

# Attributed Data for CHR Indexing

Beata Sarna-Starosta<sup>1</sup> and Tom Schrijvers<sup>\*2</sup>

<sup>1</sup> LogicBlox Inc., Atlanta, Georgia, USA  
bss@logicblox.com

<sup>2</sup> Department of Computer Science, K.U.Leuven, Belgium  
tom.schrijvers@cs.kuleuven.be

**Abstract.** The overhead of matching CHR rules is alleviated by constraint store indexing. Attributed variables provide an efficient means of indexing on logical variables. Existing indexing strategies for ground terms, based on hash tables, incur considerable performance overhead, especially when frequently computing hash values for large terms.

In this paper we (1) propose *attributed data*, a new data representation for ground terms inspired by attributed variables, that avoids the overhead of hash-table indexing, (2) describe program analysis and transformation techniques that make attributed data more effective, and (3) provide experimental results that establish the usefulness of our approach.

**Keywords:** Constraint Handling Rules, indexing, program transformation, term representation, attributed variables

## 1 Introduction

Constraint Handling Rules (CHR) [3] is a high-level rule-based declarative programming language, usually embedded in a host language such as Prolog or Haskell. 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, 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 [14], this source of expressiveness often leads to performance bottlenecks. Aware of this problem, CHR developers have built data structures to support efficient indexing on variables (attributed variables [7]) and ground data (search trees [8]). With [12] came the realization that  $\mathcal{O}(1)$  indexing is essential to implement CHR algorithms with optimal complexity, leading 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 [14].

In this paper we advance the research on CHR indexing with the following contributions. We present *attributed data*, an alternative to hash tables for indexing ground data that does not suffer from as much overhead (Section 3); we

---

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

describe a sequence of program post-processing steps that reduce the overhead incurred by the indexing transformations (Section 4); we propose an analysis to decide when to use the attributed data (Section 5); and we provide the experimental measurements that demonstrate the performance gain and the practical usefulness of our approach in K.U.Leuven CHR (Section 6).

Parts of this work described in Sections 3 and 4 have previously appeared at the CICLOPS 2008 symposium [9]. The implementation of the presented transformation is available at <http://www.cs.kuleuven.be/~toms/CHR/Indexing/>.

## 2 Motivation

A CHR rule is applicable when there exists a constraint substitution that matches the rule’s head. Our experience has shown that the efficiency of CHR evaluation is significantly affected by the procedure of selecting such matching head substitutions for multi-headed rules. Indeed, in Frühwirth’s analysis [14] the number of heads appears in the exponent of the worst-case time complexity formula.

CHR’s on-demand approach builds head substitutions incrementally, by first matching the active constraint, and then adding stored constraints one at a time. The purpose of *indexing* is to bring relief to the matching bottleneck. While the naive approach considers all stored constraints as candidates for the substitution, indexing aims to considerably narrow down the number of candidates to consider.

### 2.1 Attributed Variables

Efficient (constant-time) constraint store indexing has been traditionally implemented by means of *attributed variables* [6]. Attributed variables [5] provide a way to associate Prolog variables with mutable data represented as arbitrary terms. In the context of CHR, a variable’s attribute corresponds to those stored constraints, in which the variable is involved. The attribute term has the form: `attr(Index1, ..., Indexn)`, where each *Index<sub>i</sub>* is a data structure, typically a list, that contains all constraints on the variable with a particular constraint symbol. The presence of all variable’s constraints in its attribute expedites matching when the variable is shared among the constraints in the heads of the rules.

*Example 1.* Consider the rule:

$$\mathbf{a}(X), \mathbf{b}(X,Y) \implies \mathbf{write}(Y). \quad (2.1)$$

Assuming that `a/1` and `b/2` are the only declared constraints, the attribute term of a constrained variable `X` has the form `attr(Indexa, Indexb)`, where *Index<sub>a</sub>* represents all stored constraints `a(X)` and *Index<sub>b</sub>* represents all stored constraints `b(X,Y)`. Figure 1(a) depicts a constraint store containing the constraints `a(X)` and `b(X,Y)`. The single-compartment boxes denote constraints, whereas the double-compartment boxes denote variables with attributes. The dashed arrows and ovals represent the index lists *Index<sub>a</sub>* and *Index<sub>b</sub>*. Using such representation of the constraint store, given the constraint `a(X)` we can quickly find the matching constraint `b(X,Y)` by consulting the *Index<sub>b</sub>* list of variable `X`.

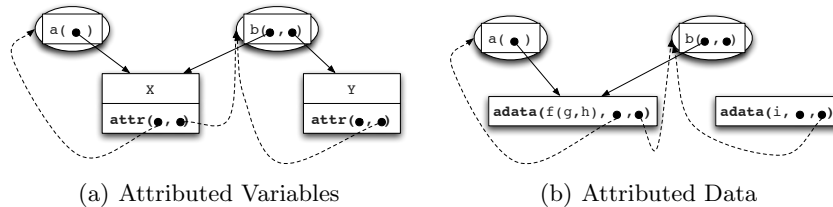


Fig. 1. Constraints  $a(X)$  and  $b(X,Y)$  with two types of indexing.

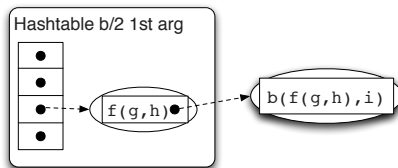


Fig. 2. Constraint  $b(f(g,h), i)$  with hash-table indexing.

## 2.2 Ground Term Pattern Matching

Clearly, attributed variables are useful only when constraints involve Prolog variables. They cannot represent ground constraints, i.e. constraints in which all arguments are ground terms.

*Example 2.* The constraints  $a(f(g,h))$  and  $b(f(g,h), i)$  match the head of rule (2.1) under the substitution  $\{f(g,h)/X, i/Y\}$ . However, as these two constraints do not share any variables, attributed variable indexing cannot be exploited to extend the partial match  $a(f(g,h))$ . Note that even if the atom  $i$  was a variable, attributed variable indexing could not be used.

To account for cases such as that described in Example 2, early implementations of CHR accumulated constraints in global, unordered lists. This representation supported  $\mathcal{O}(1)$ -time insertion of the constraints, however, constraint lookup and deletion were—in the worst case—linear in the size of the store. The introduction of hash tables [12] facilitated indexing on ground data with amortized constant-time complexity for all operations.

*Example 3.* Figure 2 depicts the hash-table index on the first argument of the constraint  $b/2$ . When an active constraint  $a(f(g,h))$  is looking for a partner constraint to apply rule (2.1), it consults this hash table; A hash value computed for the term  $f(g,h)$  yields (modulo the array size) a position in the array; The bucket list at this position is traversed until detection of the bucket for  $f(g,h)$ , which contains a linked list of all  $b/1$  constraints with the first argument having the form  $f(g,h)$  (i.e.,  $b(f(g,h), i)$  in our example).

The hash table is initialized to a small size, and dynamically expanded whenever the number of constraints exceeds a given threshold. The expansion involves

replacing the current array with an array of doubled size, and re-evaluating the hash function for all elements. Frequent evaluation of the hash function, the traversal of the bucket lists, and the resizing operation incur constant, but potentially large, overhead on processing the hash tables, which makes them, as the means for constraint indexing, considerably slower than attributed variables: for a benchmark with tight loops involving no more than two constraints, we have measured a relative slow-down of about 50%.

*Solution: Attributed Data* In order to facilitate ground-term indexing with performance characteristics of attributed variables, we propose a representation which associates ground terms directly with their constraint store indexes. We call this term representation *attributed data*. Our approach considers only variable patterns; we have addressed indexing structure patterns in prior work [10]. Also, in the rest of the presentation we assume that our approach applies to only one constraint argument at a time.

*Example 4.* Figure 1(b) shows attributed data indexing applied to Example 2. Note how little it differs from the attributed variable indexing of Figure 1(a).

Even though based on the same idea as attributed variables, attributed data cannot be implemented by a simple adaptation of the attributed variable infrastructure to the domain of ground terms because of the different ways ground terms and variables are represented by Prolog systems. Every logic variable is created exactly once, and systems, such as WAM [1], maintain its single physical representation and update it in place. As a consequence, all occurrences of the same variable observe the effects of any updates—in particular, changing it into an attributed variable—through the shared representation. On the other hand, a ground term may have multiple physical representations, created at different times, and hence changing such a term into its attributed data representation in place has no effect on its copies. This difference imposes the need to implement a new way of supporting attributed data updates. Our answer to this need—a conversion function turning any copy of a ground term into the canonical, shared attributed data representation—is described in Section 3.2.

### 3 Attributed Data

The key insight underlying our new approach to pattern matching ground terms is that the externally provided ground terms can be transformed into the internal, attributed-variable-like representation by the CHR run time.

#### 3.1 Attributed Data Representation

The internal representation  $\mathcal{I}$  of a ground term  $\mathcal{E}$  resembles an attributed variable in that it contains the ground term itself and its associated data:

$$\mathcal{I} = \text{adata}(\mathcal{E}, \text{Index}_1, \dots, \text{Index}_n)$$

where each  $Index_i$  is a constraint store index on an argument position of the term  $\mathcal{E}$  in a head constraint of some program rule.

The number and form of attributed data indexes is orthogonal to the use of attributed data, and is determined by the CHR compiler based on the form of the rule heads and the set of constraints available when looking for a matching partner. A detailed discussion of this issue can be found in Section 3.2 of [8].

In this paper, we assume that each  $Index_i$  is a flat list of constraint suspensions, with predefined operations for constraint addition and removal. The list can be updated (e.g., to replace an old index with a new one) by the destructive argument update predicate `setarg/3` implemented in most Prolog systems.

### 3.2 Conversion Functions

As discussed in Section 2.2, in order to support transforming ground terms into their attributed-data representations, we require an operation more than is more involved than the built-in predicate `put_attr/3` used to turn ordinary logic variables into attributed variables. Hence, we use a conversion function  $\phi$  which turns any copy of a ground term into the canonical, shared attributed-data representation.

**Definition 1 (Conversion Functions).** *The injective conversion function  $\phi$  maps a ground term  $t_\varepsilon$  of  $t$  onto its attributed-data representation  $t_\mathcal{I}$ :*

$$\phi(t_\varepsilon) = \begin{cases} h[t_\varepsilon] & \text{if } h[t_\varepsilon] \text{ is defined} \\ t_\mathcal{I} & \text{otherwise} \\ & \text{such that } t_\mathcal{I} = \mathbf{adata}(t_\varepsilon, \emptyset_1, \dots, \emptyset_n) \\ & \text{and } h := h[t_\varepsilon \rightarrow t_\mathcal{I}] \end{cases}$$

where  $h$  is a global hash table relating the ground terms to their known attributed-data representations. Each  $\emptyset_i$  is an empty set of constraints, one for each argument position  $j$  of each constraint symbol  $c$  that is represented by attributed data. The injective conversion function  $\psi = \phi^{-1}$  maps the attributed-data representations  $t_\mathcal{I}$  back to a copy of the ground term  $t_\varepsilon$ :

$$\psi(\mathbf{adata}(t_\varepsilon, Index_1, \dots, Index_n)) = t_\varepsilon$$

Note that  $\phi$  has an impure implementation, but a pure interface.

### 3.3 Source-to-Source Transformation

Apart from the performance aspect, the use of attributed data should be fully transparent to the programmer. Hence, to relieve the programmers from the need to explicitly call the conversion functions of Section 3.2, we provide a fully-automatable program transformation that introduces the conversions at *well-chosen* points in the program. The transformation serves two purposes: it (1) makes the programmers oblivious of the attributed-data representation, and (2) makes the attributed-data representation available for indexing to the CHR

compiler. The first purpose implies that a CHR constraint  $c/n$  should be callable with ground terms as arguments, e.g. from the interactive Prolog shell. However, this conflicts with the second purpose, which requires the arguments of  $c/n$  to have the attributed-data form for indexing.

Our solution is to split the constraint  $c/n$  into two forms. The first form,  $c/n$ , is used externally by the programmers, and its arguments are ground terms. The second form,  $c'/n$ , is used internally when applying CHR rules, and its arguments are attributed data. The external form is defined in terms of the internal form by means of the *conversion CHR rule*, that applies the conversion function  $\phi$ :

**Definition 2 (Conversion Rule).** *The conversion rule  $\Phi$  replaces ground term argument  $t_i$  in constraint term  $c/n$  with attributed-data representation  $t'_i = \phi(t_i)$ :*

$$c(t_1, \dots, t_i, \dots, t_n) \Leftrightarrow t'_i = \phi(t_i), c'(t_1, \dots, t'_i, \dots, t_n).$$

*Example 5.* Consider the constraint `arrow/2`, which in Thom Frühwirth's merge-sort program represents the numbers subject to the sort. The second argument of `arrow/2` is always ground. Thus, the conversion rule for this constraint has the form:

$$\text{arrow}(X, \text{Ne}) \Leftrightarrow \text{Ni} = \phi(\text{Ne}), \text{arrow}'(X, \text{Ni}).$$

The original CHR rules should operate on the internal constraint form  $c'/n$  rather than  $c/n$ . For this purpose, we transform each rule into a *converted* rule.

*Example 6.* Consider the following rule on the `arrow/2` constraint:

$$\text{arrow}(X, A) \setminus \text{arrow}(X, B) \Leftrightarrow A < B \mid \text{arrow}(A, B). \quad (3.2)$$

In order to benefit from attributed data indexing, the rule head should be expressed in terms of the internal constraint form `arrow'/2`:

$$\text{arrow}'(X, A) \setminus \text{arrow}'(X, B) \Leftrightarrow A < B \mid \text{arrow}(A, B).$$

However, the rule as formulated above does not work: both the guard  $A < B$  and the body `arrow(A,B)` expect  $A$  and  $B$  to be ground terms rather than attributed data. Hence, we need to introduce the conversion functions:

$$\begin{aligned} \text{arrow}'(X, A) \setminus \text{arrow}'(X, B) \Leftrightarrow A = \psi(AI), B = \psi(BI), \\ A < B \mid \text{arrow}(A, B). \end{aligned}$$

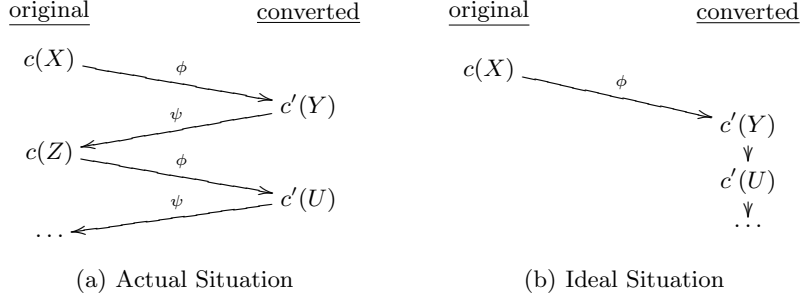
More systematically:

**Definition 3 (Converted Rule).** *The converted CHR rule is defined as:*

$$\phi(H \{\Leftrightarrow\} G \mid B) = H' \{\Leftrightarrow\} G', G \mid B$$

where

- $H'$  differs from  $H$  in that any constraint  $c(t_1, \dots, t_i, \dots, t_n)$  is replaced by its converted form  $c'(t_1, \dots, x_i, \dots, t_n)$ , where  $x_i$  is a fresh variable.



**Fig. 3.** Transitions between the original and converted constraints

- the new guard  $G'$  relates the original arguments of each constraint to the new ones:  $G'$  contains one  $t_i = \psi(x_i)$  for each converted argument.

Putting everything together:

**Definition 4 (Converted Program).** The converted CHR program  $\phi(P)$  is defined as the set of converted rules  $\bar{R}$  comprising the original program, the functions  $\phi$  and  $\psi$ , and the encoding of  $\Phi$ :

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

## 4 Post-Processing

A converted program involves repeated application of the conversion functions to alternate between the external and internal representations of the constraints, which may be a major source of overhead. We now describe a four-step rewriting procedure, which statically eliminates most of this overhead. The procedure is based on the approach taken in our prior work on partial structure indexing [10].

In a typical execution scenario (Figure 3(a)), an external value is converted into the internal representation and matched in a head of a rule, then converted back in the rule's body for calling a new constraint, converted again to match another rule, and so on. Ideally (Figure 3(b)), converted rules should operate solely on the internal representation of the arguments, whereas the external values should be used only by the queries from outside the programs. Our rewriting procedure aims to trigger this ideal scenario. Rewritten programs execute in two phases: (1) conversion of arguments' external value to the internal representations, and (2) processing of the internal representations. For all but the most trivial programs, we expect the run-time cost of (1) to be marginal with respect to the cost of (2). This conjecture is born out by the benchmarks in Section 6.

We outline the rewriting steps by applying them to an example rule.

*Example 7.* Consider normalized version of Rule (3.2), which after conversion has the form:

$$\begin{aligned} \text{arrow}'(I_1, AI) \setminus \text{arrow}'(I_2, BI) \Leftrightarrow & X = \psi(I_1), X = \psi(I_2), \\ & A = \psi(AI), B = \psi(BI), \\ & A < B \mid \text{arrow}(A, B). \end{aligned}$$

**Step 1: Make conversion explicit** unfolds constraint calls according to the conversion rules:

$$\begin{aligned} \text{arrow}'(I_1, AI) \setminus \text{arrow}'(I_2, BI) \Leftrightarrow & X = \psi(I_1), X = \psi(I_2), \\ & A = \psi(AI), B = \psi(BI), \\ & A < B \mid \text{arrow}'(\phi(A), \phi(B)). \end{aligned}$$

**Step 2: Eliminate identity conversion** applies, from left to right, the equivalence  $\forall t : \phi \circ \psi(t) = t$ :

$$\begin{aligned} \text{arrow}'(I_1, AI) \setminus \text{arrow}'(I_2, BI) \Leftrightarrow & X = \psi(I_1), X = \psi(I_2), \\ & A = \psi(AI), B = \psi(BI), \\ & A < B \mid \text{arrow}'(AI, BI). \end{aligned}$$

**Step 3: Convert external matching values to the internal representations** applies from left to right the equivalence  $\forall t_1, t_2 : \psi(t_1) = \psi(t_2) \Leftrightarrow t_1 = t_2$ :

$$\begin{aligned} \text{arrow}'(I, AI) \setminus \text{arrow}'(I, BI) \Leftrightarrow & X = \psi(I), A = \psi(AI), B = \psi(BI), \\ & A < B \mid \text{arrow}'(AI, BI). \end{aligned}$$

**Step 4: Clean up** drops unused conversion guards:

$$\begin{aligned} \text{arrow}'(I, AI) \setminus \text{arrow}'(I, BI) \Leftrightarrow & A = \psi(AI), B = \psi(BI), \\ & A < B \mid \text{arrow}'(AI, BI). \end{aligned}$$

The proposed rewriting steps are not sufficient to enforce the ideal scenario of Figure 3(b). However, as shown in Section 6, they have good practical effects.

## 5 Analysis

The attributed data framework offers an attractive alternative to hash tables. Should our approach replace hash-table indexing for ground programs? It turns out that the overhead of setting up attributed data—with the help of a hash table—may be larger than the resulting run-time gain. Our experimental evaluation<sup>3</sup> indicates that, for overall performance improvement, each attributed data index should be used more than once. In this section we consider two strategies for deciding when to represent constraint arguments as attributed data.

<sup>3</sup> See the `fib` and `fib2` benchmarks in Section 6.

## 5.1 Manual Attributed Data Declaration

For some programs, the best decision as to when to use attributed data can be made by the programmer. Thus, we introduce the `attr_data` modifier to annotate individual arguments of a constraint in the constraint’s declaration. The modifier is used in combination with a ground mode declaration `+`, and, optionally, a type declaration such as `int`.

*Example 8.* We indicate that attributed data shall be used for both integer-typed arguments of the merge-sort constraint `arrow/2` by means of the declaration:  
`:- chr_constraint arrow(+int attr_data,+int attr_data).`

## 5.2 Automatic Attributed Data Index Selection

Although easy to implement, selecting the attributed data indices by hand may be challenging, and hence is prone to performance errors. Preferably, this task should be delegated to the CHR system, which has the advantage over the programmer in that it determines on which constraint arguments to index during matching. We propose an analysis that facilitates automatic selection of indexing arguments. The analysis, based on the abstract interpretation framework for CHR [13], approximates the number of times an argument is used for indexing. Based on our experimentally determined heuristic, we then select the arguments found to be used for indexing more than once.

In order to capture argument lookup information, we need to extend CHR’s operational semantics. We assume that the built-in store is of the form  $\bigwedge X_i = t_i/l_i$  and the constraints in the queries are of the form  $c(X_1, \dots, X_n)$ . Here  $X_i$  are possibly identical variables,  $t_i$  are ground terms, and  $l_i$  are *lookup counts*. Informally, a lookup count for a term  $t$  is a natural number denoting how often  $t$  has been used to look up partner constraints. We omit the formal definition due to lack of space. Additionally, we assume that all stored constraints are ground.

Our analysis framework comprises an abstract domain of execution states, a function defining the conversion between the concrete and abstract execution states, and the abstraction of CHR’s operational semantics.

*Abstract Domain  $\Sigma_a$*  An abstract execution state has two components: (1) the program point information present in the goal in order to determine applicable abstract semantic step, and (2) the abstraction of the lookup counts  $l_i$ . A concrete state is reduced to the corresponding abstract state by the abstraction function  $\alpha_{ad}$ :

**Definition 5 (Abstraction Function).**

$$\alpha_{ad}(\langle A, S, B, T \rangle_n) = \langle \alpha_{ad-c}(A), \alpha_{ad-b}(B) \rangle$$

where the auxiliary functions  $\alpha_{ad-c}$  and  $\alpha_{ad-b}$  respectively determine the abstract goal and abstract indexing count components.

The abstraction functions for the two components are defined as:

$$\begin{aligned}\alpha_{ad-c}(c(X_i, \dots, X_n)) &= c(X_i, \dots, X_n) \\ \alpha_{ad-c}(c\#i:o) &= \alpha_{ad-c}(c):o \\ \alpha_{ad-c}(c) &= \mathbf{builtin} \quad (c \text{ built-in}) \\ \alpha_{ad-c}([c_1, \dots, c_n]) &= [\alpha_{ad-c}(c_1), \dots, \alpha_{ad-c}(c_n)]\end{aligned}$$

and  $\alpha_{ad-b}(\wedge_i X_i = t_i/l_i) = \{X_i : \alpha_l(l_i)\}$  where the abstraction of lookups is defined as  $\alpha_l(n) = \begin{cases} n & \text{if } n \leq 1 \\ * & \text{if } n > 1 \end{cases}$

That is, we reduce the lookup counts to 0, 1 or *many* (denoted by \*).

The partial order  $\prec$  and least upper bound operator  $\sqcup$  for abstract states both assume that the program point component is identical. They are defined in terms of the point-wise application of the natural abstractions of  $<$  and  $\max$  over the lookup counts.

**Definition 6 (Partial Ordering).**

$$\langle A_1, B_1 \rangle \preceq \langle A_2, B_2 \rangle = A_1 \equiv A_2 \wedge B_1 \preceq B_2$$

where  $B_1 \preceq B_2 = \forall (X:l_1) \in B_1 \exists (X:l_2) \in B_2 : l_1 \preceq l_2$ , and  $0 \prec 1 \prec *$ .

**Definition 7 (Least Upper Bound).**

$$\langle A, B_1 \rangle \sqcup \langle A, B_2 \rangle = \langle A, B_1 \sqcup B_2 \rangle$$

where  $B_1 \sqcup B_2 = \{(X : l_1 \sqcup l_2) \mid (X : l_1) \in B_1, (X : l_2) \in B_2\}$ ,

$$\text{and } l_1 \sqcup l_2 = \begin{cases} l_2 & \text{if } l_1 \prec l_2 \\ l_1 & \text{otherwise} \end{cases}$$

*Abstract Semantic Function* The abstract semantic function  $\mathcal{AS}[\mathcal{P}]$  is the abstraction of the operational semantics of CHR. The abstract semantic function exploits two pieces of information derived during the program analysis phase by the CHR compiler: which arguments of a constraint are used for indexing, and which occurrences of a constraint are passive.

The latter datum—passiveness of an occurrence  $o$  of a constraint  $c/n$ —means that the occurrence  $o$  does not fire a rule. We denote the conservative approximation derived by the CHR compiler as  $passive(c/n, o)$ . Since the CHR compiler does not generate code for passive occurrences, it is not necessary to increase lookup counts in this case, as no lookups are performed.

The former datum—which constraint arguments are used for indexing—is determined based on the rule head patterns and active occurrences of the constraints. We capture this information as  $B' = indexing(c, B, j, r)$  meaning that, after the attempt to fire rule  $r$  with active abstract constraint  $c$  at occurrence  $j$ , the abstract lookup counts change from  $B$  to  $B'$ .

**Definition 8 (Abstract Semantic Function).**

**1. AbstractSolve**

$$\mathcal{AS}[\mathcal{P}](\langle \text{builtin}, B \rangle) = \langle \square, B \rangle$$

As we have assumed that all constraints are ground, this transition does not reactivate any CHR constraints, and does not affect lookup counts.

**2/3. AbstractActivate** Let  $c$  be a CHR constraint.

$$\mathcal{AS}[\mathcal{P}](\langle c, B \rangle) = \mathcal{AS}[\mathcal{P}](\langle c:1, B \rangle)$$

This transition stores the new constraint. It does not affect lookup counts.

**4. AbstractDrop** Let  $c$  be a CHR constraint with no occurrence  $c:j$ 

$$\mathcal{AS}[\mathcal{P}](\langle c:j, B \rangle) = \langle \square, B \rangle$$

This transition deactivates the active constraint. It does not affect lookup counts.

**5a. AbstractSimplify** Let  $d$  be the  $j^{\text{th}}$  occurrence of  $c$  in a (renamed apart) rule  $r \in \mathcal{P}$  with  $\neg \text{passive}(c, j)$ :

$$r @ H'_1 \setminus H'_2, d_{[j]}, H'_3 \iff g \mid C$$

$$\text{then } \mathcal{AS}[\mathcal{P}](\langle c:j, B \rangle) = \mathcal{AS}[\mathcal{P}](s_1) \sqcup \mathcal{AS}[\mathcal{P}](s_2)$$

where

$$\begin{cases} B' = \text{indexing}(c, B, j, r) \\ s_1 = \langle \alpha_{ad-c}(C), B' \rangle \\ s_2 = \langle c:j+1, B' \rangle \end{cases}$$

**6a. AbstractPropagate** Let  $d$  be the  $j^{\text{th}}$  occurrence of  $c$  in a (renamed apart) rule  $r \in \mathcal{P}$  with  $\neg \text{passive}(c, j)$ :

$$r @ H'_1, d_{[j]}, H'_2 \setminus H'_3 \iff g \mid C$$

$$\text{then } \mathcal{AS}[\mathcal{P}](\langle c:j, B \rangle) = \mathcal{AS}[\mathcal{P}](\langle c:j+1, B' \rangle)$$

where  $B' = \text{lfp}(f, \langle B \rangle)$ .

The auxiliary function  $f$  is defined as:

$$f(B_0) = B_2$$

where

$$\begin{cases} B_1 = \text{indexing}(c, B_0, j, r) \\ \langle \square, B_2 \rangle = \mathcal{AS}[\mathcal{P}](\langle \alpha_{ad-c}(C), B_1 \rangle) \end{cases}$$

**5b/6b. AbstractPassive** Let  $d$  be the  $j^{\text{th}}$  occurrence of  $c$  in a (renamed apart) rule  $r \in \mathcal{P}$  with  $\text{passive}(c, j)$ , then

$$\mathcal{AS}[\mathcal{P}](\langle c:j, B \rangle) = \mathcal{AS}[\mathcal{P}](\langle c:j+1, B \rangle)$$

$$\begin{array}{ll}
f(N) \setminus f(N) \Leftrightarrow \text{true}. & f(N), f(N) \Leftrightarrow f(N). \\
f(0) \Leftrightarrow \text{true}. & f(0) \Leftrightarrow \text{true}. \\
f(N) \Rightarrow M \text{ is } N - 1, f(M). & f(N) \Rightarrow M \text{ is } N - 1, f(M).
\end{array}$$

(a) (b)

**Fig. 4.** Programs showing worse (a) and better (b) performance with attributed data

**7. AbstractGoal**

$$\mathcal{AS}[\mathcal{P}](\langle [c_1, \dots, c_n], B_0 \rangle) = \langle \square, B_n \rangle$$

where for  $i \in 1..n$

$$\mathcal{AS}[\mathcal{P}](\langle c_i, \text{proj}(B_{i-1}, c_i) \rangle) = \langle \square, B'_i \rangle$$

and

$$\text{proj}(B, c(X_1, \dots, X_n)) = \{(X_i : l_i) \in B \mid i \in 1..n\}$$

$$\text{ext}(B_x, B_y) = B_x \cup \{(X : l) \in B_y \mid \neg \exists l' : (X \mapsto l') \in B_x\}$$

$$B_i = \text{ext}(B'_i, B_{i-1}) \quad (i \in 1..n)$$

This transition sequences a list (conjunction) of goals.

*Example* Consider the two programs in Fig. 4. The use of attributed data slows down the program in Fig. 4(a), but improves the performance of the program in Fig 4(b). The reason for this difference is that in the program in Fig. 4(a) each argument is used for indexed lookup only once, whereas in the program in Fig. 4(b) some arguments are used multiple times.

Consider now the analysis for the program in Fig. 4(a):

$$\begin{aligned}
& \mathcal{AS}[\mathcal{P}](\langle f(N), \{N : 0\} \rangle) \\
&= \mathcal{AS}[\mathcal{P}](\langle f(N) : 1, \{N : 0\} \rangle) \\
&= \mathcal{AS}[\mathcal{P}](\langle f(N) : 2, \{N : 1\} \rangle) \sqcup \mathcal{AS}[\mathcal{P}](\langle \text{builtin}, \{N : 1\} \rangle) \\
&= \langle \square, \{N : 1\} \rangle \sqcup \langle \square, \{N : 1\} \rangle \\
&= \langle \square, \{N : 1\} \rangle
\end{aligned}$$

where (since the second occurrence is passive)

$$\begin{aligned}
& \mathcal{AS}[\mathcal{P}](\langle f(N) : 2, \{N : 1\} \rangle) \\
&= \mathcal{AS}[\mathcal{P}](\langle f(N) : 3, \{N : 1\} \rangle) \\
&= \mathcal{AS}[\mathcal{P}](\langle f(N) : 4, \{N : 1\} \rangle) \sqcup \mathcal{AS}[\mathcal{P}](\langle \text{builtin}, \{N : 1\} \rangle) \\
&= \langle \square, \{N : 1\} \rangle \sqcup \langle \square, \{N : 1\} \rangle \\
&= \langle \square, \{N : 1\} \rangle
\end{aligned}$$

and

$$\begin{aligned}
& \mathcal{AS}[\mathcal{P}](\langle f(N) : 4, \{N : 1\} \rangle) \\
&= \mathcal{AS}[\mathcal{P}](\langle f(N) : 5, \{N : 1\} \rangle) \sqcup \mathcal{AS}[\mathcal{P}](\langle [\text{builtin}, f(M)], \{N : 1\} \rangle) \\
&= \langle \square, \{N : 1\} \rangle \sqcup \langle \square, \{N : 1\} \rangle \\
&= \langle \square, \{N : 1\} \rangle
\end{aligned}$$

Hence, the lookup count never exceeds 1, and thus our heuristic indicates that attributed data should not be used. If, instead, we consider the program in Fig. 4(b), the main derivation becomes:

$$\begin{aligned}
\mathcal{AS}[\mathcal{P}](\langle f(N), \{N : 0\} \rangle) & \\
= \mathcal{AS}[\mathcal{P}](\langle f(N) : 1, \{N : 0\} \rangle) & \\
= \mathcal{AS}[\mathcal{P}](\langle f(N) : 2, \{N : 1\} \rangle) \sqcup \mathcal{AS}[\mathcal{P}](\langle f(N), \{N : 1\} \rangle) & \\
= \langle \square, \{N : 1\} \rangle \sqcup \langle \square, \{N : * \} \rangle & \\
= \langle \square, \{N : * \} \rangle &
\end{aligned}$$

because the first rule now involves a recursive call. Now the lookup count is \* and using attributed data is recommended.

## 6 Evaluation

We implemented our approach in K.U.Leuven CHR [11] on SWI-Prolog [15], and tested it on several benchmarks<sup>4</sup>. All run times, given in seconds, as well as relative to the original for the transformed versions, were measured on an Intel Pentium 4, 2.00 GHz, with 512 MB RAM.

Our implementation of attributed data consists of two components: (1) a pre-processor, which transforms a CHR program with key annotations into its converted form, and (2) the actual code generator of the CHR compiler, which generates attributed data indexing instructions and emits definitions for the conversion functions. The function  $\phi$  is implemented in terms of the hash tables used for hash-table indexing. The function  $\psi$  is always called as  $B = \psi(A)$ ; we inline it as  $A = \text{adata}(B, -, \dots, -)$  at each call site.

Table 1 lists the run-time results of exploiting attributed data in K.U.Leuven CHR, measured for plain hash tables, plain attributed data instead of the hash table indexes, and attributed data with post-processed rule bodies.

The block of the first seven benchmarks clearly demonstrates the positive effects of our approach. Although, the attributed data used on its own causes a slow-down (up to almost 50% for `mergesort`), when augmented with post-processing, it improves the run time by 20% to 50%.

The block of the last four benchmarks shows two programs for which the use of attributed data impairs the performance. The first program, `fib`, involves one hash-table lookup per new constraint. Hence, the attributed data manipulation is pure overhead (25%). For this reason, our analysis from Section 5 flags the program as unsuitable for attributed data; it does not flag any of the other benchmarks.

The second benchmark, `fib2`, replaces the simpagation rule of `fib`:

$$\text{fib}(N, F1) \setminus \text{fib}(N, F2) \Leftrightarrow F1 = F2.$$

with the simplification rule:

$$\frac{}{\text{fib}(N, F1), \text{fib}(N, F2) \Leftrightarrow F1 = F2, \text{fib}(N, F1).}$$

<sup>4</sup> available at <http://www.cs.kuleuven.ac.be/~toms/CHR/Indexing/>

benchmark	index representation				
	hash table	attr. data	relative	post-processed	relative
<code>chrg</code>	2.17	2.10	96.8%	1.58	72.8 %
<code>flat_ram</code>	4.69	4.31	91.9%	2.50	53.3%
<code>mergesort</code>	3.33	4.89	146.8%	1.85	55.6 %
<code>reverse</code>	2.55	3.25	127.4%	1.92	75.3%
<code>uf_opt</code>	0.34	0.38	111.8%	0.25	73.5%
<code>turing</code>	1.50	1.31	87.3%	1.19	79.3%
<code>wfs</code>	1.32	0.88	66.7%	0.85	64.4%
<code>fib</code>	1.24	1.53	123.4%	1.52	122.6%
<code>fib2</code>	1.61	1.30	80.7%	1.05	65.2%
<code>dijkstra</code>	2.26	4.52	200.0%	3.53	156.2%
<code>dijkstra2</code>	1.54	2.20	142.9%	1.25	81.2 %

**Table 1.** K.U.Leuven CHR run times (in sec.) for attributed data benchmarks

This modification causes the parameter `N` to be reused in the new call in the rule’s body. As a consequence, attributed data requires only one hash-table lookup for every two new constraints, which results in a noticeable speed-up.

The second slow-down, in `dijkstra`, results from a limitation of our current implementation, which does not allow multi-argument indices involving attributed data arguments. Thus, we were required to replace a two-argument hash table index by a single-argument attributed data index. For this benchmark, the two-argument index turns out to be more selective and more efficient. However, the use of *symbol specialization* [10] allowed to entirely eliminate the second matching argument from this benchmark, and thus obtain a speed-up in the resulting program `dijkstra2`.

## 7 Discussion and Related Work

We have presented attributed data—a new term representation that improves the efficiency of CHR indexing at a high level—and a complementary post-processing procedure that reduces the overhead of conversions between the new representation and the standard representation of Prolog terms. We have implemented our approach for the K.U.Leuven CHR system on SWI-Prolog. The evaluation on a set of benchmarks shows that attributed data enables performance improvement, and that post-processing is critical to fully realize this potential.

Several programming languages define features that resemble the concept of attributed data. The conversion function  $\phi$  relates to *hash consing*—a technique, originated in Lisp, for mapping terms to, and representing them 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.

Like attributed data, Mercury’s *solver types* [2] impose 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<sup>5</sup> optimization technique in C/C++ adds (pointer) fields to structures to concisely represent lists (and other data types) that contain them.

**Acknowledgements** The authors thank Leslie De Koninck, Bart Demoen and Jon Sneyers for their helpful comments on the paper.

## References

1. Hassan Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
2. Ralph Becket et al. Adding constraint solving to Mercury. In *8th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2006.
3. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
4. Thom Frühwirth. As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs. *Electronic Notes in Theoretical Computer Science*, 59(3), 2002.
5. Christian Holzbaaur. 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 Holzbaaur and Thom Frühwirth. Compiling Constraint Handling Rules into Prolog with attributed variables. In G. Nadathur, editor, *PPDP '99*, volume 1702 of *LNCS*, pages 117–133, Paris, France, 1999. Springer.
7. Christian Holzbaaur 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.
8. Christian Holzbaaur, 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.
9. Beata Sarna-Starosta and Tom Schrijvers. An efficient term representation for CHR indexing. In M. Carro and B. Demoen, editors, *Proceedings of CICLOPS 2008*, pages 172–186, 2008.
10. Beata Sarna-Starosta and Tom Schrijvers. Transformation-based indexing techniques for constraint handling rules. In T. Schrijvers, F. Raiser, and T. Frühwirth, editors, *CHR '08*, RISC Report Series 08-10, University of Linz, Austria, pages 3–18, Hagenberg, Austria, July 2008.
11. 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.
12. Tom Schrijvers and Thom Frühwirth. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.
13. Tom Schrijvers, Peter J. Stuckey, and Gregory J. Duck. Abstract interpretation for Constraint Handling Rules. In P. Barahona and A.P. Felty, editors, *PPDP '05*, pages 218–229, Lisbon, Portugal, July 2005. ACM Press.
14. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(12), 2009.
15. Jan Wielemaker. SWI-Prolog release 5.6.0, 2006. <http://www.swi-prolog.org/>.

---

<sup>5</sup> e.g. the Linux kernel linked list: <http://isis.poly.edu/kulesh/stuff/src/klist/>