Transformation-based Indexing Techniques for Constraint Handling Rules

Beata Sarna-Starosta
and Tom Schrijvers*

[1] Department of Computer Science and Engineering, Michigan State University, USA
LogicBlox Inc., Atlanta, Georgia, USA
bss@cse.msu.edu
[2] Department of Computer Science, K.U.Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract. Multi-headed rules are essential for the expressiveness of Constraint Handling Rules (CHR), but incur considerable performance overhead. Current indexing techniques are often unable to address this problem—they require matchings to have particular form, or offer good run-time complexity rather than good absolute figures.
We introduce two lightweight program transformations, based on term flattening, which improve the effectiveness of existing CHR indexing techniques, in terms of both complexity and constant factors. We also describe a set of complementary post-processing program transformations, which considerably reduce the flattening overhead.
We compare our techniques with the current state of the art in CHR compilation, and measure their efficacy in K.U.Leuven CHR and CHRd.

1 Introduction

Constraint Handling Rules (CHR) [5] is a high-level rule-based declarative programming language, usually embedded in a host language such as Prolog or Haskell. Typical applications of CHR include scheduling [1] and type checking [20]. CHR features *multi-headed rules*, i.e., rules with multiple predicates on the left-hand side (the *head*), which sets it apart from conventional declarative languages, e.g., Prolog or Haskell, where a rule's head admits only one predicate or function.

Multi-headed rules afford much of CHR's expressive power by allowing to easily combine information from distinct constraints via matching. However, as the matching procedure significantly affects the complexity of rule evaluation, this source of expressiveness often leads to performance bottlenecks. This effect is borne out by the approximative complexity formula of [6], where the multiplicity of rule's head appears in the exponent.

Aware of this problem, CHR developers have built data structures to support efficient indexing on variables (attributed variables [8]) and ground data

* Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen)

(search trees [9]). With [16] came the realization that $\mathcal{O}(1)$ indexing is essential to implement CHR algorithms with optimal complexity, which led to the use of hash tables for indexing ground data, and the general result that the complexity of CHR systems equals that of RAM machines [18]. CHRd [13] has slimmed the original indexing techniques based on attributed variables for faster evaluation of the class of direct-indexed CHR and use in a tabulated environment.

In this paper we advance the research on CHR indexing with the following contributions:

- two independent program transformations that improve the indexing behavior of CHR in the presence of function symbols
- a sequence of program post-processing steps that eliminate the overhead incurred by the indexing transformations
- the experimental measurements that clearly demonstrate the potential for time complexity improvements and the practical usefulness of our approach in K.U.Leuven CHR and CHRd
- the implementation of the presented transformations[1]

Our presentation begins with an overview of the problem of indexing with partial structures (Section 2). We then introduce our two indexing techniques (Sections 3 and 4), and describe applicable post-processing steps (Section 5). Next, we present the experimental evaluation of all proposed transformations (Section 6), relate our approach to other work (Section 7), and conclude (Section 8).

2 Problem Overview

The Problem CHR systems build indexes on the constraint store to speed up matching multi-headed rules. Consider the rule:

```
a(X), b(X,Y) ==> write(Y).
```

Given `X`, an index returns all stored constraints `b(X,Y)`. Thus, for a new `a(X)` we can quickly find all matchings of the form `b(X,Y)` that make the rule fire. Now consider the variant of the previous rule:

```
a(X), b(f(_,_),Y) ==> write(Y).
```

Here, for efficient matching, we need an index that returns all instances of `b/2` in which the first argument has top-level function symbol `f/1`. Currently, CHR systems do not generate indexes that involve partial structure of the constraints. Instead, they enumerate all `b(A,B)` in the constraint store and, for each `A`, test whether its top-level function symbol is `f/1`. When only a small fraction of all

[1] available at `http://www.cs.kuleuven.be/~toms/CHR/Indexing/`

As have this form, there are many failing tests. The problem becomes even more apparent if partial structures are parameterized, as in the rule:

```
a(X), b(f(X,_),Y) ==> write(Y).
```
(2.1)

Existing CHR systems cannot exploit the variable `X` to find all stored constraints matching `b(f(X,_),Y)` more quickly: as before, finding the matchings for an active constraint `a(X)` requires that all stored `b/2` constraints are enumerated.

The Solution We propose two techniques—both based on *term flattening*—for building indexes on partial structures. The first, *generic flattening* (Section 3), transforms rule (2.1) into:

```
a(X), b'(f,X,_,Y) ==> write(Y),
```

and the second, *constraint symbol specialization* (Section 4), into:

```
a(X), b_f(X,_,Y) ==> write(Y).
```

As source-to-source transformations, both proposed techniques are portable to many CHR systems. Moreover, since they both reuse available indexing data structures, further optimizations of such data structures also improve the indexing performance of our techniques. As we prove in [14], both proposed techniques preserve the theoretical [5] and refined [2] semantics of CHR, as well as the set-based semantics of CHRd [13].

Preliminaries We restrict our presentation to CHR programs where each rule head contains at most one occurence of a functional term, at a fixed argument position of some constraint c/n. We consider the ith argument of c/n, and a given set F of function symbols f_j/a_j that appear in the rule heads at the ith position of c/n. We define the *maximal arity* of F as $a_{max} = \max_{f_j/a_j \in F}(a_j)$.

We assume that, at run time, all instances of c/n have the top-level function symbol in their ith argument instantiated, but not necessarily to a symbol in F. This assumption can be satisfied by groundness analysis [17] or programmer-supplied mode annotations.

Example 1. The first argument of the constraint $c/1$ from the CHR program in Table 1 takes on function symbols given by the set $F = \{f/2, g/1\}$. The maximal arity of F, a_{max}, is 2.

3 Generic Flattening

Our first flattening approach augments the arity of each constraint symbol c, which appears in the heads of the program rules with function-term arguments, to accommodate new arguments of c representing these function terms.

```
:- chr_constraint c/1.

r1 @ c(X) \ c(X) <=> true.
r2 @ c(f(X,Y))   ==> c(X), c(Y).
r3 @ c(g(X))     ==> c(X).
r4 @ c(X)        <=> write(X).
```

Table 1. A CHR program with constraints on function symbols

Definition 1 (Flattening and Unflattening Functions). *The* flattening function ϕ *with respect to the set of function symbols* F*, maps a term* T *onto a sequence of terms:*

$$\phi(T) = \begin{cases} f_j, \bar{t}, \underbrace{e, \ldots, e}_{d_j} & \text{if } T = f_j(\bar{t}) \text{ s.t. } f_j/a_j \in F \text{ and } |\bar{t}| = a_j \\ T, \underbrace{e, \ldots, e}_{a_{max}} & \text{otherwise} \end{cases}$$

where $d_j = a_{max} - a_j$ *and* e *is an arbitrary constant.*

The unflattening function $\psi = \phi^{-1}$ *maps a sequence of terms onto a term:*

$$\psi(T) = \begin{cases} f_j(\bar{t}) & \text{if } T = f_j, \bar{t}, \bar{e} \text{ s.t. } f_j/a_j \in F \text{ and } |\bar{t}| = a_j \text{ and } |\bar{e}| = a_{max} - a_j \\ t & \text{if } T = t, \bar{e} \text{ s.t. } |\bar{e}| = a_{max} \end{cases}$$

Example 2. The flattening and unflattening functions for the CHR program in Table 1, with $F = \{f/2, g/1\}$, are defined as:

$$\phi(T) = \begin{cases} f, X, Y & \texttt{if } T = f(X,Y) \\ g, Z, e & \texttt{if } T = g(Z) \\ T, e, e & \texttt{otherwise} \end{cases}$$

$$\psi(A,B,C) = \begin{cases} f(X,Y) & \texttt{if } A,B,C = f,X,Y \\ g(Z) & \texttt{if } A,B,C = g,Z,e \\ T & \texttt{if } A,B,C = T,e,e \end{cases}$$

The original and flattened instances of the constraints are related by the flattening rules:

Definition 2 (Flattening Rule). *The* flattening rule Φ *replaces a given constraint* c/n *with its flat form:*

$$\Phi \texttt{ @ } c(\overline{t_{1,\ldots,i-1}}, t_i, \overline{t_{i+1,\ldots,n}}) \texttt{ <=> } c'(\overline{t_{1,\ldots,i-1}}, \phi(t_i), \overline{t_{i+1,\ldots,n}}) \tag{3.2}$$

Example 3. The flattening rule for constraint `c/1` of the CHR program in Table 1 is listed in line 3 of Table 2.

```
:- chr_constraint c/1, c'/3.                                    1
                                                                2
 Φ  @ c(T) <=> c'(φ(T)).                                        3
                                                                4
r1' @ c'(A,B,C) \ c'(A,B,C) <=> true.                           5
r2' @ c'(f,X,Y) ==> c(X), c(Y).                                 6
r3' @ c'(g,X,e) ==> c(X).                                       7
r4' @ c'(A,B,C) <=> T = ψ(A,B,C) | write(T).                    8
```

Table 2. CHR program from Table 1 after generic flattening

Definition 3 (Flattened Rule). *The flattening ϕ of a CHR rule with respect to the ith argument of a constraint c/n for a set of function symbols F is defined as:*

$$\phi(H \texttt{ ?=> } G \mid B) \;=\; H' \texttt{ ?=> } G', G \mid B$$

where

- *H' differs from H in that any constraint $c(t_1, \ldots, t_i, \ldots, t_n)$ is replaced by the flattened form $c'(t_1, \ldots, \phi(t_i), \ldots, t_n)$*
- *the new guard G' relates the flattened arguments back to the original ones, and contains one t_i `=` $\psi(\phi(t_i))$ for each flattened argument.*

Example 4. Lines 5-8 of Table 2 list flattened rules of the CHR program in Table 1, partially post-processed (Section 5) for readability.

Definition 4 (Flattened Program). *The flattening $\phi(\mathcal{P})$ of a CHR program $\mathcal{P}$ given by the set of rules $\overline{R}$, with respect to the ith argument of a constraint c/n, for a set of function symbols F, is defined as the flattening of each rule in $\overline{R}$, the flattening and unflattening functions, and the encoding of Φ:*

$$\phi(P) = \phi(\overline{R}) \cup \phi \cup \psi \cup \Phi$$

Example 5. Table 2 shows the program from Table 1 after generic flattening.

4 Constraint Symbol Specialization

Our second flattening technique differs from the first one in that it uses a different constraint symbol for each function symbol. As a consequence, it defines one flattening function and multiple unflattening functions:

Definition 5 (Flattening and Unflattening Functions). *The* flattening function *ϕ with respect to the set of function symbols F, maps a term T onto a sequence of terms:*

$$\phi(T) = \begin{cases} \bar{t} & \text{if } T = f_j(\bar{t}) \text{ s.t. } f_j/a_j \in F \text{ and } |\bar{t}| = a_j \\ T & \text{otherwise} \end{cases}$$

The unflattening function *for a function symbol f_j/a_j, ψ_{f_j}, maps a sequence of terms onto a term: $\psi_{f_j}(t_1, \ldots, t_{a_j}) = f_j(t_1, \ldots, t_{a_j})$. The* default unflattening function *is the identity function': $\psi'(t) = t$.*

Example 6. The flattening and unflattening functions for the CHR program in Table 1, with $F = \{f/2, g/1\}$, are defined as:

$$\phi(T) = \begin{cases} X, Y & \texttt{if } T = f(X,Y) \\ Z & \texttt{if } T = g(Z) \\ T & \texttt{otherwise} \end{cases}$$

$$\begin{aligned} \psi_f(X,Y) &= f(X,Y) \\ \psi_g(Z) &= g(Z) \\ \psi'(T) &= T \end{aligned}$$

Note that the flattening function is the *left inverse* of each unflattening function:

$$\phi \circ \psi_{f_j}(\bar{t}) = \bar{t} \tag{4.3}$$
$$\phi \circ \psi'(t) = t \tag{4.4}$$

and it is the *right inverse* for terms with appropriate function symbols:

$$\psi_{f_j} \circ \phi(f_j(\bar{t})) = f_j(\bar{t}) \qquad \text{where } f_j \in F \tag{4.5}$$
$$\psi' \circ \phi(f(\bar{t})) = f(\bar{t}) \qquad \text{where } f \notin F \tag{4.6}$$

Hence, unlike in the previous section, the flattening function is (in general) not injective: the original term t cannot be reconstructed from its flattened form $\bar{t}$ alone. Consider $F = \{f/2, g/2\}$ and two terms, $f(a,b)$ and $g(a,b)$. Since both terms flatten to a, b, the original term cannot be determined based solely on this flat form, because it does not preserve the function symbol. To enable recovery of this information, we encode it in the flattened constraint symbol.

Definition 6 (Flattening Rules). *The* flattening rules Φ_{f_j} *and* Φ' *replace a given constraint* c/n *with its flat form:*

$$\Phi_{f_j} \ \texttt{@}\ c(\overline{t_{1,\ldots,i-1}}, t_i, \overline{t_{i+1,\ldots,n}}) \ \texttt{<=>}\ t_i = f_j(t'_1, \ldots, t'_{a_j}) \wedge f_j/a_j \in F$$
$$\texttt{|}\ c_{f_j}(\overline{t_{1,\ldots,i-1}}, \phi(t_i), \overline{t_{i+1,\ldots,n}}) \tag{4.7}$$
$$\Phi' \ \texttt{@}\ c(\overline{t_{1,\ldots,i-1}}, t_i, \overline{t_{i+1,\ldots,n}}) \ \texttt{<=>}\ t_i = f(t'_1, \ldots, t'_a) \wedge f/a \notin F$$
$$\texttt{|}\ c'(\overline{t_{1,\ldots,i-1}}, \phi(t_i), \overline{t_{i+1,\ldots,n}}) \tag{4.8}$$

Now we can distinguish between the flattened constraints $c_f(a,b)$ and $c_g(a,b)$ by looking at constraint symbols c_f and c_g: the former constraint originates from $c(f(a,b))$, and the latter from $c(g(a,b))$.

As each original constraint symbol maps to multiple specialized constraint symbols, rule flattening in case of symbol specialization is more complex than for generic flattening. Every original rule maps to multiple flattened rules, one for each combination of specialized constraint symbols.

Definition 7 (Flattened Rules). *The flattening* ϕ *of a CHR rule* $(H \ \texttt{?=>}\ G \mid B)$ *with respect to the* i*th argument of a constraint* c/n *for the set of function symbols* F *is the set with maximal cardinality, containing rules of the form:*

$$H' \ \texttt{?=>}\ G', G \mid B$$

```
:- chr_constraint c/1, c_f/2, c_g/1, c'/1.

Φ_f @ c(T) <=> T = f(X,Y) | c_f(φ(T)).
Φ_g @ c(T) <=> T = g(Z)   | c_g(φ(T)).
Φ'  @ c(T) <=> T ≠ f(X,Y), T ≠ g(Z) | c'(φ(T)).

r1_f @ c_f(X,Y) \ c_f(X,Y) <=> true.
r1_g @ c_g(Z)   \ c_g(Z)   <=> true.
r1'  @ c'(T)    \ c'(T)    <=> true.
r2_f @ c_f(X,Y) ==> c(X), c(Y).
r3_g @ c_g(Z)   ==> c(Z).
r4_f @ c_f(X,Y) <=> R = ψ_f(X,Y) | write(R).
r4_g @ c_g(Z)   <=> R = ψ_g(Z)   | write(R).
r4'  @ c'(T)    <=> R = ψ'(T)    | write(R).
```

Table 3. CHR program from Table 1 after constraint symbol specialization

where

- *H' differs from H in that any constraint $c(t_1, \ldots, t_i, \ldots, t_n)$ is replaced by a specialized flattened form, $c_{f_j}(t_1, \ldots, \overline{t'}, \ldots, t_n)$ if $t_i = f_j(t'_1, \ldots, t'_{a_j})$ s.t. $f_j/a_j \in F$, or $c'(t_1, \ldots, t_i, \ldots, t_n)$ otherwise*
- *the new guard G' contains the pre-condition: one $\psi_{f_j}(\overline{t'})$ = t_i for each flattened argument $t_i = f_j(t'_1, \ldots, t'_{a_j})$ s.t. $f_j/a_j \in F$, or $\psi'(t')$ = t' otherwise.*

The flattened program is defined as for generic flattening:

Definition 8 (Flattened Program). *The flattening $\phi(P)$ of a CHR program P given by the set of rules $\overline{R}$, with respect to the ith argument of a constraint c/n, for the set of function symbols F, is defined as the flattening of each rule in $\overline{R}$, the flattening and unflattening functions and the flattening rules:*

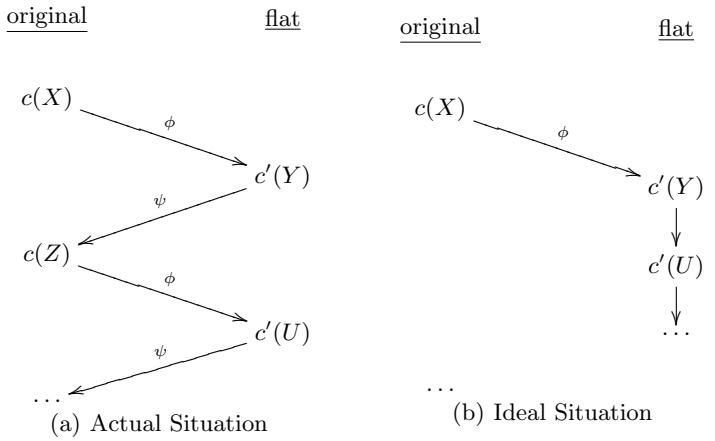
$$\phi(P) = \phi(\overline{R}) \cup \phi \cup \bigcup_{f_j \in F} (\psi_{f_j} \cup \Phi_{f_j}) \cup (\psi' \cup \Phi')$$

Example 7. Table 3 shows the program from Table 1 flattened using constraint symbol specialization. The rules $\texttt{r1}_f$ to $\texttt{r3}_g$ have been simplified for readability (see Section 5).

5 Post-Processing

The experimental results in Section 6 indicate that the performance improvement attained by better indexing is offset, or in some cases even surpassed, by the run-time overhead of applying the flattening and unflattening functions. In this section we outline two transformations that statically eliminate most of this overhead. The effectiveness of these transformations is borne out by the benchmarks in Section 6.

5.1 Repeated Flattening and Unflattening



(a) Actual Situation (b) Ideal Situation

Fig. 1. Transitions between original and flattened constraints

Alternating flattening and unflattening of values is a major source of runtime overhead. In a typical scenario (Fig. 1(a)), a value is flattened and matched in a head of a rule, then it is unflattened in that rule's body for calling a new constraint, flattened again to match another rule, and so on. To avoid this overhead, the transformed rules should operate solely on the flattened constraints, whereas the unflattened constraints should be called only by the queries external to the programs.

We propose a four-step rewriting procedure that aims to trigger this ideal scenario (Fig. 1(b)). Execution of a program enhanced with our procedure consists of two phases:

(1) constraint flattening, and
(2) processing of the flattened constraints.

For all but the most trivial programs, we expect the runtime cost of (1) to be marginal with respect to the cost of (2).

We formulate the steps of the procedure in terms of generic flattening; their counterparts for constraint symbol specialization can be easily derived.

Step 1: Make flattening explicit.
Unfold constraint calls according to the flattening rules.

Example 8. Flattening the rule `d(X,N) <=> N>0, d(X,N-1)` w.r.t. the first argument of `d(X,N)` yields:

`d'(A,B,N) <=> X = ψ(A,B), N>0 | d(X,N-1).`

By applying Step 1 to the above rule we obtain:

```
d'(A,B,N) <=> X = ψ(A,B), N>0
            | (A1,B1) = φ(X), d'(A1,B1,N-1).
```

We refer the reader to the work of Tacchella et al. [21] for the formal definition and correctness proof of unfolding of CHR rules.

Step 2: Eliminate flattening after unflattening.
Apply the following equation from left to right:

$$\forall \bar{t} : \phi \circ \psi(\bar{t}) = \bar{t}$$

which is valid since ϕ is the (left) inverse of ψ.

Example 9. Applying Step 2 to the last rule in Example 8 yields:
`d'(A,B,N) <=> X = ψ(A,B), N>0 | d'(A,B,N-1).`

Step 3: Move matchings from unflattened to flattened values.
Apply the equivalence from left to right:

$$\forall \overline{t_1}, \overline{t_2} : \psi(t_1) = \psi(t_2) \Leftrightarrow \overline{t_1} = \overline{t_2}$$

which is valid since ψ is injective.

Example 10. Consider rule `r1` in Table 1. Since the head constraints share the variable `X`, before transformation the rule should be normalized. Flattening the normalized rule yields:

```
c'(A1,B1,C1) \ c'(A2,B2,C2) <=>
        TX = ψ(A1,B1,C1), TY = ψ(A2,B2,C2), TX = TY | true.
```

By applying Step 3, we obtain:

```
c'(A,B,C) \ c'(A,B,C) <=> TX = ψ(A,B,C) | true.
```

Step 4: Clean up.
Drop unused unflattening guards and refold the unfolded constraint calls that could not be simplified.

Example 11. Applying Step 4 to the last rule in Example 10 yields:
`c'(A,B,C) \ c'(A,B,C) <=> true.`

In general, these rewriting steps are not sufficient to enforce our ideal scenario. However, as the results in Section 6 show, they have good practical effects.

5.2 Flattening Indirection

Another source of overhead stems from the processing indirection imposed by flattening, with which all constraints are flattened before the rules are executed. Usually this overhead is marginal. The main exception are, common in CHR, *unconditional simplification rules*—single-headed simplification rules in which the form of a rule's head uniquely determines whether or not that rule is applicable. As a typical example, consider the following fragment of the `zebra` program in our benchmark suite (Section 6):

```
domain(X,[])  <=> fail.
domain(X,[V]) <=> X=V.
domain(X,L1), domain(X,L2) <=> intersect(L1,L2,L), domain(X,L).
```

which after constraint symbol specialization takes the form:

```
domain(X,[]) <=> domain_[](X).
domain(X,[H|T]) <=> domain_[|](X,H,T).

domain_[](_) <=> fail.
...
```

The `domain_[]/1` constraint denotes a base case, which obviously *always*[2] simplifies to `fail`. Because of the flattening indirection, the otherwise short-lived `domain(X,[])` constraints live longer—the liftetime of two calls instead of one. To avoid this indirection overhead we inline the rule body:

```
domain(X,[])    <=> fail.
domain(X,[H|T]) <=> domain_[|](X,H,T).
```

The same technique applies to generic flattening.

6 Evaluation

We have implemented our optimizations in two CHR systems on SWI-Prolog: CHRd [13] and K.U.Leuven CHR [15]. All run times, given in seconds for the original programs and relative to the original for the transformed versions, were measured on an Intel Pentium 4, 2.00 GHz, with 512 MB RAM. Our benchmark suite[3] includes several common CHR programs [14]. For each optimization we consider only the relevant benchmarks, for which the transformed programs differ from the original ones.

6.1 Generic Flattening

Table 4 shows the results of generic flattening. For each CHR system we list run times measured without flattening or post-processing (−flat,−pp), with flattening but without post-processing (+flat,−pp), and with both flattening and post-processing (+flat,+pp).

In K.U.Leuven CHR generic flattening has little (but mainly positive) effect on all benchmarks, except for `mergesort`, with a speed-up close to 50%, and `gamma_prime`, with almost 40% of slow-down. In CHRd we observe an improvement in `gamma_prime`, `listdom` and `ram`, and a minimal overhead in `mergesort` and `zebra`. For these two small programs, the run-time cost of unflattening exceeds the savings provided by the transformation. All benchmarks demonstrate positive effects of post-processing, and we blame insufficient post-processing for the slow-down of `gamma_prime` in K.U.Leuven CHR: With stronger reasoning on the constraint argument types we (manually) achieved a relative timing of 99%. Hence, further improvement of the automated post-processing seems worthwhile.

[2] Note that the flattening transformation exposes the unconditionality.
[3] Available at `http://www.cs.kuleuven.ac.be/~toms/CHR/Indexing/`

benchmark	K.U.Leuven CHR function symbols −flat −pp	+flat −pp	+flat +pp	CHRd function symbols −flat −pp	+flat −pp	+flat +pp
gamma_prime	3.1	219%	137%	5.4	111%	93%
listdom	5.0	114%	108%	6.9	86%	84%
mergesort	1.9	592%	56%	6.6	113%	103%
ram_op	8.8	130%	96%	8.3	94%	89%
ram_prog	2.9	102%	96%	4.4	91%	86%
zebra	5.1	124%	93%	6.4	113%	102%

Table 4. Run times (in sec.) for generic flattening benchmarks

6.2 Constraint symbol specialization

Table 5 shows the results of constraint symbol specialization. The columns in the table have the same meaning as in Table 4. Table 5 includes two new benchmarks, `zebra2` and `manners`, not reported in Table 4. The benchmark `zebra2` shows the effect of repeated (until a fixed point is reached) flattening on the `zebra` program: the unoptimized entry in `zebra2` corresponds to the entry in `zebra` processed with the (+flat,+pp) option. The `manners` benchmark involves constraints with constant but no partial-structure arguments, and hence it is not improved by generic flattening. We use this benchmark to demonstrate that constraint symbol specialization may improve the performance of programs without partial structures.

Even before post-processing, constraint symbol specialization has good effects in both systems. In K.U.Leuven CHR only `gamma_prime` and `listdom` suffer performance slow-downs, whereas other benchmarks show run-time improvement. This success is caused by the system's guard optimization [19], which detects dead code for the specialized constraint symbols. Post-processing considerably improves the performance of `listdom` and eliminates the overhead of `gamma_prime`. It has no significant effect on other benchmarks. In CHRd we observe initial performance slow-down in `listdom` and `manners`, the former of which is eliminated by the post-processing step. For `manners`, the cost of processing extra constraints outweighs the benefits of specialization apparent in K.U.Leuven CHR.

In both systems, the repeated flattening of `zebra2` is unsuccessful—its incremental benefit is offset by the incremental overhead.

6.3 Improved Time Complexity

Although flattening improves the performance of most benchmarks in our suite, it does not decrease the complexity of the evaluation. We attribute this to the fact that the programmers—aware of CHR's poor handling of partial structures—tend to write already flattened programs, especially for problems which involve referencing partial structure arguments (as in rule (2.1)). Such practice, however, obscures formulation of problems where partial structures appear naturally and

benchmark	K.U.Leuven CHR function symbols −flat −pp	K.U.Leuven CHR function symbols +flat −pp	K.U.Leuven CHR function symbols +flat +pp	CHRd function symbols −flat −pp	CHRd function symbols +flat −pp	CHRd function symbols +flat +pp
gamma_prime	1.5	114%	94%	4.6	93%	83%
listdom	5.2	174%	99%	7.2	129%	90%
manners	2.2	70%	65%	4.9	131%	124%
mergesort	7.8	30%	30%	6.7	82%	81%
ram_op	8.5	81%	82%	7.5	87%	88%
ram_prog	2.9	93%	93%	4.5	82%	80%
zebra	5.1	96%	91%	6.3	97%	97%
zebra2	4.8	103%	100%	6.9	96%	97%

Table 5. Run times (in sec.) for constraint symbol specialization benchmarks

are extensively used, e.g., database reasoning. For problems of this kind, flattening does cause complexity improvement, thus extending applicability of CHR to their natural specifications.

For instance, consider a database of employees represented using the constraint `employee(`*Name*`,`*Date*`)`, in which the date of birth *Date* is a compound term `date(`*Day*`,`*Month*`,`*Year*`)`. The following rule finds out which employees' birthdays to celebrate on the current date:

```
check_birthdays(date(Day, Month, CurrentYear)),
        employee(Name, date(Day, Month, YearOfBirth)) ==>
                Age is CurrentYear - YearOfBirth,
                celebrate(Name, Age)
```

The following table lists the run times[4], in milliseconds, before and after flattening the compound date, for three database sizes. The original program exhibits a linear behavior, whereas the run time for the flattened version remains constant.

program version	number of employees 1,000	10,000	50,000
−flat −pp	2.000	22.000	108.000
+flat +pp	0.029	0.028	0.029

The impact of symbol specialization on the birthday program is virtually the same as for generic flattening w.r.t. both the complexity and absolute run times.

6.4 Discussion

The results in Tables 4 and 5 suggest that our flattening transformations may improve performance of CHR, however, additional optimizations—such as post-processing—are needed to fully exploit their potential.

Overall, in both systems constraint symbol specialization yields more run-time savings than generic flattening. This, in part, comes from the nature of

[4] in K.U.Leuven CHR; CHRd can evaluate only a transformed version of the rule.

our benchmarks, which do not exhibit the potential of symbol specialization for increasing the program size. The following table shows the number of CHR rules in the original benchmark programs and their two flattened versions:

benchmarks	original	generic	symbol specialization
`gamma_prime`	6	6	8
`listdom`	14	13	39 (13)
`manners`	12	n/a	14
`mergesort`	2	2	3
`ram`	13	13	13
`zebra`	5	4	4

Note that three transformed programs actually contain fewer rules than the corresponding original programs beacuse of inlining of the exposed unconditional simplification rules (see Section 5.2). We see a modest increase in two cases and a considerable increase for the symbol-specialized `listdom` benchmark: from 13 to 39 rules. In case of `listdom`, the blow-up is fully compensated by K.U.Leuven's guard simplification [19]: of the 39 rules, 26 rules have inconsistent guards. In general, however, the increase in program size may be too large to be contained. Such pathological cases must be detected and transformed using generic flattening rather than constraint symbol specialization.

7 Related Work

Most relational databases we are aware of do not support compound values. Hence, to map compound data onto (flat) rows requires application of techniques similar to the flattening transformations presented in this section.

Program Specialization The need for symbol specialization arises naturally in the context of partial evaluation [10]. Similar, but less ambitious in scope, is the work on constructor specialization for the Glasgow Haskell Compiler [12]. These two approaches, for *single-headed* languages, aim in the first place at reducing intermediate data structures and matching costs, and specializing the body. In contrast, the foremost goal of our approach, for multi-headed CHR rules, is to provide better constraint store indexes. Of course, our techniques benefit from the other effects as well.

Symbol Indexing Structures In XSB [22] specialized trie-like structures store previously computed answer substitutions. These substitutions are indexed on their call patterns, and interpreted as partial structure indexes for subsumption-based tabling. However, this approach requires excessive data structure implementation, does not enable further rule specialization, and does not easily compose with other indexes.

The join-calculus [4] features multi-headed rules that are similar in nature to CHR's rules. However, the expressivity of these rules is severly limited: arguments cannot be matched and rules cannot be otherwise guarded. The main

motivation for this limited expressivity is enabled compilation to a highly efficient finite-state automata [3]. In recent work [11], a flattening specialization similar to ours has been proposed to somewhat lift the severe expressivity restrictions. In this context, the flattening does not improve performance, but rather makes it possible for the rules to be compiled at all.

CHR Program Transformation We are not the first to consider program transformation in the context of CHR. Frühwirth [7] proposed the specialization of rules (rather than constraints) with respect to a given goal (rather than head matchings). Tacchella et al. [21] introduced a general technique for unfolding CHR rules in the body of other rules. However, to the best of our knowledge, we are the first to utilize program transformation for performance improvement: we have a fully automated implementation and empirical evidence of its effectiveness. Neither of the above works makes any specific claims about performance improvement, nor demonstrates practical usefulness of the reported technique.

8 Conclusion & Future Work

We presented two transformational techniques improving the performance of CHR indexing: generic and specialized function symbol flattening, and a complementary post-processing procedure that compensates their potential overhead.

All techniques have been implemented for the CHRd and K.U.Leuven CHR systems on SWI-Prolog. Evaluation on a set of benchmarks shows that the indexing optimizations enable performance improvement, and that the post-processing is a critical step towards the full realization of their potential. Our approach enables CHR programmers to exploit structured constraint arguments that most naturally fit their applications.

We restrict our attention to function-symbol arguments that are always instantiated. Adding support for uninstantiated arguments is a natural next step in our work. We anticipate this step to further increase the flattening overhead, as well as its complexity when abiding by CHR's refined operational semantics.

The presented framework for CHR program transformation (based on flattening, unflattening and post-processing) is proving useful for expressing other indexing approaches. Our current work concerns a technique for ground terms, called *attributed data* [14], which has better constant factors than hash-tables.

References

1. Slim Abdennadher and Michael Marte. University course timetabling using constraint handling rules. *Applied Artificial Intelligence*, 14(4):311–325, 2000.
2. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *20th International Conference on Logic Programming (ICLP)*, pages 90–104, 2004.
3. Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. *Electronic Notes on Theoretical Computer Science*, 16(3), 1998.

4. Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *International Summer School on Applied Semantics*, pages 268–332, 2000.
5. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
6. Thom Frühwirth. As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs. *Electronic Notes in Theoretical Computer Science*, 59(3), 2002.
7. Thom Frühwirth. Specialization of Concurrent Guarded Multi-Set Transformation Rules. In *Logic Based Program Synthesis and Transformation (LOPSTR), Revised Selected Papers*, 2005.
8. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), April 2000.
9. Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming*, 5(Issue 4 & 5):503–531, 2005.
10. Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
11. Qin Ma and Luc Maranget. Compiling pattern matching in join-patterns. In *15th International Conference on Concurrency Theory (CONCUR)*, pages 417–431, 2004.
12. Simon Peyton-Jones. Constructor Specialization for Haskell Programs. In *12th International Conference on Functional Programming (ICFP)*, 2007.
13. Beata Sarna-Starosta and C.R. Ramakrishnan. Compiling Constraint Handling Rules for Efficient Tabled Evaluation. In *9th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2007.
14. Beata Sarna-Starosta and Tom Schrijvers. Indexing techniques for CHR based on program transformation. Report CW 500, K.U.Leuven, Department of Computer Science, August 2007.
15. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In *1st Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.
16. Tom Schrijvers and Thom Frühwirth. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.
17. Tom Schrijvers, Peter Stuckey, and Gregory Duck. Abstract Interpretation for Constraint Handling Rules. In *7th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2005.
18. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules (CHR)*, pages 3–17, 2005.
19. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and continuation optimization for occurrence representations of CHR. In *21st International Conference on Logic Programming (ICLP)*, pages 83–97, 2005.
20. Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transactions on Programming Languages and Systems*, 27(6):1216–1269, 2005.
21. Paolo Tacchella, Maurizio Gabbrielli, and Maria Chiara Meo. Unfolding in chr. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 179–186, 2007.
22. David S. Warren et al. The XSB Programmer's Manual: version 2.7, vols. 1 and 2, January 2005. http://xsb.sf.net.