Optimizing Compilation of CHR with Rule Priorities

Leslie De Koninck[1⋆], Peter J. Stuckey[2], and Gregory J. Duck[2]

[1] Department of Computer Science, K.U.Leuven, Belgium
Leslie.DeKoninck@cs.kuleuven.be
[2] NICTA Victoria Laboratory
University of Melbourne, 3010, Australia
{pjs,gjd}@cs.mu.oz.au

Abstract Constraint Handling Rules were recently extended with user-definable rule priorities. This paper shows how this extended language can be efficiently compiled into the underlying host language. It extends previous work by supporting rules with dynamic priorities and by introducing various optimizations. The effects of the optimizations are empirically evaluated and the new compiler is compared with the state-of-the-art K.U.Leuven CHR system.

1 Introduction

Constraint Handling Rules (CHR) [7] is a rule based language, originally designed for the implementation of constraint programming systems, but also increasingly used as a general purpose programming language [11,15]. CHR is very flexible with respect to the specification of program logic, but it lacks high-level facilities for execution control. In particular, the control flow is most often fixed by the call-stack based refined operational semantics of CHR [5]. In [2], CHR is extended with user-definable rule priorities. This extended language, called CHR$^{\text{rp}}$, supports more high-level and flexible execution control than previously available while retaining the expressive power needed for the implementation of general purpose algorithms. An example of CHR with rule priorities is:

Example 1 (Less-or-Equal). The less-or-equal (`leq`) program is a classic CHR example. It implements a less-than-or-equal constraint by eventually translating it into equality constraints. Below is a CHR$^{\text{rp}}$ version of the `leq` program.

```
1 :: reflexivity  @ leq(X,X) <=> true.
1 :: antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
1 :: idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
2 :: transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

⋆ Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

The first rule has priority 1 (before `::`) and name `reflexivity` (before `@`). It is a simplification rule that states that any constraint of the form `leq`(a,a) should be "simplified" to (i.e. replaced by) *true*. The second rule `antisymmetry` states that two constraints `leq`(a,b) and `leq`(b,a) should be replaced with $a = b$, constraining the arguments to be equal. The third rule is a simpagation rule that says that given two constraints of the form `leq`(a,b) we should replace the second one (after the \) by *true*. The fourth rule is a propagation rule, which says given constraints `leq`(a,b) and `leq`(b,c) we should add a new constraint `leq`(a,c) without deleting anything. We have given the transitivity rule a lower priority (2), because we should only apply it if other rules do not apply. □

Dynamic rule priorities allow the priority of a rule to depend on the variables occurring on the left hand side of the rule.

Example 2 (Dijkstra's Shortest Path). Dijkstra's single source shortest path algorithm can be implemented in CHR^{rp} as follows:

```
  1 :: source(V) ==> dist(V,0).
  1 :: dist(V,D1) \ dist(V,D2) <=> D1 =< D2 | true.
D+2 :: dist(V,D), edge(V,C,U) ==> dist(U,D+C).
```

The input consists of a set of directed weighted edges, represented as `edge`/3 constraints where the first and last arguments respectively denote the begin and end nodes, and the middle argument represents the weight. The source node is given by the `source`/1 constraint. The first rule initiates the algorithm by creating the distance to the source node. The second rule introduces a guard `D1 =< D2` which must hold before the rule can fire. It ensures only the shortest path to node V is kept in the store. The last rule has a dynamic priority that orders the updates of distances as required by Dijkstra's algorithm. □

In the paper defining CHR^{rp} [2], its theoretical operational semantics as well as an implementation based on a source-to-source transformation were presented. In this paper, we show how CHR^{rp} programs can be efficiently compiled into the host language. We present the compilation process based on a refined version of the CHR^{rp} operational semantics, which is similar in concept to the refined operational semantics of CHR [5]. This semantics requires that every active constraint determines the priorities of all the rules in which it may participate. The compilation of rules with a dynamic priority is therefore handled by first applying a pseudo code source-to-source transformation which ensures that this condition holds. Next, it is shown how the generated code can be made more efficient by introducing optimizations that prevent unnecessary indexing and operations on the schedule. These optimizations are evaluated on benchmarks and the optimized system's performance is compared (with respect to equivalent programs in regular CHR) with the state-of-the-art K.U.Leuven CHR system [13] as well as with the result of the source-to-source transformation presented in [2].

This paper presents the *first* implementation of CHR^{rp} with *dynamic* priorities. It is about an order of magnitude faster than the one of [2] which is limited to programs with static priorities only, and is already almost as fast as the highly

optimized K.U.Leuven CHR system while offering a much more high level form of execution control.[1] The rest of this paper is organized as follows. Section 2 reviews the syntax and semantics of CHR^{rp}. A basic compilation schema is presented in Section 3 and optimizations for this schema are given in Section 4. The resulting system is evaluated in Section 5. We conclude in Section 6.

2 Preliminaries

This section reviews the syntax and semantics of Constraint Handling Rules with Rule Priorities (CHR^{rp}). For a more thorough introduction into CHR, see [7] or [12]. See [2] for more information about CHR^{rp}.

Syntax A constraint $c(t_1, \ldots, t_n)$ is an atom of predicate c/n with t_i a host language value (e.g., a Herbrand term in Prolog) for $1 \leq i \leq n$. There are two types of constraints: built-in constraints and CHR constraints (also called user-defined constraints). The CHR constraints are solved by the CHR program whereas the built-in constraints are solved by an underlying constraint solver (e.g., the Prolog unification algorithm).

There are three types of Constraint Handling Rules: *simplification rules*, *propagation rules* and *simpagation rules*. They have the following form:

$$\begin{array}{lllll} \textbf{Simplification} & p :: r \ @ & & H^r & \iff g \mid B \\ \textbf{Propagation} & p :: r \ @ \ H^k & & & \implies g \mid B \\ \textbf{Simpagation} & p :: r \ @ \ H^k \setminus H^r & & & \iff g \mid B \end{array}$$

where p is the rule priority, r is the rule *name*, H^k and H^r are non-empty sequences of CHR constraints and are called the *heads* of the rule. The rule *guard* g is a sequence of built-in constraints and the rule *body* B is a sequence of both CHR and built-in constraints. The rule priority is either a number in which case the rule is called a *static* priority rule, or an arithmetic expression whose variables appear in the heads H^k and/or H^r in which case the rule is called a *dynamic* priority rule. We say that priority p is higher than priority p' if $p < p'$. For simplicity, we sometimes assume priorities are integers and the highest priority is 1. Finally, a program P is a set of CHR rules.

Operational Semantics Operationally, CHR constraints have a multi-set semantics. To distinguish between different occurrences of syntactically equal constraints, CHR constraints are extended with a unique identifier. An identified CHR constraint is denoted by $c\#i$ with c a CHR constraint and i the identifier. We write $\mathsf{chr}(c\#i) = c$ and $\mathsf{id}(c\#i) = i$. We extend these to map sequences in the obvious manner. We use $+\!+$ for sequence concatenation.

The operational semantics of CHR^{rp}, called the priority semantics and denoted by ω_p, is given in [2] as a state transition system, similar to the approach

[1] When benchmarked on operationally equivalent programs.

of [5] for the theoretical and refined operational semantics of CHR. A CHR execution state σ is represented as a tuple $\langle G, S, B, T\rangle_n$ where G is the goal, a multi-set of constraints to be solved; S is the CHR constraint store, a set of identified CHR constraints; B is the built-in store, a conjunction of built-in constraints; T is the propagation history, a set of tuples denoting the rule instances that have already fired; and n is the next free identifier, used to identify new CHR constraints. The transitions of ω_p are shown in Table 1. They are exhaustively applied starting from the state $\langle G, \emptyset, true, \emptyset\rangle_1$ with G the initial goal.

Example 3. And example derivation for the `leq` program given in Example 1 and initial goal $G = \{leq(A,B), leq(B,C), leq(B,A)\}$ is shown below:

$$\langle\{leq(A,B), leq(B,C), leq(B,A)\}, \emptyset, true, \emptyset\rangle_1$$

Introduce $\overset{\omega_p}{\mapsto}_P \langle\{leq(B,C), leq(B,A)\}, \{leq(A,B)\#1\}, true, \emptyset\rangle_2$

Introduce $\overset{\omega_p}{\mapsto}_P \langle\{leq(B,A)\}, \{leq(A,B)\#1, leq(B,C)\#2\}, true, \emptyset\rangle_3$

Introduce $\overset{\omega_p}{\mapsto}_P \langle\emptyset, \{leq(A,B)\#1, leq(B,C)\#2, leq(B,A)\#3\}, true, \emptyset\rangle_4$

Apply `antisymmetry` $\theta = \{X/A, Y/B\} \overset{\omega_p}{\mapsto}_P \langle\{A = B\}, \{leq(B,C)\#2\}, true, \emptyset\rangle_4$

Solve $\overset{\omega_p}{\mapsto}_P \langle\emptyset, \{leq(B,C)\#2\}, A = B, \emptyset\rangle_4$

For termination, the antisymmetry rule must fire before the transitivity rule. □

3 Basic Compilation Schema

This section gives an overview of the basic compilation schema for CHR$^{\text{rp}}$ programs. First, in Section 3.1, we present a refinement of the ω_p semantics that follows the actual implementation more closely. This refinement, called the refined priority semantics and denoted by ω_{rp}, is based on the refined operational semantics ω_r of (regular) CHR and is thus also based on lazy matching and the concept of active constraints. The ω_{rp} semantics requires that each active constraint determines the actual (ground) priorities of all rules in which they may participate. In Section 3.2, we show how dynamic priority rules can be transformed so that this property holds for all active constraints. Finally, Section 3.3 gives an abstract version of the code generated for each of the ω_{rp} transitions.

3.1 The Refined Priority Semantics ω_{rp}

The refined priority semantics ω_{rp} is given as a state transition system. Its states are represented by tuples of the form $\langle A, Q, S, B, T\rangle_n$, where S, B, T and n are as in the ω_p semantics, A is a sequence of constraints, called the activation stack, and Q is a priority queue. In the ω_{rp} semantics, constraints are scheduled for activation at a given priority. By $c\#i : j \ @ \ p$ we denote the identified constraint $c\#i$ being tried at its j^{th} occurrence of fixed priority p. In what follows, the priority queue is considered a set supporting the operation find_min which returns one of its highest priority elements.

The transitions of the ω_{rp} semantics are shown in Table 2. The main differences compared to the ω_r semantics are the following. Instead of adding new or

1. Solve $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, S, c \wedge B, T\rangle_n$ where c is a built-in constraint.
2. Introduce $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}$ where c is a CHR constraint.
3. Apply $\langle \emptyset, H_1 \cup H_2 \cup S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle C, H_1 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n$ where P contains a rule of priority p of the form $$p :: r \ @ \ H_1' \backslash H_2' \iff g \mid C$$ and a matching substitution θ such that $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, $\theta(p)$ is a ground arithmetic expression and $t = \mathsf{id}(H_1) +\!\!+ \mathsf{id}(H_2) +\!\!+ [r] \notin T$. Furthermore, no rule of priority p' and substitution θ' exists with $\theta'(p') < \theta(p)$ for which the above conditions hold.

Table1. Transitions of ω_p

1. Solve $\langle [c\|A], Q, S_0 \cup S_1, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle A, Q', S_0 \cup S_1, c \wedge B, T\rangle_n$ where c is a built-in constraint, $vars(S_0) \subseteq fixed(B)$ is the set of variables fixed by B, and $Q' = Q \cup \{c\#i \ @ \ p \mid c\#i \in S_1 \wedge c$ has an occurrence in a priority p rule$\}$. This reschedules constraints whose matches might be affected by c.
2. Schedule $\langle [c\|A], Q, S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle A, Q', \{c\#n\} \cup S, B, T\rangle_{n+1}$ with c a CHR constraint and $Q' = Q \cup \{c\#n \ @ \ p \mid c$ has an occurrence in a priority p rule$\}$.
3. Activate $\langle A, Q, S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle [c\#i : 1 \ @ \ p\|A], Q \setminus \{c\#i \ @ \ p\}, S, B, T\rangle_n$ where $c\#i \ @ \ p = \mathsf{find\_min}(Q)$, and $A = [c'\#i' : j' \ @ \ p'\|A']$ with $p < p'$ or $A = \epsilon$.
4. Drop $\langle [c\#i : j \ @ \ p\|A], Q, S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle A, Q, S, B, T\rangle_n$ if there is no j^{th} priority p occurrence of c in P.
5. Simplify $\langle [c\#i : j \ @ \ p\|A], Q, \{c\#i\} \cup H_1 \cup H_2 \cup H_3 \cup S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle C +\!\!+ A, Q, H_1 \cup S, \theta \wedge B, T\rangle_n$ where the j^{th} priority p occurrence of c is d_j in rule $$p' :: r \ @ \ H_1' \backslash H_2', d_j, H_3' \iff g \mid C$$ and there exists a matching substitution θ such that $c = \theta(d_j)$, $p = \theta(p')$, $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $\mathsf{chr}(H_3) = \theta(H_3')$ and $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$. This transition only applies if the **Activate** transition does not.
6. Propagate $\langle [c\#i : j \ @ \ p\|A], Q, \{c\#i\} \cup H_1 \cup H_2 \cup H_3 \cup S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle C +\!\!+ [c\#i : j \ @ \ p \mid A], Q, H_1 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n$ where the j^{th} priority p occurrence of c is d_j in $$p' :: r \ @ \ H_1', d_j, H_2' \backslash H_3' \iff g \mid C$$ and there exists a matching substitution θ such that $c = \theta(d_j)$, $p = \theta(p')$, $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $\mathsf{chr}(H_3) = \theta(H_3')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, and $t = \mathsf{id}(H_1) +\!\!+ [i] +\!\!+ \mathsf{id}(H_2) +\!\!+ [r] \notin T$. This transition only applies if the **Activate** transition does not.
7. Default $\langle [c\#i : j \ @ \ p\|A], Q, S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle [c\#i : j+1 \ @ \ p\|A], Q, S, B, T\rangle_n$ if the current state cannot fire any other transition.

Table2. Transitions of ω_{rp}

reactivated constraints to the activation stack, the **Solve** and **Schedule**[2] transitions schedule them for activation, once for each priority at which they have occurrences. The **Activate** transition activates the highest priority scheduled constraint if it has a higher priority than the current active constraint (if any). This transition only applies if the **Solve** and **Schedule** transitions are not applicable, i.e., after processing the initial goal or a rule body. Noteworthy is that once a constraint is active at a given priority, it remains so at least until a rule fires or it is made passive by the **Drop** transition. Hence we should only check the priority queue for a higher priority scheduled constraint at these program points. Again, the transitions are exhaustively applied starting from an initial state $\langle G, \emptyset, \emptyset, true, \emptyset \rangle_1$ with G the goal, given as a sequence.

Example 4. The ω_{rp} state corresponding to the ω_p state after the 3 **Introduce** transitions in Example 3 is:[3]

$$\langle [], \{leq(A,B)\#1@\{1,2\}, leq(B,C)\#2@\{1,2\}, leq(B,A)\#3@\{1,2\}\}, \\ \{leq(A,B)\#1, leq(B,C)\#2, leq(B,A)\#3\}, true, \emptyset \rangle_4$$

If $leq(B,C)\#2@1$ is activated first then it finds no matching partners and is eventually dropped. If $leq(A,B)\#1@1$ is activated next, then we have

$$\begin{array}{l} \langle [leq(A,B)\#1:1@1], \{leq(A,B)\#1@2, leq(B,C)\#2@2, leq(B,A)\#3@\{1,2\}\}, \\ \qquad \{leq(A,B)\#1, leq(B,C)\#2, leq(B,A)\#3\}, true, \emptyset \rangle_4 \overset{\omega_{rp}}{\mapsto}_P \textbf{(Default)} \\ \langle [leq(A,B)\#1:2@1], \{leq(A,B)\#1@2, leq(B,C)\#2@2, leq(B,A)\#3@\{1,2\}\}, \\ \qquad \{leq(A,B)\#1, leq(B,C)\#2, leq(B,A)\#3\}, true, \emptyset \rangle_4 \overset{\omega_{rp}}{\mapsto}_P \textbf{(Simplify)} \\ \langle [A=B], \{leq(A,B)\#1@2, leq(B,C)\#2@2, leq(B,A)\#3@\{1,2\}\}, \\ \qquad \{leq(B,C)\#2\}, true, \emptyset \rangle_4 \overset{\omega_{rp}}{\mapsto}_P \textbf{(Solve)} \\ \langle [], \{leq(A,B)\#1@2, leq(B,C)\#2@\{1,2\}, leq(B,A)\#3@\{1,2\}\}, \\ \qquad \{leq(B,C)\#2\}, A=B, \emptyset \rangle_4 \end{array}$$

This last transition reschedules the $leq(B,C)\#2$ constraint at priorities 1 and 2. None of the remaining constraints in the schedule lead to a rule firing. □

3.2 Transforming Dynamic Priority Rules

In the description of the ω_{rp} semantics, we have assumed that every constraint knows the priorities of all rules in which it may participate. For rules with a dynamic priority, this is obviously not always the case.

Example 5. Consider the rule

```
X+Y :: r @ a(X,Z) \ b(Y,Z), c(X,Y) <=> d(X).
```

[2] The **Schedule** transition corresponds to the **Activate** transition in ω_r.

[3] We use the notation $c\#i \ @ \ \{p_1, \ldots, p_n\}$ to denote $\{c\#i \ @ \ p_1, \ldots, c\#i \ @ \ p_n\}$.

In this rule the `c`/2 constraint with ground arguments X and Y knows the priority of the rule, but neither the `a`/2 nor the `b`/2 constraints do. Given the `a`/2 constraint, we need to combine (join) it with either the `b`/2 or `c`/2 constraint to determine the actual priority. □

In this section, we present a pseudo code source-to-source transformation to transform a program such that this property is satisfied. In what follows, we refer to the *join order* for a given constraint occurrence, which is the order in which the partner constraints for this occurrence are retrieved (by nested loops). We consider a join order Θ to be a permutation of $\{1, \ldots, n\}$ where n is the number of heads of the rule. Now, consider a dynamic priority rule

$$p :: r \ @ \ C_1, \ldots, C_i \backslash C_{i+1}, \ldots, C_n \iff g \mid B$$

an active head C_j, a join order Θ with $\Theta(1) = j$ and a number k, $1 \leq k \leq n$ such that the first k heads, starting with C_j and following join order Θ, determine the rule priority. We rewrite rule r as follows (for every j, $1 \leq j \leq n$):

$$\begin{aligned}
&1 :: r_j \ @ \ C_{\Theta(1)} \# Id_1, \ldots, C_{\Theta(k)} \# Id_k \implies \\
&\qquad r\text{-match}_j(Id_1, \ldots, Id_k, \mathit{Vars}) \text{ pragma passive}(Id_2), \ldots, \text{passive}(Id_k) \\
&1 :: r'_j \ @ \ r\text{-match}_j(Id_1, \ldots, Id_k, \mathit{Vars}) \iff \\
&\qquad\qquad \text{ground}(p) \mid r\text{-match}'_j(Id_1, \ldots, Id_k, \mathit{Vars}) \\
&p :: r''_j \ @ \ r\text{-match}'_j(Id_1, \ldots, Id_k, \mathit{Vars}), C_{\Theta(k+1)} \# Id_{k+1}, \ldots, C_{\Theta(n)} \# Id_n \implies \\
&\qquad \text{alive}(Id_1), \ldots, \text{alive}(Id_k), g \mid \text{kill}(Id_{\Theta^{-1}(i+1)}), \ldots, \text{kill}(Id_{\Theta^{-1}(n)}), B \\
&\qquad \text{pragma passive}(Id_{k+1}), \ldots, \text{passive}(Id_n), \\
&\qquad\qquad\qquad\qquad \text{history}([Id_{\Theta^{-1}(1)}, \ldots, Id_{\Theta^{-1}(n)}, r])
\end{aligned}$$

where *Vars* are the variables shared by the first k heads on the one hand, and the remaining heads, the guard, the body and the priority expression on the other, i.e., $\mathit{Vars} = \big(\cup_{i=1}^{k} vars(C_{\Theta(i)})\big) \cap \big(\big(\cup_{i=k+1}^{n} vars(C_{\Theta(i)})\big) \cup vars(g \wedge B \wedge p)\big)$. The first rule generates a partial match that knows its priority once the necessary arguments are ground (fixed). It runs at the highest possible value of the dynamic priority expression.[4] The second rule ensures that the priority expression is ground before the partial match is scheduled at its dynamic priority. The rule runs at the same priority as the first one. Finally, the third rule extends the partial match (with ground priority) into a full match. There we check whether all constraints in the partial match are still alive (calls to `alive`/1), and delete the removed heads (calls to `kill`/1). The *pragma*[5] `passive`/1 denotes that a given head is passive, i.e., no occurrence code is generated for it (see further in Section 3.3). The pragma `history`/1 states the tuple layout for the propagation history. All rule copies share the same history which ensures that each instance of the original rule can fire only once.

Example 6. Given the rule `r` of Example 5 and join orders $\Theta_1 = [1, 2, 3]$, $\Theta_2 = [2, 3, 1]$ and $\Theta_3 = [3, 2, 1]$.[6] Furthermore assuming we schedule at a dynamic priority as soon as we know it, we generate the following rules:

[4] We assume 1 is an upperbound. A tighter one can be used instead if such is known.
[5] Most CHR systems support compiler directives by using the keyword `pragma`.
[6] By slight abuse of syntax, we denote $\Theta(1) = \theta_1, \ldots, \Theta(n) = \theta_n$ by $\Theta = [\theta_1, \ldots, \theta_n]$.

```
  1 :: r1 @ a(X,Z) #Id1, b(Y,Z) #Id2 ==>
       r-match1(Id1,Id2,X,Y) pragma passive(Id2).
  1 :: r1' @ r-match1(Id1,Id2,X,Y) <=>
       ground(X+Y) | r-match1'(Id1,Id2,X,Y).
X+Y :: r1'' @ r-match1'(Id1,Id2,X,Y), c(X,Y) #Id3 ==>
       alive(Id1), alive(Id2) | kill(Id2), kill(Id3), d(X)
       pragma passive(Id3), history([Id1,Id2,Id3],r).

  1 :: r2 @ b(Y,Z) #Id1, c(X,Y) #Id2 ==>
       r-match2(Id1,Id2,X,Y,Z) pragma passive(Id2).
  1 :: r2' @ r-match2(Id1,Id2,X,Y,Z) <=>
       ground(X+Y) | r-match2'(Id1,Id2,X,Y,Z).
X+Y :: r2'' @ r-match2'(Id1,Id2,X,Y,Z), a(X,Z) #Id3 ==>
       alive(Id1), alive(Id2) | kill(Id1), kill(Id2), d(X)
       pragma passive(Id3), history([Id3,Id1,Id2,r]).

  1 :: r3 @ c(X,Y) #Id1 ==> r-match3(Id1,X,Y).
  1 :: r3' @ r-match3(Id1,X,Y) <=> ground(X+Y) | r-match3'(Id1,X,Y).
X+Y :: r3'' @ r-match3'(Id1,X,Y), b(Y,Z) #Id2, a(X,Z) #Id3 ==> alive(Id1) |
       kill(Id1), kill(Id2), d(X) pragma history([Id3,Id2,Id1,r]).
```

Note that since this is a simpagation rule, a propagation history is not necessary. We only show it for illustrative purposes. □

The proposed translation schema implements a form of eager matching: all r-match$_j$ constraints are generated eagerly at the highest priority before one is fired. This approach resembles the TREAT matching algorithm [10]. Also similar to the TREAT algorithm and unlike the RETE algorithm [6], we allow different join orders for each active head.

3.3 Compilation

Now that we have shown how a program can be transformed such that each constraint knows the priorities of all rules in which it may participate, we are ready to present the compilation schema. The generated code follows the ω_{rp} semantics closely. In what follows, we assume the host language is Prolog, although the compilation process easily translates to other host languages as well. We note that the generated code presented in this section, much resembles that of regular CHR under the refined operational semantics, as described in for example [12]. The differences correspond to those between ω_r and ω_{rp} as given in Section 3.1.

CHR Constraints Whenever a new CHR constraint is asserted, it is scheduled at all priorities at which it may fire (**Schedule** transition). Furthermore, it is attached to its variables for the purpose of facilitating the **Solve** transition. In Prolog this is done using attributed variables. The idea is similar to that of subscribing to event notifiers. Finally, the constraint is inserted into all indexes on its arguments. Schematically, the generated code looks as follows:

```
c(X1,...,Xn) :- GenerateSuspension, S = Suspension,
                schedule(p1,c/n_prio_p1_occ_1_1(S)),
                ...
                schedule(pm,c/n_prio_pm_occ_1_1(S)),
                AttachToVariables, InsertIntoIndexes.
```

The *GenerateSuspension* code creates a data structure (called the *suspension term* in CHR terminology) for representing the constraint in the constraint store. It has amongst others fields for the constraint identifier, its state (dead or alive), its propagation history,[7] its arguments, and pointers for index management. The scheduling code consists of insertions of calls to the code for the first occurrence of each priority p_i ($1 \leq i \leq m$) into the priority queue. With respect to the usual code for CHR constraints under the ω_r semantics, we have added the `schedule/2` calls and removed the call to the code of the first occurrence of the constraint.

Built-in Constraints Built-in constraints are dealt with by the underlying constraint solver, in this case the Prolog Herbrand solver. Whenever this solver binds a variable to another variable or a term (during unification), a so-called *unification hook* is called. In this hook, the CHR part of the **Solve** transition is implemented. It consists of reattaching the affected constraints, updating the indexes, and scheduling the affected constraints again at each priority for which they have occurrences.

Occurrence Code For each constraint occurrence, a separate predicate is generated, implementing the **Simplify** and **Propagate** transitions. Its clauses are shown below. The approach is very similar to how occurrences are compiled under the refined operational semantics of CHR. The differences are that only the occurrences of the same priority are linked, where occurrences with a dynamic priority are assumed to run at different priorities, and the priority queue is checked (`check_activation/1`) after each rule firing. The code below is for the j^{th} priority p occurrence of the c/n constraint which is in an m-headed rule. The indices $r(1), \ldots, r(i)$ refer to the removed heads.

```
c/n_prio_p_occ_j_1(S1) :-
        (   alive(S1), HeadMatch, LookupNext(S̄2)
        ->  c/n_prio_p_occ_j_2(S̄2,S1).
        ;   c/n_prio_p_occ_j+1_1(S1)
        ).

c/n_prio_p_occ_j_2([S2|S̄2],S1) :-
        (   alive(S2), S2 \= S1, HeadMatch, LookupNext(S̄3)
        ->  c/n_prio_p_occ_j_3(S̄3,S2,S̄2,S1)
        ;   c/n_prio_p_occ_j_2(S̄2,S1)
        ).
```

[7] We use a distributed propagation history, like in the K.U.Leuven CHR system [12].

```
c/n_prio_p_occ_j_2([],S1) :- c/n_prio_p_occ_j + 1_1(S1).

...

c/n_prio_p_occ_j_m([Sm|Sm],Sm−1,...,S1) :-
        (   alive(Sm), Sm \= S1, ..., Sm \= Sm−1,
            HeadMatch, RemainingGuard, HistoryCheck
        ->  AddToHistory, kill(Sr(1)), ..., kill(Sr(i)),
            Body, check_activation(p),
            (   alive(S1)
            ->  (   ...
                ... (   alive(Sm−1)
                    ->  c/n_prio_p_occ_j_m(Sm,...,S1)
                    ;   c/n_prio_p_occ_j_m − 1(Sm−1,...,S1)
                    )
                ...
            ;   true
            )
        ;   c/n_prio_p_occ_j_m(Sm,Sm−1,...,S1)
        ).
c/n_prio_p_occ_j_m([],_,Sm−1,...,S1) :-
        c/n_prio_p_occ_j_m − 1(Sm−1,...,S1).
```

The *HeadMatch* call checks whether the newly looked up head matches with the rule and with the previous heads. A list of all candidates for the next head is returned by *LookupNext*/1. *RemainingGuard* is the part of the guard that has not already been tested by the *HeadMatch* calls. Propagation history checking and extending is handled by respectively *HistoryCheck* and *AddToHistory*. After having gone through all rule instances for the given occurrence, the next occurrence is tried (**Default**) or the activation call returns (**Drop**). The `check_activation`/1 call in the occurrence code checks whether a constraint occurrence is scheduled at a higher priority than the current one. It implements the **Activate** transition.

4 Optimization

We now present the main optimizations implemented in the CHR$^{\text{rp}}$ compiler. We start with optimizations that reduce the number of priority queue operations.

4.1 Reducing Priority Queue Operations

A first optimization consists of only scheduling the highest priority occurrence of every new constraint. Only when the constraint has been activated at this priority and has gone through all of its occurrences without being deleted, it is scheduled for the next priority. This is a simple extension of the continuation based approach we already applied for constraint occurrences at equal priority.

In the basic compilation scheme, it is checked whether a higher priority scheduled constraint exists after each rule firing. In a number of cases, this is not

needed. If the active constraint is removed, it is popped from the top of the activation stack and the activation check that caused it to be activated in the first place, checks again to see if other constraints are ready for activation. So, since a priority queue check will take place anyway, there is no need to do this twice. If the body of a rule does not contain CHR constraints with a priority higher than the current one, nor built-in constraints that can trigger any CHR constraints to be scheduled at a higher priority, then after processing the rule body, the active constraint remains active and we do not need to check the priority queue. We denote the above optimizations by *reduced activation checking*.

Building further on this idea, we note that by analyzing the body, we can sometimes determine which constraint will be activated next. Instead of scheduling it first and then checking the priority queue, we can activate it directly at its highest priority. We call this *inline activation*. Inline activation is not limited to one constraint: we can directly activate all constraints that have the same highest priority. Indeed, when the first of these constraints returns from activation, the priority queue cannot contain any constraint scheduled at a higher priority, because such a constraint would have been activated before returning.

Example 7. We illustrate the applicability of the proposed optimizations on the `leq` program given in Example 1. The `leq`/2 constraint has 5 occurrences at priority 1 and 2 at priority 2. New `leq`/2 constraints are only scheduled at priority 1. Only if an activated constraint has passed the 5^{th} priority 1 occurrence, it is scheduled at priority 2. For the first three priority 1 occurrences, as well as for the removed occurrence in the idempotence rule, the active constraint is removed and so there is no need to check the priority queue after firing the rule body. Since the body of the remaining priority 1 occurrence equals `true`, no higher priority constraint is scheduled and so we do not need to check the queue here either. Finally, for the transitivity rule we have that the only constraint in the body has a higher priority occurrence than the current active occurrence, and so we can apply inline activation there. □

4.2 Late Indexing

Similar to an optimization from regular CHR, we can often postpone storage of constraints, reducing cost if the constraint is removed before these operations are be applied. We extend the late storage concept of [9] to late indexing, where we split up the task of storing a constraint into the subtasks of inserting it into different indexes. The main idea is that an active constraint can only lose activation to another constraint in rules of a higher priority. This implies that when a constraint is active at a given current priority, it should only be stored in those indexes that are used by higher priority rules.

Example 8. In the `leq` program (Example 1), the `leq`/2 constraints are indexed

- on the combination of both arguments (`antisymmetry` and `idempotence`);
- on the first argument and on the second argument (`transitivity`);
- on the constraint symbol for the purpose of showing the constraint store.

By using late indexing, new `leq`/2 constraints are not indexed at the moment they are asserted, but only scheduled (only at priority 1). When an active `leq`/2 constraint 'survives' the 5^{th} priority 1 occurrence, it is indexed on the combination of both arguments and rescheduled at priority 2. We can postpone the indexing this long because only one constraint can be on the execution stack for each priority and hence all partner constraints have either been indexed already, or still need to be activated. Only after a reactivated `leq`/2 constraint has passed the second priority 2 occurrence, it is stored in the remaining indexes. Note that our approach potentially changes the execution order of the program, which can sometimes contribute to changes in the running time (in either direction). □

4.3 Passive Occurrences

In [4] we give a criterion to decide whether a given constraint occurrence can be made passive. Passive occurrences allow us to avoid the overhead of looking up partner constraints, and sometimes also the overhead related to scheduling and indexing. Due to space considerations, we do not go into detail here.

5 Evaluation

Less-or-Equal The `leq` benchmark uses the program of Example 1 and for given n, the initial goal $G = G_1 \cup G_2$ with

$$G_1 = \{\texttt{leq}(X_1,X_2), \ldots, \texttt{leq}(X_{n-1},\ X_n)\} \wedge G_2 = \{\texttt{leq}(X_n,X_1)\}$$

From the goal G, a final state is derived in which $X_1 = X_2 = \ldots = X_{n-1} = X_n$.

In [2], it was shown that the benchmark scales better using priorities and batch processing of the goal, because of the order in which constraints are activated (i.e., more recently added constraints are preferred). Using this order, the $\texttt{leq}(X_1,X_n)$ constraint that causes the loop to be detected, is asserted after a linear number of firings of the transitivity rule. Interestingly, by using the late indexing optimization, we get a higher complexity because the necessary partner constraints for the optimal firing order are not yet stored. However, when we first assert subgoal G_1, wait for a fixpoint, and then assert G_2, then both the versions with and without late indexing behave the same.

Optimizations The following table shows benchmark results for various programs where the effect of each of the optimizations is measured. The `loop` benchmark consists of the following two rules (and does not rely on priorities):

```
1 :: a(X) <=> X > 0 | a(X-1).        1 :: a(0) <=> true.
```

and initial goal $\{\texttt{a}(2^{20})\}$; the `leq` benchmark is the same as in the previous subsection, with $n = 80$; the `dijkstra` benchmark uses the program of Example 2 with a graph of 2^{15} nodes and $3 \cdot 2^{15}$ edges; the `union-find` benchmark is based

on the naive union-find program given in [15] and uses 2^{12} random `union`/2 constraints over an equal number of elements (see also [4]). Finally, the `sudoku` benchmark uses an adapted version of the Sudoku solver from the CHR website [14] (see also [2]) and solves a puzzle in which initially 16 cells have a value. The benchmarks are executed with the late indexing (LI), inline activation (IA) and reduced activation checking (RAC) optimizations switched on and off.

LI	IA	RAC	`loop`	`leq`	`dijkstra`	`union-find`	`sudoku`
			100%	100%	100%	100%	100%
		√	88%	99%	98%	93%	98%
	√		73%	97%	97%	82%	99%
√			46%	63%	97%	40%	112%
√	√	√	8%	56%	92%	17%	109%

The inline activation analysis assumes that dynamic priority rules run at the highest possible value of the priority expression. It currently assumes this value is 1, but a bounds analysis or a user declaration can give a tighter upperbound. In the `dijkstra` and `sudoku` benchmarks, we have used a tight upperbound of 2 for the dynamic priority rules. The passive analysis applied to the `union-find` benchmark cuts off another 2% and reduces the runtime with full optimization to about 15% of the runtime without optimization. The late indexing optimization can change the execution order. We have already shown how this affects the `leq` benchmark. Similarly, it also affects the `sudoku` benchmark which has (amongst others) 11% more rule firings, hence the increase in runtime. Moreover, late indexing only reduces the amount of index insertions with 3% in this benchmark.

We also compare CHR$^{\mathrm{rp}}$ against the K.U.Leuven CHR system under the ω_r semantics. For `leq`, `loop` and `union-find`, we execute the same code ignoring priorities (though sometimes relying on rule order). For `dijkstra` and `sudoku` the K.U.Leuven CHR code encodes equivalent behavior obtained using priorities by other methods. Hence the rules are more involved. The `leq` benchmark takes about 4% less time on our system, the `loop` benchmark takes about 5.3 times more, and the `union-find` benchmark takes 53% more time. Comparison for the `sudoku` benchmark is difficult because the search trees are different. In this particular case, K.U.Leuven CHR is about 10% faster than our CHR$^{\mathrm{rp}}$ system (without late indexing), but also has 5% less rule firings.

For the `dijkstra` benchmark, we compared with the CHR program given in [16].[8] Our implementation runs about 2.4 times slower than the (regular) CHR implementation, but it is also arguably more high level. Noteworthy is the following optimization, implemented in [16] and reformulated here in terms of our CHR$^{\mathrm{rp}}$ implementation. The rule

```
1 :: dist(V,D1) \ dist(V,D2) <=> D1 =< D2 | true.
```

removes the `dist(V,`$\mathtt{D}_2$`)` constraint which might still be scheduled at priority $\mathtt{D}_2 + 2$. After firing the rule, the `dist(V,`$\mathtt{D}_1$`)` constraint is scheduled at priority

[8] For a fair comparison, we use a combination of Fibonacci heaps for the dynamic priorities, and an array for static priorities 1 and 2, as priority queue.

$D_1 + 2$. Instead of first (lazily) deleting a scheduled item, and then inserting a new one, the cheaper decrease_key operation can be used instead (because $D_1 \leq D_2$). Compared to an altered version of the original CHR implementation in which this optimization is turned off, our code remains (only) 13% slower. The results are for the described problem instances and vary somewhat over different problem sizes. Nonetheless, the asymptotic time complexities are the same in both CHR and CHR^{rp} versions and so the results are sufficiently generalizable.

Finally, we compare to the source-to-source transformation given in [2]: the `leq` benchmark runs about 4.7 times faster on our system; the `loop` benchmark about 39 times, and the `union-find` benchmark about 19 times. The remaining benchmarks could not run because they contain rules with a dynamic priority which are not supported by the source-to-source transformation.

6 Concluding Remarks

This paper presents a compilation schema for CHR^{rp}: CHR with rule priorities. We have shown the feasibility of implementing rules with both static and dynamic rule priorities using a lazy matching approach, in contrast with the eager matching as implemented by the RETE algorithm and derivatives. We have proposed various ways to optimize the generated code and shown their effectiveness on benchmarks. Our benchmark results furthermore indicate that our implementation already comes close to the state-of-the-art K.U.Leuven CHR system (and sometimes even surpasses it), while offering a much more high level form of execution control. Compared to the implementation given in [2], our system is about an order of magnitude faster on the benchmarks. This work extends [2] by introducing the refined priority semantics, offering support for dynamic priority rules, presenting the first *compiler* for CHR^{rp}, and by proposing several optimizations for the generated code. The optimizations consist of both completely new optimizations (those related to reducing priority queue operations), as well as refinements of previously known optimizations for (regular) CHR (i.e., late indexing and the passive analysis).

Related Work Rule priorities (sometimes called *salience*) are found in many rule based languages, including production rule systems and active database systems. Priority based execution control is also found in many Constraint (Logic) Programming systems. We refer to [2] for a deeper discussion. The implementation presented here is based on lazy matching and hence has the advantage of low memory requirements compared to RETE style eager matching. In [8], a rule based language with prioritized rules is presented, and an implementation based on a form of eager matching is proposed. [1] shows this language easily translates into CHR^{rp}. CHR^{rp} goes beyond earlier priority based rewriting systems by interacting with an underlying solver and supporting propagation rules.

Future Work The late indexing and passive occurrence optimizations were inspired by similar optimizations in the K.U.Leuven CHR system (and earlier

systems). Some other optimizations could easily be transferred to the CHR^{rp} compiler. The analyses implemented so far are very ad hoc and a more general approach based on abstract interpretation could be worthwhile. Finally, we have not taken advantage of some of the nondeterminism introduced by the ω_p semantics. In particular this concerns reordering or merging rules with equal priority. An extension of the join ordering cost model of [3] could help us choose a more optimal rule order within the boundaries imposed by the priorities.

References

1. L. De Koninck, T. Schrijvers, and B. Demoen. The correspondence between the Logical Algorithms language and CHR. In *23rd Intl. Conf. on Logic Programming*, volume 4670 of *LNCS*, pages 209–223, 2007.
2. L. De Koninck, T. Schrijvers, and B. Demoen. User-definable rule priorities for CHR. In *9th ACM SIGPLAN Symp. on Principles and Practice of Declarative Programming*, pages 25–36, 2007.
3. L. De Koninck and J. Sneyers. Join ordering for Constraint Handling Rules. In *4th Workshop on Constraint Handling Rules*, pages 107–121, 2007.
4. L. De Koninck, P. J. Stuckey, and G. J. Duck. Optimized compilation of CHR^{rp}. Technical Report CW 499, K.U.Leuven, Belgium, 2007.
5. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *20th Intl. Conf. on Logic Programming*, volume 3132 of *LNCS*, pages 90–104, 2004.
6. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artif. Intell.*, 19(1):17–37, 1982.
7. T. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Log. Program.*, 37(1-3):95–138, 1998.
8. H. Ganzinger and D. A. McAllester. Logical algorithms. In *18th Intl. Conf. on Logic Programming*, volume 2401 of *LNCS*, pages 209–223, 2002.
9. C. Holzbaur, M. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(4 & 5):503–531, 2005.
10. D. P. Miranker. TREAT: A better match algorithm for AI production system matching. In *6th National Conf. on Artificial Intelligence*, pages 42–47. AAAI Press, 1987.
11. F. Morawietz. Chart parsing and constraint programming. In *18th Intl. Conf. Computational Linguistics*, pages 551–557. Morgan Kaufmann, 2000.
12. T. Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, 2005.
13. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and application. In *1st Workshop on CHR, Selected Contributions*, Ulmer Informatik-Berichte 2004-01, pages 1–5. Universität Ulm, Germany, 2004.
14. T. Schrijvers et al. The Constraint Handling Rules home page, 2007. http://www.cs.kuleuven.be/~dtai/projects/CHR/.
15. T. Schrijvers and T. Frühwirth. Optimal union-find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.
16. J. Sneyers, T. Schrijvers, and B. Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In *20th Workshop on Logic Programming*, INFSYS Research Report 1843-06-02, pages 182–191. TU Wien, 2006.