] and translates CHR into Prolog on-the-fly, while the file is loaded (consulted). Its kernel consists of a definite clause grammar that generates the target instructions (clauses) driven by *templates.* We will use example 1.1 to explain the three phases of the compiler:

1. Parsing,

2. translating CHR into clauses using templates and
3. partial evaluation using macros.

Phase 2 is the essential one that encodes the algorithm.

3.1 Parsing Phase

Using the appropriate operator declarations, a CHR can be read and written as a Prolog term. Hence parsing basically reduces to computing information from the parse tree and to producing a canonical form of the rules. Information needed from the parse tree includes:

- The set of global variables, i.e. those that appear in the heads of a rule.
- The set of variables shared between the heads.

In the canonical form of the rules,

- each rule is associated with a unique identifier,
- rule heads are collected into two lists (named `Keep` and `Remove`), and
- guard and body are made explicit with defaults applied.

One list, called `Keep`, contains all head constraints that are kept when the rule is applied, the other list, called `Remove`, contains all head constraints that are removed. One list may be empty. As a result of this representation, simplification, propagation and simpagation rules can be treated uniformly.

Example 3.1 (Primes, contd.) *The canonical form of the rules for the prime number example is given below.*

```
% rule(Id,Keep,       Remove,          Guard,         Body)

  rule(1, [],         [candidates(1)], true,          true).
  rule(2, [],         [candidates(A)], A>1, (B is A-1,prime(A),candidates(B))).
  rule(3, [prime(A)],[prime(B)],       B mod A =:= 0, true).
```

3.2 Translation Phase

Each occurrence of a CHR constraint in the head of a rule gives rise to one Prolog clause for that constraint. The clause head contains the active constraint, while the clause body does the following:

- match formal parameters to actual arguments of head constraint
- find and match passive partner constraints in constraint store
- check the guard
- commit via cut
- remove matched constraints from constraint store if required
- execute body of rule

We first illustrate the compilation with a simple example, a single-headed simplification CHR, then we consider general cases of arbitrary multi-headed rules.

Example 3.2 (Primes, contd.) *For the constraint* candidates/1 *the compiler generates the following intermediate code (edited for readability).*

```
% for each occurrence of the constraint as a head of a rule:

% in rule candidates(1) <=> true
candidates(A) :-                        % 1
        match([1], [A]),                % 2
        check_guard([], true),          % 3
        !,                              % 4
        true.                           % 5

% in rule candidates(N) <=> N>1 | M is N-1, prime(N), candidates(M)
candidates(A) :-                        % 6
        match([C], [A]),                % 7
        check_guard([C], C>1),          % 8
        !,                              % 9
        D is C-1,                       % 10
        prime(C),                       % 11
        candidates(D).                  % 12

% if no rule applied, suspend the constraint on its variables
candidates(A) :-                        % 13
        suspend(candidates(A)).         % 14
```

The predicate `match(L1,L2)` *matches the actual arguments (in list* `L2`*) against the formal parameters (in list* `L1`*). The predicate* `check_guard(VL,G)` *checks the guard* `G`*.* `check_guard/2` *fails as soon as the global variables (list* `VL`*) are touched*[2]*.*

When no rule applied, the last clause inserts the constraint into the constraint store using a suspension mechanism. It allocates the suspension data structure and associates it with each variable occurring in the constraint. Touching any such variable will wake the constraint.

Join Computation for Finding Partner Constraints

The real challenge left is to implement *multi-headed* CHR. In a naive implementation of a rule, the constraint store is queried for the cross-product of matching head constraints. For each tuple in the cross-product the guard is checked in the corresponding environment. If the guard is satisfied, constraints that matched heads in the `Remove` list are removed from the store and the instance of the rule's body is executed. Note that the removal of constraints removes tuples from the cross-product. 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 debatable in the presence of guards. Thus, the CHR compilation draws upon a state-less incremental matching mechanism.

There are two design alternatives for the join computation: Either a *deterministic recursive loop* or a *nondeterministic backtracking search* for the partner constraints, in case at least one constraint gets removed.

The direct join computation code template employs one deterministic recursive predicate per partner constraint. A runtime predicate (`init_iteration/4`) provides data for these loops in the form of lists of constraints for a given functor and arity `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.

[2]In most Prolog implementations, it is more efficient to re-execute head matching and guards instead of suspending all of them and executing them incrementally.

The second backtracking join computation scheme is applicable if at least one constraint gets removed by the rule: Instead of deterministic recursion for each partner, we find individual partners nondeterministically within a single predicate. The nondeterministic formulation produces more compact code. In terms of the underlying WAM [Ait90] we trade environment allocation against choice point 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 overcompensated by the time required for garbage collection.

In figure 1 we compare the recursive and backtracking join computation. 10 random graphs with 10 to 200 edges were fed through the rule from example 1.2. $n = 200$ edges means that in order to find all cycles of length five, we may have to look at $\binom{n}{5}$ edge combinations. Each data point represents mean and standard deviation from 10 experiments. The recursive and the backtracking code operated on the same 10 random graphs. The vertical axis represents runtime in seconds including garbage collection and operating system management time[3].

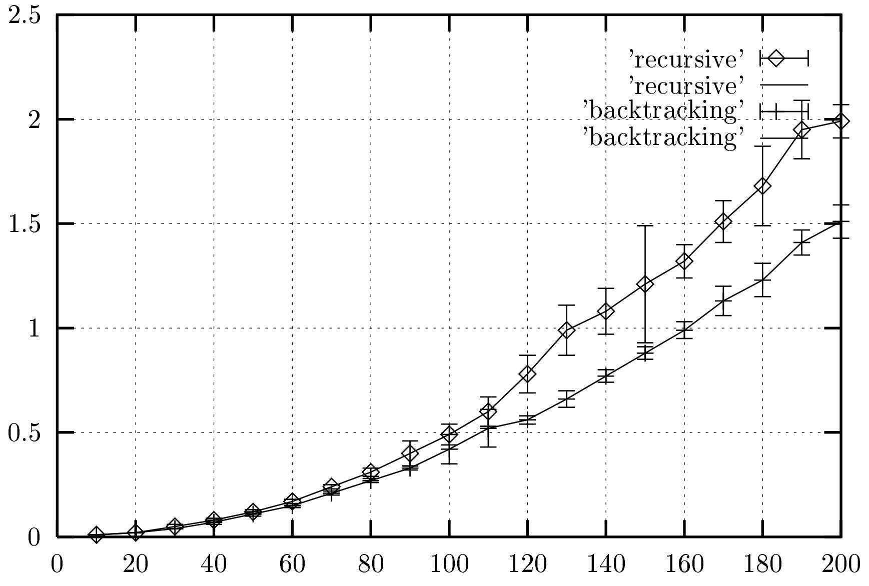

Figure 1: Recursive vs. backtracking join computation

Summarizing, our implementation computes only those tuples in the cross-product that are really needed (as in [FrBr95a]). Nondeterministic enumeration of the constraints is preferred over deterministic iteration whenever possible, because Prolog is good at backtracking [HoFr98b].

Compilation Templates

Whether the active constraint is removed when a given rule applies and whether any head constraints are removed, leads to the following three prototypical cases, each covered by a *code generating template* in the compiler:

1. Case Active constraint from `Remove` list
2. Case Active constraint from `Keep` list, `Remove` list nonempty
3. Case Active constraint from `Keep` list, `Remove` list empty

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

Case 1. Active constraint from `Remove` **list**

The active head constraint is to be removed if the rule applies, so the rule under consideration is either a simplification or simpagation rule. It can be applied at most once with the current active constraint. The search for the partner constraints in this case can be performed through nondeterministic enumeration. Here is the template, slightly abridged. The predicate `ndmpc` generates the code to nondeterministically enumerate and match the partners, one by one.

```
compile(remove(Active), Remove, Keep, Guard, Body, ...
        % generated code
        ((constraint(head(F/A,R-N), args(Actual)) :-
                   match(Args, Actual),
                   RemoveCode,                % Identify Remove partners
                   KeepCode,                  % Identify Keep partners
                   check_guard(Vars, Guard),
                   !,
                   remove_constraints(RemCs),
                   Body
        ))
        ) :-
        % compiler code
          Active =.. [_|Args],
          same_length(Args, Actual),
          ...
          ndmpc(Remove, RemoveCode, RemCs, ...),
          ndmpc(Keep, KeepCode, ...).
```

The variables `F`,`A`,`R` and `N` stand for functor, arity of the constraint, rule identifier and number of head in rule, respectively.

Example 3.3 (Primes, contd.) *The second occurrence of prime/1 in rule 3 of Example 1.1 matches this template, and here is its instantiation:*

```
% prime(I) \ prime(J) <=> J mod I =:= 0 | true.

constraint(head(prime/1,3-2), args([A])) :-
        match([C], [A]),
    % RemoveCode (for one partner constraint)
        get_constr_via([], Constraints),
        nd_init_iteration(Constraints, prime/1, Candidate),
        get_args(Candidate, [F]),
        match([C]-[G], [C]-[F]),
    % KeepCode (no partner constraints to be kept in this case)
        true,
    % Guard
        check_guard([G,C], (C mod G =:= 0)),
        !,
        remove_constraints([]),  % no constraints to remove here
    % Body
        true.
```

The predicate `get_constr_via(VL,Cs)` *returns the constraints suspended on a free variable occurring in the list* `VL`*. If there is no variable in* `VL`*, it returns all the constraints in the store.* `nd_init_iteration(Constraints, F/A, Candidate)` *nondeterministically returns a candidate constraint with functor* `F` *and arity* `A` *from the constraint store.*

Case 2. Active constraint from `Keep` list, `Remove` list nonempty

This case applies only if there is at least one constraint to be removed, but the active constraint will be kept. It can only originate from a simpagation rule. Since the active constraint is kept, one has to continue looking for applicable rules, even after the rule applied. 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 in place of the removed one. Therefore, we can deterministically iterate over the constraints that are candidates for matching the corresponding head from `Remove`, while the remaining partners can be found via nondeterministic enumeration as before. At the end of the iteration, we have to continue with the remaining rules for the active constraint.

Example 3.4 (Primes, contd.) *For space reasons, we just present a simple instance of the template, originating from the first occurrence of prime/1 in rule 3 (for readability with the constraint predicate already flattened, as described in Section 3.3):*

```
% rule prime(I) \ prime(J) <=> J mod I =:= 0 | true.
prime(A, B) :-
        get_constr_via([], C),              % get constraints from store
        init_iteration(C, prime/1, D),      % get partner candidates
        !,
        prime(D, B, A).                     % try to apply the rule

prime(A, B, C) :-
        iteration_last(A),                  % no more partner candidate
        prime_1(C, B).                      % try next rule head

prime(A, B, C) :-
        iteration_next(A, D, E),            % try next partner candidate
        (  get_args(D, [F]),
           match([C]-[G], [C]-[F]),
           check_guard([C,G], (G mod C =:= 0))
        ->                                  % rule applies
           remove_constraints([D]),         % remove the partner from store
        ;
           true                             % rule did not apply
        ),                                  % in any case, try same rule
        prime(E, B, C).                     % with another partner candidate

prime_1(C, B) :- ...                        % code to try next rule head
```

One instance (for lists) of the generic predicates steering the iteration is:

```
        iteration_last([]).
        iteration_next([D|E], D, E).
```

Case 3. Active constraint from `Keep` list, `Remove` list empty

This case originates from propagation rules. Since no constraint will be removed, all possible combinations of matching constraints have to be tried. The rule under consideration may apply with each combination. Therefore, all the partners (not just one as in the previous case) have to be searched through nested deterministic iteration. No matter if and how often the rule was applicable, we have to continue with the remaining rules for the active constraint.

Example 3.5 *This propagation rule is part of an interval solver.* `X::Min:Max` *constrains* `X` *to be within given lower and upper bounds* `Min` *and* `Max`. `le` *means less-or-equal.*

```
  X le Y, X::MinX:MaxX, Y::MinY:MaxY ==> X::MinX:MaxY, Y::MinX:MaxY.
```

The propagation rule produces basically the following code for `X le Y`.

```
X le Y :- le_1(X, Y).

le_1(X, Y) :-                          % active constraint (X le Y)
        get_constr_via([X], CXs),       % get constraints on X
        init_iteration(CXs, ::/2, PCXs), % get partner candidates
        !,
        le_1_0(PCXs, X, Y).            % try to apply the rule
le_1(X, Y) :-                          % rule was not applicable at all
        le_2(X, Y).                    % continue with next rule

le_2(X, Y) :-                          % no next rule
        suspend(X le Y).               % done, suspend the constraint

le_1_0(PCXs, X, Y) :-                  % outer loop for X::MinX:MaxX
        iteration_last(PCXs),          % no more partner candidate
        le_2(X, Y).                    % continue with next rule
le_1_0(PCXs, X, Y) :-
        iteration_next(PCXs, CX, PCXs1), % try next partner candidate for X
        (   get_args(CX,...), match(...),% match arguments
            get_constr_via([Y], CYs),    % get constraints on Y for next head
            init_iteration(CYs, ::/2, PCYs)
        ->
              le_1_1(PCYs, PCXs1, X, Y)  % try to apply the rule
        ;
              le_1_0(PCXs1, X, Y)        % try next partner candidate for X
        ).

le_1_1(PCYs, PCXs, X, Y) :-            % inner loop for Y::MinY:MaxY
        iteration_last(PCYs),          % no more partner candidate for Y
        le_1_0(PCXs, X, Y).            % continue with outer loop for X
le_1_1(PCYs, PCXs, X, Y) :-
        iteration_next(PCYs, CY, PCYs1), % try next partner candidate for Y
        (   get_args(CY,...), match(...),% match arguments
        ->                             % rule applies finally
              X::MinX:MaxY, Y::MinX:MaxY,% rule body
              le_1_1(PCYs1, PCXs, X, Y)  % continue, find another Y partner
        ;                              % rule did not apply
              le_1_1(PCYs1, PCXs, X, Y)  % continue, find another Y partner
        ).
```

3.3 Partial Evaluation Phase

The translation granularity was chosen so that the generated code would roughly run as is, with little emphasis on efficiency coming from local optimizations and specializations. These are performed in the final, third phase of the compiler using a simple instance of partial evaluation (PE). It is performed by using macros as they are available in most Prolog systems, e.g. [CaWi95]. In contrast to approaches that address all aspects of a language in a partial evaluator such as [Sah91], our restricted form of PE can be realized with an efficiency that meets the requirements of a production compiler. The functionalities of the main compiler macros are as follows:

- The generic predicates steering the iteration over partner constraints are specialized with respect to a particular representation of these multi-sets.
- Recursions are unfolded at compile time when the argument they recurse over is sufficiently known (typically lists with a known length).
- Head matching is specialized into unification instructions guarded by `nonvar/1` tests (as in [UeCh85]).
- The intermediate code uses redundant function symbols for the convenience of the compiler writers, e.g. to keep object, compiler and runtime-system variables visually apart. These symbols also help in type-checking the compiler. Redundant function symbols are removed by flattening, in particular in the head to facilitate clause indexing. For example, `constraint(head(prime/1,3-2), args([A]))` will be transformed into `prime1_3_2(A)`.

Example 3.6 (Primes, contd.) *The macro expansion phase results in the following code for our example 3.2. The code for matching and guard checking has been in-lined. The resulting trivial matchings (line 7), guards (line 3) and bodies (line 5) have been removed by PE.*

```
% rule candidates(1) <=> true.
candidates(A) :-                % 1
        A==1,                   % 2
        !.                      % 4
% rule candidates(N) <=> N>1 | M is N-1, prime(N), candidates(M).
candidates(A) :-                % 6
        nonvar(A),              % 8
        A>1,                    % 8
        !,                      % 9
        B is A-1,               % 10
        prime(A),               % 11
        candidates(B).          % 12
candidates(A) :-                % 13
        suspend(candidates(A)).  % 14
```

4 The Runtime System

The compiler generates Prolog clauses. Thus e.g. memory management is already taken care of. There are however functionalities that are not provided directly by most Prolog implementations:

- We need means to suspend, wake and re-suspend constraint predicates.
- We need efficient access to suspended constraints in the store through different access paths.

4.1 Suspensions

Typically, the attributes of variables are goals that suspend on that variable. They are re-executed (woken) each time one of their variables is touched. Via the attributed variables interface as found in SICStus or ECL^{i}PS^e Prolog the behaviour of attributed variables under unification is specified with a user-defined predicate. In the CHR implementation, suspended goals are our means to store constraints.

In more detail, the components of the CHR suspension data structure are:

- Constraint goal

- State of constraint
- Unique identifier
- Propagation history
- Re-use counter

The state indicates if the constraint is active, matched, removed or passive. The unique identifier is used, together with the propagation history, to ensure termination for propagation rules. Each propagation rule fires at most once for each tuple formed by the set of matched head constraints. The re-use counter is incremented with every re-use of the suspension. It is used for profiling and some more subtle aspects of controlling rule termination outside the scope of this paper.

To reuse suspensions, we made the suspension itself an argument of the re-executed goal. Internally, each constraint has an additional argument. When first executed, the argument is a free variable. When the constraint suspends, this extra argument is bound to the suspension itself. When it runs again, the suspension mechanism now has a handle to the suspension and can update its state. Code for this mechanism was removed from the listed code samples in this paper to avoid clutter.

4.2 Access Paths and Indexing

When a CHR searches for a partner constraint, a variable common to two heads of a rule considerably restricts the number of candidate constraints to be checked, because both partners must be suspended on this variable. The variables shared between partner constraints *index* the constraint store. Like with traditional data bases, the index may speed up join computations. Thus we usually access the constraint store by looking at only those constraints (cf. `get_constr_via/2`). 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.

Since functor and arity of the partner constraints we are searching for are known, direct access to the set of constraints of given functor/arity is desirable. Earlier implementations performed this selection by linear search over a part of the suspended constraints. Access to data through a variable, and then functor/arity, is exactly the functionality provided efficiently by attributed variables. In our runtime system we map every functor/arity pair to a fixed attribute slot of a variable at compile time yielding *constant time* access to the constraints. Only the arguments need to be matched at runtime.

Example 4.1 (Graph, nonground) *We keep the rule from example 1.2 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 produces:

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 index mentioned. The difference in computation time is two orders of magnitude. The difference between the deterministic and nondeterministic versions is rather insignificant.

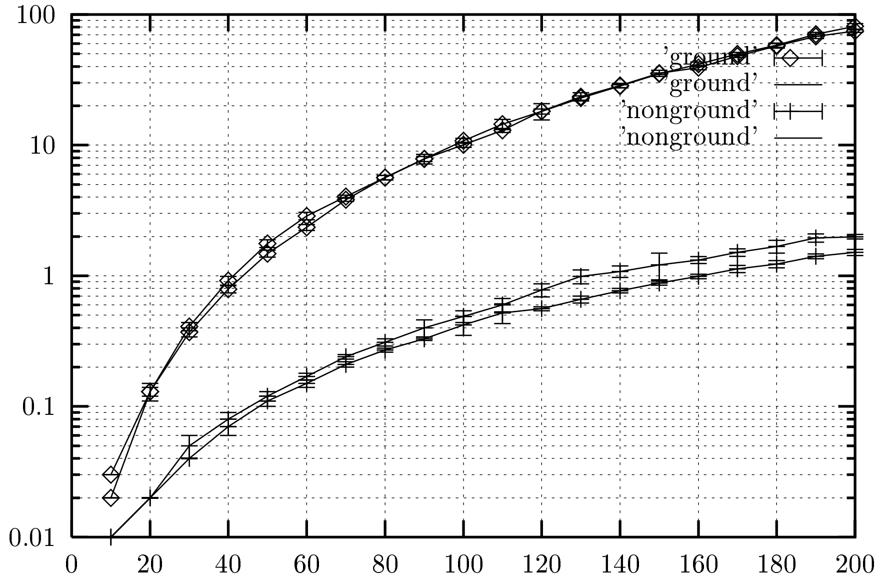

Figure 2: Join computation with and without indexing

5 Conclusions

The CHR system outlined in this paper was implemented in four man-months. The compiler is 1100 lines of Prolog, the runtime system around 600, which together is less than half of the ECL^{i}PS^e implementation. The new implementation removes some limitations of former implementations:

- The number of heads in a rule is no longer limited to two. The restriction was motivated originally by efficiency considerations since more heads need more search time. One can encode rules with more than two heads using additional auxiliary intermediate constraints. But then, the resulting rules are not only hard to understand, they are also less efficient than a true multi-headed implementation.
- Guards now support *Ask* and *Tell* [Sar93]. In this way, CHR can also be used as a general-purpose concurrent constraint language. (In this paper we only considered *Ask* parts of guards.)
- Attributed variables let us efficiently implement the generalized suspension mechanism needed for CHR at the source level. In particular, constant time access to constraints has been provided, instead of linear time in previous implementations.
- The CHR compiler has been "orthogonalized" by introducing three clearly defined compilation phases. Compilation is now on-the-fly, while loading. The template-based translation with subsequent macro-based partial evaluation allows for easy experimentation with different translation schemata.

- CHR specific demands, such as access paths using indexing and suspension recycling, are taken care of explicitly through customized versions of the suspension mechanism.
- Due to space limitations we also have not discussed *options* and *pragmas* in this paper - these are annotations to programs, rules or constraints that enable the compiler to perform powerful optimizations, that can sometimes make programs terminate or reduce their complexity class. In addition, rules apply now in textual order, which gives the programmer more control.

Benchmarking is difficult, because the new implementation is in SICStusProlog, while the previous one was in ECL^{i}PS^{e}Prolog. Attributed variables are implemented differently in these Prologs. Our measurements indicate that the new compiler produces code that is roughly twice as fast. The speed ratio improves the more different constraints are present, due to improved data structures and access paths.

Among the plans for the future development of the CHR implementation is the specification of the constraint store as an abstract data type. The default implementation would be the one based on suspensions via attributed variables. In that way the user can exploit peculiarities of his/her application. If all the constraints are ground for example, they make no reference to the suspend/wake mechanism. In that case they are probably better kept in a relational data base, which quite likely provides indices to facilitate the join computations.

More information about CHR is available at the CHR homepage
`http://www.informatik.uni-muenchen.de/~fruehwir/chr-intro.html`

References

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

[BCL88] Banatre J.-P., Coutant A. and Le Metayer D., A Parallel Machine for Multiset Transformation and its Programming Style, Future Generation Computer Systems 4:133-144, 1988.

[BeOl92] F. Benhamou and W.J. Older, Bell Northern Research, Applying interval arithmetic to Integer and Boolean constraints, Technical Report, June 1992.

[Br*98] P. Brisset et al., ECL^{i}PS^e 4.0 User Manual, IC-Parc at Imperial College, London, July 1998.

[CaWi95] Carlsson M., Widen J, Sicstus Prolog Users Manual, Release 3#0, Swedish Institute of Computer Science, SICS/R-88/88007C, 1995.

[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.

[Di*88] M. Dincbas et al., The Constraint Logic Programming Language CHIP, Fifth Generation Computer Systems, Tokyo, Japan, December 1988.

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

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

[FrAb97] T. Fr"uhwirth und S. Abdennadher, Constraint-Programmierung, Lehrbuch, Springer Verlag, September 1997.

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

[FrBr95b] T. Frühwirth and P. Brisset, Chapter on Constraint Handling Rules, in ECL^{i}PS^e 3.5.1 Extensions User Manual, ECRC Munich, Germany, December 1995.

[FAM99] T. Frühwirth, S. Abdennadher and H. Meuss, Confluence and Semantics of Constraint Simplification Rules, in E. Freuder (Ed.), Special Issue on the Second International Conference on Principles and Practice of Constraint Programming, Constraint Journal, Kluwer Academic Publishers, Vol 4(2), Mai 1999.

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

[Her93] B. Herbig, Eine homogene Implementierungsebene für einen hybriden Wissensrepräsentationsformalismus, Master Thesis, in German, University of Kaiserslautern, Germany, April 1993.

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

[HoFr98b] C. Holzbaur and T. Frühwirth, Join Evaluation Schemata for Constraint Handling Rules, 13th Workshop Logische Programmierung WLP'98, TU Vienna, Austria, September 1998.

[HoFr99a] C. Holzbaur and T. Frühwirth, Compiling Constraint Handling Rules into Prolog with Attributed Variables, International Conference on Principles and Practice of Declarative Programming (PPDP'99), Paris, France, September/October 1999.

[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, pp. 136-150, 1990.

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

[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, 1987.

[MaSt98] K. Marriott and P. J. Stuckey, Programming with Constraints, MIT Press, USA, March 1998.

[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, pp. 720-722, September 1985.

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

[Sah91] Sahlin D, An Automatic Partial Evaluator for Full Prolog, Swedish Institute of Computer Science, 1991.

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

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

[She98] K. Shen, The Extended CHR Implementation, chapter in ECL^{i}PS^e 4.0 Library Manual, IC-Parc at Imperial College, London, July 1998.

[Sid93] G.A. Sidebottom, A Language for Optimizing Constraint Propagation, Simon Fraser University, Canada, 1993.

[SmTr94] G. Smolka and R. Treinen (Eds.), DFKI Oz Documentation Series, DFKI, Saarbrücken, Germany, 1994.

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

[vHSD92] P. van Hentenryck, H. Simonis and M. Dincbas, Constraint Satisfaction Using Constraint Logic Programming, Artificial Intelligence, Vol 58(1-3):113–159, December 1992.