Indexing Techniques for CHR based on program transformation

Beata Sarna-Starosta
Tom Schrijvers

Report CW 500, August 2007

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Indexing Techniques for CHR based on program transformation

Beata Sarna-Starosta
Tom Schrijvers

Report CW 500, August 2007

Department of Computer Science, K.U.Leuven

Abstract

Multi-headed rules are essential for the expressiveness of CHR, but incur a considerable performance penalty. Current indexing techniques are often unable to address this problem. They are effective only when matchings have a particular form, or offer good run-time complexity rather than good absolute figures.

In this paper we describe three advanced indexing techniques: (1) two program transformations that make other indexing techniques more effective, (2) an index for ground terms more efficient than hash tables, and (3) a post-processing program transformation that eliminates runtime overhead of (1) and (2). We compare these techniques with the current state of the art, and give measurements of their effectiveness in K.U.Leuven CHR and CHRd.

Keywords : Constraint Handling Rules, indexing, program transformation, program specialization
CR Subject Classification : D.3.2 Language Classifications (Constraint and logic languages), D.3.4 Processors (Compilers).

Indexing Techniques for CHR based on program transformation

Beata Sarna-Starosta[1] and Tom Schrijvers*[2]

[1] Department of Computer Science and Engineering, Michigan State University, 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 CHR, but incur a considerable performance penalty. Current indexing techniques are often unable to address this problem. They are effective only when matchings have a particular form, or offer good run-time complexity rather than good absolute figures.
In this paper we describe three advanced indexing techniques: (1) two program transformations that make other indexing techniques more effective, (2) an index for ground terms more efficient than hash tables, and (3) a post-processing program transformation that eliminates run-time overhead of (1) and (2). We compare these techniques with the current state of the art, and give measurements of their effectiveness in K.U.Leuven CHR and CHRd.
keywords: Constraint Handling Rules, indexing, program transformation, program specialization

1 Introduction

CHR is a high-level rule-based programming paradigm. Much of the expressive power of CHR stems from the multi-headedness of its rules, which allows to easily combine information from distinct constraints via matching. As the head multiplicity exponentially affects the complexity of rule evaluation [4], this source of expressiveness often contributes to performance bottlenecks. Aware of this problem, CHR developers have built data structures supporting efficient indexing on variables (attributed variables [6]) and ground data (search trees [7]). With [14] came the realization that $\mathcal{O}(1)$ indexing is essential for implementing 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 [16]. CHRd [11] has slimmed the original attributed variable indexing 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:

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

– two transformational approaches that improve indexing behavior of CHR programs with respect to function symbols (Section 2),
– an alternative to hash tables for indexing ground data that does not suffer from amortization-related overhead (Section 3),
– post-processing program transformations that eliminate the disadvantages of the indexing optimizations (Section 4),
– measurements that demonstrate the usefulness of all presented techniques in K.U.Leuven CHR and CHRd (Section 5).

The effectiveness of an indexing technique depends on the structure of the program it is applied to. The techniques we propose focus on two structural elements—function symbols and ground data—and for many programs that rely on these elements, provide satisfactory performance improvement, thus helping increase the reliability of CHR evaluation in general.

We assume that the reader is familiar with the syntax and operational semantics of CHR. For an overview we refer to [3, 12].

2 Function Symbol Indexing

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`. Current 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 `A`s have this form, there are many failing tests. Partial structures may even be parameterized:

```
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 indexing on partial structures. The first, *generic flattening* (Section 2.1), transforms rule (2.1) into: `a(X), b'(f,X,_,Y) ==> write(Y)`, and the second, *constraint symbol specialization* (Section 2.2), into: `a(X),` $\texttt{b}_f$`(X,_,Y) ==> write(Y)`. As source-to-source transformations, the proposed techniques are portable to many CHR systems. Furthermore, they both reuse available indexing data structures. Hence, further optimizations of such data structures, e.g. that of Section 3, also improve partial structure indexing.

Preliminaries For both approaches, we consider the ith argument of a constraint 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 the constraint 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 their top-level function symbol instantiated, but not necessarily to one in F. This assumption can be satisfied by groundness analysis [15] or programmer-supplied mode annotations.

Example 1. For the CHR program in Table 1(a), we are concerned with the first argument of the constraint $c/1$. In the program, this argument takes on function symbols given by the set $F = \{f/2, g/0\}$. The maximal arity of F, a_{max}, is 2.

2.1 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. Formally:

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:*

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

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:*

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

We encode the unflattening and flattening functions as auxiliary Prolog predicates, the former invoked when the rule bodies reference function symbols from the heads, and the latter when the rule bodies pose the unflattened constraints.

Example 2. Table 1(b) shows the flattening (lines 2-4) and unflattening (lines 5-7) predicates for the constraint `c/1` from Table 1(a).

Another auxiliary predicate encodes the correspondence between the original and flattened instances of the constraints:

Definition 2 (Flattening Correspondence). *The* flattening correspondence *Φ relates a constraint instance $c(\overline{s}, t, \overline{u})$ to a flattened constraint instance $c'(\overline{s}, \overline{t}, \overline{u})$:*

$$\forall \overline{s}, t, \overline{t}, \overline{u} : \phi(t) = \overline{t} \quad \Longleftrightarrow \quad c(\overline{s}, t, \overline{u}) \Leftrightarrow c'(\overline{s}, \overline{t}, \overline{u}) \tag{2.2}$$

```
:- chr_constraint c/1.

c(X) \ c(X) <=> true.
c(f(X,Y)) ==> c(X), c(Y).
c(g(X)) ==> c(X).
c(X) <=> write(X).
```

(a)

```
:- chr_constraint c'/3.                    (1)

flatten(f(X,Y),A,B,C) :- !, A=f,B=X,C=Y. (2)
flatten(g(X),A,B,C    :- !, A=g,B=X,C=e. (3)
flatten(T,A,B,C)      :- A=T,B=e,c=e.    (4)

unflatten(f,A,B,T) :- !, T=f(A,B)).      (5)
unflatten(g,A,_,T) :- !, T=g(A).         (6)
unflatten(A,_,_,T) :- T=A.               (7)

c(T) :- flatten(T,A,B,C), c'(A,B,C).     (8)

c'(Y,Z,W) \ c'(Y,Z,W) <=> true.          (9)
c'(f,X,Y) ==> c(X), c(Y).               (10)
c'(g,X,Y) ==> c(X).                     (11)
c'(Y,Z,W) <=> unflatten(Y,Z,W,X)        (12)
              | write(X).
```

(b)

Table 1. CHR program with function-term arguments (a) after generic flattening (b)

Example 3. For the head constraints in the rules of the program in Table 1(a), $\Phi = \{c(f(X,Y)) \Leftrightarrow c'(f,X,Y),\ c(g(X)) \Leftrightarrow c'(g,X,e),\ c(X) \Leftrightarrow c'(X,e,e)\}$, and is encoded as the predicate `c/1` (Table 1(b), line 8).

Definition 3 (Flattened Rule). *The flattening ϕ of a CHR rule with respect to the ith argument of a constraint c/n for 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 $c'(t_1, \ldots, \overline{t}, \ldots, t_n)$, where $\overline{t}$ are fresh variables*
- *the new guard G' contains the pre-condition of the Equation (2.2): one t_i = $\psi(\overline{t})$ for each flattened argument.*

Example 4. Lines 9-12 of Table 1(b) show flattened rules of the program in Table 1(a), partially post-processed (Section 4) for readability.

Definition 4 (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 functions ϕ and ψ, and the encoding of Φ: $\phi(P) = \phi(\overline{R}) \cup \phi \cup \psi \cup \Phi$.*

Example 5. Table 1(b) shows generic flattening of the program from Table 1(a).

Our approach is sound and complete w.r.t. CHR's theoretical and refined operational semantics [2], and the set-based operational semantics [11]:

Theorem 1 (Soundness & Completeness). *Given a CHR program P, its flattening $\phi(P)$ and an initial state σ, a derivation from σ under $\phi(P)$ reaches the same (modulo flattening) final state as the derivation under P:*

$$\sigma \rightarrowtail_P^* \sigma_f \quad \Longleftrightarrow \quad \sigma \rightarrowtail_{\phi(P)}^* \phi(\sigma_f)$$

where ϕ is naturally extended to range over CHR execution states.

Proof. Sketch. We consider the theoretical operational semantics.

Soundness:

$$\sigma \rightarrowtail_{\phi(P)}^* \phi(\sigma_f) \quad \Longrightarrow \quad \sigma \rightarrowtail_P^* \sigma_f$$

In words, for any derivation in the flattened program of the form:

$$\sigma = \sigma_0 \rightarrowtail_{\phi(P)} \sigma_1 \rightarrowtail_{\phi(P)} \ldots \rightarrowtail_{\phi(P)} \sigma_n = \phi(\sigma_f)$$

there is a corresponding derivation in the original program:

$$\sigma = \sigma_0' \rightarrowtail_P \sigma_1' \rightarrowtail_P \ldots \rightarrowtail_P \sigma_m' = \sigma_f$$

with $m \leq n$.

We show this by induction from left to right through the derivation. For a derivation of length zero, i.e. only one state is involved, the theorem obviously holds. For every subsequent transition step $\sigma_i \rightarrowtail_{\phi(P)} \sigma_{i+1}$, we have that

- either a flattening transformation is applied, such that $\psi(\sigma_i) = \psi(\sigma_i + 1)$,
- or another transition step is taken.

In the first case, we do not extend the derivation for the original program. In the latter case, we can extend the derivation in the original program with $\psi(\sigma_i) \rightarrowtail_P \psi(\sigma_{i+1})$.

Finally, when the derivation for the transformed program reaches a fixed point, we have that $\sigma_n = \psi(\sigma_m)$. Hence, $\sigma_m = \phi(\sigma_n)$. Moreover, σ_m is a final state as σ_n is a final state and there are fewer derivations possible from σ_m in P than from σ_n in $\phi(P)$.

Completeness:

$$\sigma \rightarrowtail_P^* \sigma_f \quad \Longrightarrow \quad \sigma \rightarrowtail_{\phi(P)}^* \phi(\sigma_f)$$

The completeness proof has a similar structure, but now we introduce rather than eliminate flattening steps.

Both proofs are similar for the refined and set-based operational semantics, but involve more transition steps.

Confluence, termination and the complexity of the number of derivation steps of the transformed programs are preserved from the original ones. The encoding of Φ introduces additional derivation steps, the number of which is proportional to the number of new constraints introduced during the transformation.

Since the formal results of the optimizations proposed in the following sections are similar, we omit their discussion in this paper.

2.2 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:*

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

For a function symbol f_j/a_j, the unflattening function $\psi_{f_j}(t_1, \ldots, t_{a_j}) = f_j(t_1, \ldots, t_{a_j})$. *The default unflattening function is the identity function: $\psi'(t) = t$.*

Definition 6 (Flattening Correspondence). *The* flattening correspondence *Φ relates a constraint instance $c(\overline{s}, t, \overline{u})$ to a flattened constraint instance $c'(\overline{s}, \overline{t}, \overline{u})$:*

$$\forall \overline{s}, t, \overline{t}, \overline{u} : t = f_j(\overline{t}) \wedge \phi(t) = \overline{t} \iff c(\overline{s}, t, \overline{u}) \Leftrightarrow c_{f_j}(\overline{s}, \overline{t}, \overline{u})$$

$$\forall \overline{s}, t, \overline{t}, \overline{u} : \phi(t) = t \iff c(\overline{s}, t, \overline{u}) \Leftrightarrow c'(\overline{s}, t, \overline{u})$$

Definition 6 yields an important corollary:

Corollary 1.

$$\forall \overline{s}, t, \overline{t}, \overline{u} : c(\overline{s}, t, \overline{u}) \iff \bigvee_{j=1}^{n} \Big(\psi_{f_j}(\overline{t}) = t \wedge c_{f_j}(\overline{s}, \overline{t}, \overline{u})\Big) \vee \Big(\psi'(t) = t \wedge c'(\overline{s}, t, \overline{u})\Big)$$

Constraint symbol specialization differs from generic flattening in that it may map a single CHR rule to multiple flattened rules.

Definition 7 (Flattened Rule). *The flattening ϕ of a CHR rule (H `?=>` $G \mid B$) with respect to the ith argument of a constraint c/n for the set of function symbols F is a set of rules of the form:*

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

There is one rule for each disjunction in Corollary 1. In each flattened rule:

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

The flattened program is defined as for generic flattening.

Example 6. Table 2 shows the program from Table 1(a) flattened using constraint symbol specialization: lines 2-4 and 5-7 encode, resp., the flattening and unflattening functions; line 8 implements flattening corrspondence Φ; whereas lines 9-16 show the flattened rules of the original program (the rules in lines 9-11 (resp. 14-16) represent flattening of the first (resp. last) rule in Table 1(a)).

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

flatten(f(X,Y),C) :- !, C = c_f(X,Y).                      (2)
flatten(g(X),C)   :- !, C = c_g(X).                        (3)
flatten(X,C)      :- C = c'(X).                            (4)

unflatten_f(A,B,T) :- T = f(A,B)                           (5)
unflatten_g(A,T)   :- T = g(A).                            (6)
unflatten'(A,T)    :- T = A.                               (7)

c(T) :- flatten(T,C), call(C).                             (8)

c_f(X,Y) \ c_f(X,Y) <=> true.                              (9)
c_g(X) \ c_g(X) <=> true.                                 (10)
c'(X) \ c'(X) <=> true.                                   (11)
c_f(X,Y) ==> c(X), c(Y).                                  (12)
c_g(X) ==> c(X).                                          (13)
c_f(Y,Z) <=> unflatten_f(Y,Z,X) | write(X).               (14)
c_g(Y) <=> unflatten_g(Y,X) | write(X).                   (15)
c'(Y) <=> unflatten'(Y,X) | write(X).                     (16)
```

Table 2. CHR program with function terms after constraint symbol specialization

2.3 Discussion

Both presented flattening approaches enable indexing on partial structures. Generic flattening separates the function symbol from the rest of the term, so that it can be matched against as an ordinary argument. Constraint symbol specialization makes the matching against the function symbol implicit by considering only the constraints with the specialized symbol.

Each approach has its own trade-offs: Constraint symbol specialization admits more program specialization through copying the rules, whereas generic flattening keeps the size of the transformed programs linear w.r.t. the original ones. Although better specialization may improve the performance, increased program size causes slow-downs resulting from a larger number of cache misses.

Moreover, each technique suffers from its own form of space fragmentation: Generic flattening wastes $(a_{max} - a_j)$ argument slots for function symbol f_j/a_j per constraint instance, whereas constraint symbol specialization allocates space for specialized constraint symbols that may never be used.

Experimental evaluation in Section 5 allows us to determine whether either technique stands out in practice.

2.4 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.

The need for symbol specialization arises naturally in the context of partial evaluation [8]. Similar, but less ambitious in scope, is the work on constructor specialization for the Glasgow Haskell Compiler [10]. 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, we benefit from the other effects as well.

In XSB [18] 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.

2.5 Future Work

We consider only top-level function symbols in a single argument position. Repeated application extends either technique to multiple constraint arguments and nested function symbols. Although straightforward, repeated application may cause the program size—and hence performance—grow out of bounds. Constraint symbol specialization in particular quickly suffers from program size blow-ups. Thus, we note a need for heuristics to control the extent of specialization.

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 increase the overhead of flattening in general, and its complexity when abiding by the refined operational semantics.

The indexing of function symbols is one of the last gaps in the domain of equality-based indexing for CHR. However, the field for non-equality-based guards and indexing remains open. We believe that our approach can be adapted to this domain. A set of mutually exclusive guards, e.g. $N > 0$ and $N \leq 0$, could be used to classify the instances of a constraint $c(N)$. In the spirit of generic flattening a field would be added to identify the particular class, e.g. $c'(\mathit{positive}, N)$ for $N > 0$. Similarly, in the spirit of constraint symbol specialization, a distinct constraint symbol would be used for each class, e.g. $c_{>0}(N)$ for $N > 0$.

3 Attributed Data

In this section, we assume that a constraint contains ground-term arguments which are not matched against explicit function symbols, but only against other terms, i.e. that the ground terms are used as indexing keys. Hash tables provide $\mathcal{O}(1)$ indexing operations for indexing on ground terms, as do attributes for variables. However, the constant factor involved in hash tables is usually much larger. We propose a form of *attributed data* that provides $\mathcal{O}(1)$ indexing for ground data, with constant factors closer to those of attributed variables [5]. The key insight is that the CHR run time can internally use an attributed-variable–like representation for the externally provided ground term.

3.1 Indexing Key Declarations

We introduce the annotation '`as_chr_key`' to indicate that a particular ground argument of a constraint is to be used as an indexing key. An argument specifier '`+`*type* `as_chr_key` *keytype*' states that the argument is ground (`+`), and uses *type* as its external representation and *keytype* as its internal representation. The abstract *keytype* is generated automatically by the CHR compiler.

Example 7. Consider the RAM simulator [16] shown in Appendix A. The first argument of the `prog/4` constraint (representing the label of the current program instruction) is to be used as an indexing key with name `pkey`:

```
:- chr_constraint
        prog(+int as_chr_key pkey,+int as_chr_key pkey,+,+),
        prog_counter(+int as_chr_key pkey).
```

The same key is used for the second argument of `prog/4` (representing the label of the next program instruction), and for the only argument of `prog_counter/1`. In each case, the external representation is an integer, whereas the internal representation is of type `pkey`.

3.2 Indexing Key Representation

The representation of the internal data type *keytype* resembles that of the attributed variable indexes, except that it does not include the variable itself to avoid an unnecessary indirection. Thus, the internal representation *Int* is a term, the arguments of which are indexes $Index_i$ on different argument positions. In addition, *Int* maintains the original external representation *Ext*:

$$Int = \texttt{key(}Ext\texttt{,}Index_1\texttt{,}\ldots\texttt{,}Index_n\texttt{)}$$

There is exactly one internal representation for any given external representation.

Our default representation for the argument indexes $Index_i$ is a flat list of constraint suspensions, with predefined operations for adding and removing the constraints. The main structure itself can be updated (e.g. for replacing an old index with a new one) by the destructive argument update predicate `setarg/3` implemented by most Prolog systems.

3.3 Source-to-Source Transformation

In this section we define the source-to-source transformation for mapping between the external and internal representations of the indexing keys.

Definition 8 (Value Mapping Functions). *For an indexing key type t, the injective* internalizing function *ϕ maps an external value t_{Ext} of t onto the internal value t_{Int} of t: $\phi(t_{Ext}) = t_{Int}$. The injective* externalizing function *ψ is ϕ^{-1}: $\psi(t_{Int}) = t_{Ext}$.*

We use predicates `ext_val(`*Int*`,`*Ext*`)` and `int_val(`*Ext*`,`*Int*`)` to encode the externalizing and internalizing functions, respectively. The predicate `int_val/2` looks up the internal values in a hash table. If no internal value is present, a new one is created and added to the table. Its inverse, `ext_val/2`, is much cheaper:

```
ext_val(key(Ext,...), R) <=> R = Ext.
```

Definition 9 (Internalization Correspondence). *An external constraint instance $c(\overline{s}, t, \overline{u})$ corresponds to an internalized constraint instance $c'(\overline{s}, \overline{t}, \overline{u})$:*

$$\forall \overline{s}, t, \overline{t}, \overline{u} : \phi(t) = \overline{t} \quad \Longleftrightarrow \quad c(\overline{s}, t, \overline{u}) \Leftrightarrow c'(\overline{s}, \overline{t}, \overline{u})$$

The dynamic internalizing rule Φ applies the function ϕ at run time:

Definition 10 (Dynamic Internalizing Rule). *Given an internalizing function ϕ, Φ is of the form:*

$$\texttt{c(}t_1\texttt{,}\ldots\texttt{,}t_i\texttt{,}\ldots\texttt{,}t_n\texttt{) <=> } t'_i \texttt{ = } \phi(t_i)\texttt{, c}'\texttt{(}t_1\texttt{,}\ldots\texttt{,}t'_i\texttt{,}\ldots\texttt{,}t_n\texttt{).}$$

Example 8. The dynamic internalizing rule for `prog_counter/1` looks like:

```
prog_counter(Ext) <=> int_val(Ext,Int), prog_counter'(Int).
```

Definition 11 (Internalized Rule). *The internalization ϕ of a CHR rule 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 its internalized form $c(t_1, \ldots, x_i, \ldots, t_n)$, where x_i is a fresh variable.*
- *the new guard G' relates the original arguments of each constraint to the new ones: G' contains one $t_i \texttt{ = } \psi(x_i)$ for each internalized constraint argument.*

Example 9. The RAM simulator rule for writing a constant `Val` in memory address `Addr`:

```
prog(L,L1,const(Val),Instr,Addr) \ mem(Addr,_), prog_counter(L) <=>
    mem(Addr,Val), prog_counter(L1).
```

is internalized into:

```
prog'(X,Y,const(Val),Addr) \ mem(Addr,_), prog_counter'(Z) <=>
    ext_val(X,L), ext_val(Y,L1), ext_val(Z,L) |
    mem(Addr,Val), prog_counter(L1).
```

Definition 12 (Internalized Program). *The internalization $\phi(P)$ of a CHR program P, given by the set of rules $\overline{R}$, is defined as the internalization of each rule in $\overline{R}$, the functions ϕ and ψ, and the encoding Φ: $\phi(P) = \phi(\overline{R}) \cup \phi \cup \psi \cup \Phi$.*

3.4 Discussion

Operationally, the transformation incurs two (potential) changes in the run-time performance: (1) mapping from external to internal values, and (2) indexing (or matching) on internal rather than external values, i.e., using attribute terms rather than hash tables. While the former clearly adds performance overhead, we expect the latter to be an improvement. For an ultimate gain, (2) should outweigh (1). This is the case when there are a lot of matchings and few external calls. We expect the post-processing (Section 4) to reduce the impact of (1).

3.5 Related Work

Several programming languages define features that resemble our concept of attributed data. Firstly, the `as_chr_key` annotation is related to (primary, secondary and foreign) keys in database tables and indexing declarations in some Prolog systems. Secondly, the internalization function relates to *hash consing*—a technique, originated in Lisp, for mapping to and representing terms by unique (hash) values. Although the main aim of hash consing is to reduce memory consumption by increased sharing, it is also used to speed up equality tests. Thirdly, the *solver types* facility of Mercury [1] also imposes a dual view of constraint arguments. The internal representation type is defined by the library programmer, rather than generated automatically. Externally, the solver type is abstract, but coercion functions should be provided for external representations. Finally, a folklore optimization technique in C/C++ adds (pointer) fields to structures to compactly represent lists (and other data types) that contain them.

3.6 Future Work

In our current scheme, key declarations must be provided by a programmer. Although this approach is acceptable, we would like to investigate automatic inference of these declarations. If the programmer has no preference for an external representation, we could directly expose the abstract key type. For example, programmers often use variables and integers as identifiers in CHR constraints. The nature of a selected data type is of no concern, as long as it supports unique value creation and value comparison. By choosing the abstract key type we could eliminate unnecessary indirections of attributed variables or hash tables.

4 Post-Processing

In this section we outline a four-step rewriting procedure that statically eliminates most of the overhead introduced by the transformations proposed in Sections 2 and 3. We formulate the steps of the procedure in terms of generic flattening; their counterparts for attributed data are identical, and their counterparts for constraint symbol specialization can be easily derived.

Step 1: Unfold constraint calls according to the dynamic flattening rule.

Example 10. Say, that after flattening a rule `d(X,N) <=> N > 0, d(X,N-1)` w.r.t. the first argument of `d(X,N)` we get:

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

Applying Step 1 to the above rule yields:

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

Step 2: Apply the equivalence:

$$\psi(\overline{t}) = t \wedge \phi(t) = \overline{t'} \quad \Longleftrightarrow \quad \psi(\overline{t}) = t \wedge \overline{t} = \overline{t'}$$

Example 11. Applying Step 2 to the last rule in Example 10 yields:

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

Step 3: Flatten unifications:

$$t_1 = \psi(\overline{t_1}) \wedge t_2 = \psi(\overline{t_2}) \quad \Longrightarrow \quad t_1 = t_2 \Leftrightarrow \overline{t_1} = \overline{t_2}$$

Example 12. Consider the first rule in Table 1(a). Since the head constraints share the variable `X`, before transformation the rule should be normalized. Flattening the normalized rule yields:

```
c'(X1,X2,X3) \ c'(Y1,Y2,Y3) <=>
        unflatten(X1,X2,X3,X), unflatten(Y1,Y2,Y3,Y),
        X = Y | true.
```

By applying Step 3, we obtain:

```
c'(X1,X2,X3) \ c'(X1,X2,X3) <=>
        unflatten(X1,X2,X3,X), unflatten(X1,X2,X3,Y) | true.
```

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

Example 13. Applying Step 4 to the last rule in Example 12 yields the rule shown in line 8 of Table 1(b).

Ideally, post-processing eliminates the overhead entirely, i.e., the transformed rules operate solely on the flattened constraints, whereas the unflattened constraints may be called only by the queries, usually external to the programs. Thus, a program execution consists of two phases: (1) flattening and (2) processing of the flattened constraints. For all but the most trivial programs, we expect the run-time cost of (1) to be marginal with respect to that of (2).

5 Evaluation

We implemented our optimizations in two CHR systems on SWI-Prolog [19]: CHRd [11] (except for the attributed data) and K.U.Leuven CHR [13]. All run times, given in seconds, were measured on an Intel Pentium 4, 2.40 GHz, with 1011 MB RAM. Our benchmark suite[3] includes several common CHR programs

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

	K.U.Leuven CHR					CHRd				
	function symbols					function symbols				
benchmark	−flat −pp	+flat −pp	relative	+flat +pp	relative	−flat −pp	+flat −pp	relative	+flat +pp	relative
`gamma_prime`	3.2	6.9	215%	4.2	131%	5.4	6.0	111%	5.0	93%
`listdom`	5.7	6.2	109%	6.0	105%	6.9	5.9	86%	5.8	84%
`mergesort`	2.1	12.0	571%	1.0	48%	6.6	7.6	115%	6.8	103%
`ram_op`	8.9	11.2	126%	8.8	99%	8.3	7.8	94%	7.4	89%
`ram_prog`	3.0	2.9	97%	2.8	93%	4.4	4.0	91%	3.8	86%
`zebra`	5.5	6.8	124%	5.3	94%	6.4	7.2	113%	6.6	102%

Table 3. Run times (in sec.) for generic flattening benchmarks

– `chrg:` a pathological CHRg parser
– `dijkstra:` Dijkstra's shortest path algorithm
– `gamma_prime:` an interpreter for the Gamma chemical engine, computing prime numbers
– `listdom:` a finite domain solver for integers
– `manners:` *Miss Manners* seating assignment
– `mergesort:` mergesort algorithm
– `ram:` RAM machine interpreter, computing looping increment with constant (`ram_op`) and variable (`ram_prog`) number of constraints
– `reverse:` reversing chain of list cells
– `uf_opt:` optimal union-find algorithm
– `zebra:` the zebra puzzle.

For each optimization we consider only the relevant benchmarks, i.e. those for which the transformed programs differ from the original ones.

Whenever possible, we use (ground) mode declarations. Both flattening optimizations extend to mode declarations in a straightforward manner: a single ground argument turns into several ground arguments after flattening. In CHRd, we evaluate direct-indexed versions of `gamma_prime`, `manners`, `mergesort` and `ram`, obtained by a trivial program transformation (for details see [11]).

Generic flattening Table 3 shows the results of generic flattening in K.U.Leuven CHR and CHRd. For each 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 50% speed-up, and `gamma_prime`, with a 30% slow-down. In CHRd we observe an improvement in `gamma_prime`, `listdom` and `ram` benchmarks, 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 (e.g. success typing [9]) we (manually) achieved a relative timing of 99%. Hence, further improvement of the automated post-processing seems worthwhile.

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

Constraint symbol specialization Table 4 shows the results of constraint symbol specialization. The columns in the table have the same meaning as in Table 3.

Table 5 lists the number of static and dynamic calls to flattened constraints with respect to the number of all calls to the affected constraints. Note that the the number of dynamic calls is at least 50 % as each non-flattened constraint is flattened.

Table 4 includes two benchmarks—`zebra2` and `manners`—not reported in Table 3. The benchmark `zebra2` applies an additional round of specialization to the `zebra` program: the unoptimized entry in `zebra2` corresponds to the entry in `zebra` optimized with the (+flat,+pp) option. The original `zebra` program defines the `domain/2` constraint as:

```
domain(Var,[])        <=> fail.
domain(Var,[Val])     <=> Var = Val.
domain(Var,...), ... <=> ...
```

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

benchmark	K.U.Leuven CHR function symbols −flat −pp	+flat −pp	relative	+flat +pp	relative	CHRd function symbols −flat −pp	+flat −pp	relative	+flat +pp	relative
gamma_prime	1.5	1.7	113%	1.5	100%	4.6	4.3	93%	3.8	83%
listdom	6.0	10.4	173%	5.6	93%	7.2	9.3	129%	6.5	90%
manners	2.3	1.6	70%	1.7	74%	4.9	6.4	131%	6.1	124%
mergesort	8.5	2.3	27%	2.3	27%	6.7	5.5	82%	5.4	81%
ram_op	8.9	7.0	78%	7.3	82%	7.5	6.5	87%	6.6	88%
ram_prog	2.9	2.8	97%	2.7	93%	4.5	3.7	82%	3.6	80%
zebra	5.3	5.1	96%	5.1	96%	6.3	6.1	97%	6.1	97%
zebra2	5.1	5.1	100%	5.1	100%	6.9	6.6	96%	6.7	97%

Table 4. Run times (in sec.) for constraint symbol specialization benchmarks

	dynamic (flat / all)	static (flat / all)
gamma_prime	99.5 %	76.9 %
listdom	51.7 %	30.7 %
manners	99.2 %	90.0 %
mergesort	51.8 %	0.4 %
ram_op	50.0 %	0.0 %
ram_prog	50.0 %	0.0 %
zebra	50.0 %	0.0 %
zebra2	50.0 %	0.0 %

Table 5. Unflatten vs. flatten calls for constraint symbol specialization benchmarks

A single round of specialization treats an empty list as a special case, whereas two rounds also separate singleton lists from longer lists.

The `manners` benchmark involves constraints with constant but no partial-structure arguments, and hence it is not affected 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 behaves well 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 [17], 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 countered by the incremental overhead.

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.

	index representation				
benchmark	hash table	attr. data	relative	post-processed	relative
`chrg`	2.6	2.5	96%	2.0	77%
`dijkstra`	4.4	4.8	109%	3.9	87%
`mergesort`	2.7	5.5	204%	2.3	85%
`reverse`	2.1	2.7	129%	1.8	86%
`uf_opt`	0.3	0.4	133%	0.2	67%

Table 6. K.U.Leuven CHR run times (in sec.) for attributed data benchmarks

Generic flattening vs. constraint symbol specialization The results in Tables 3 and 4 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 our benchmarks: The partial-structure arguments in most programs uniquely identify the rule to be matched, and so, their specialization into separate constraint symbols enables immediate matching. In contrast, with generic flattening, the non-specialized constraints must be enumerated in each matching search. We expect the outcomes of generic flattening to be better for programs referencing (variable) parameters of partial-structure arguments.

Attributed data Table 6 lists the run-time results of using attributed data in K.U.Leuven CHR, measured for plain hash tables, attributed data, and attributed data with post-processed rule bodies.

The results show that the attributed data used alone negatively affects the performance (up to doubling it for `mergesort`). However, when augmented with post-processing, it improves the run time by 10% to 30% for all benchmarks.

6 Conclusion

We have presented two transformational techniques for improving the performance of CHR indexing: function symbol flattening (generic and specialized) and attributed data. A complimentary post-processing transformation compensates for potential transformation 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 to the full realization of their potential.

References

1. Ralph Becket et al. Adding constraint solving to Mercury. In *8th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2006.

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. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
4. Thom Frühwirth. As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs. In Alessandra Di Pierro and Herbert Wiklicky, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2002.
5. Christian Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. Technical Report TR-92-23, Austrian Research Institute for Artificial Intelligence, Vienna, Austria, 1992.
6. 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.
7. 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.
8. Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
9. Tobias Lindahl and Konstantinos F. Sagonas. Practical type inference based on success typings. In Annalisa Bossi and Michael J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 167–178, 2006.
10. Simon Peyton-Jones. Constructor Specialization for Haskell Programs. In *12th International Conference on Functional Programming (ICFP)*, 2007.
11. 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.
12. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
13. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, May 2004.
14. Tom Schrijvers and Thom Frühwirth. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.
15. 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.
16. 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.
17. 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.
18. David S. Warren et al. The XSB Programmer's Manual: version 2.7, vols. 1 and 2, January 2005. http://xsb.sf.net.
19. Jan Wielemaker. SWI-Prolog release 5.6.0, 2006. http://www.swi-prolog.org/.

A Appendix: Random Access Machine Simulator

```
:- use_module(library(chr)).
:- module(ram, [mem/2, prog/4, prog_counter/1]).
:- chr_constraint mem(+,+), prog(+,+,+,+), prog_counter(+).

% add value of register B to register A
prog(L,L1,add(B),A), mem(B,Y) \ mem(A,X), prog_counter(L) <=>
     Z is X+Y, mem(A,Z), prog_counter(L1).

% subtract value of register B from register A
prog(L,L1,sub(B),A), mem(B,Y) \ mem(A,X), prog_counter(L) <=>
    Z is X-Y, mem(A,Z), prog_counter(L1).

% multiply register A with value of register B
prog(L,L1,mult(B),A), mem(B,Y) \ mem(A,X), prog_counter(L) <=>
    Z is X*Y, mem(A,Z), prog_counter(L1).

% divide register A by value of register B
prog(L,L1,div(B),A), mem(B,Y) \ mem(A,X), prog_counter(L) <=>
    Z is X//Y, mem(A,Z), prog_counter(L1).

% put the value in register B in register A
prog(L,L1,move(B),A), mem(B,X) \ mem(A,_), prog_counter(L) <=>
    mem(A,X), prog_counter(L1).

% put the value in register <value in register B> in register A
prog(L,L1,i_move(B),A), mem(B,C), mem(C,X) \ mem(A,_), prog_counter(L) <=>
    mem(A,X), prog_counter(L1).

% put the value in register B in register <value in register A>
prog(L,L1,move_i(B),A), mem(B,X), mem(A,C) \ mem(C,_), prog_counter(L) <=>
    mem(C,X), prog_counter(L1).

% put the value B in register A
prog(L,L1,const(B),A) \ mem(A,_), prog_counter(L) <=> mem(A,B), prog_counter(L1).

% unconditional jump to label A
prog(L,L1,jump,A) \ prog_counter(L) <=> prog_counter(A).

% jump to label A if register R is zero, otherwise continue
prog(L,L1,cjump(R),A), mem(R,X) \ prog_counter(L) <=> X == 0 | prog_counter(A).
prog(L,L1,cjump(R),A), mem(R,X) \ prog_counter(L) <=> X =\= 0 | prog_counter(L1).

% halt
prog(L,L1,halt,_) \ prog_counter(L) <=> true.

% invalid instruction
prog_counter(L) <=> true.
```