An Efficient Term Representation for CHR Indexing

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

[1] LogicBlox Inc., Atlanta, Georgia, USA
bss@logicblox.com
[2] Department of Computer Science, K.U.Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract. The overhead of matching CHR's multi-headed rules is alleviated by constraint store indexing. The attributed variable interface provides efficient means of indexing on logical variables. Current state-of-the-art indexing strategies for ground terms use hash tables. However, the hash tables incur considerable performance overhead, especially when frequently computing hash values for large terms.
We propose a high-level approach which improves the efficiency of ground term indexing. In this approach, we introduce a new data representation for ground terms, inspired by attributed variables, that avoids the overhead of hash-table indexing. The experimental evaluation establishes the usefulness of our representation, but indicates a high cost of mapping between this representation and Prolog's standard terms. Thus, we reuse previously implemented post-processing program transformations to compensate for this overhead. We compare our approach with the current state of the art, and give measurements of its effectiveness in the K.U.Leuven CHR system.
keywords: Constraint Handling Rules, indexing, program transformation, term representation, attributed variables

1 Introduction

Constraint Handling Rules (CHR) [4] 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 [14]. 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 [5], this source of expressiveness often leads to performance bottlenecks. This effect is

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

borne out by the approximative complexity formula of [5], where the multiplicity of rule's head appears in the exponent.

Aware of this problem, CHR developers have built data structures supporting 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 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 [13]. CHRd [9] 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 a high-level approach to efficient indexing on ground terms. Specifically, we make the following contributions:

- propose an alternative to hash tables for indexing ground data, which does not suffer from amortization-related overhead (Section 3),
- reuse previously developed post-processing program transformations [10] to reduce the disadvantages of the new approach (Section 4),
- demonstrate the measurements of the usefulness of the presented technique in K.U.Leuven CHR system (Section 5), and
- provide an implementation of the presented techniques (available online at `http://www.cs.kuleuven.be/~toms/CHR/AttributedData/`).

The presentation begins with an overview of CHR and indexing in Section 2. Section 3 describes our new representation for ground terms, the conversions between the new representation and Prolog terms, and the program transformation for introducing these conversions. Section 4 discusses the overhead of the conversions, and treats it with the post-processing program transformation. Section 5 presents the experimental evaluation of the proposed transformations, Section 6 relates our approach to other work, and Section 7 concludes.

2 Preliminaries

CHR is a language of multi-headed rewriting rules that is particularly well-suited for specifying custom constraint solvers at a high-level. A CHR program prescribes the transformations of a *constraint store* (a collection of *user-defined* constraints), based on the *built-in* constraints of the host language. For the purpose of this paper we consider Prolog as the host language; the built-in constraints are Prolog predicates and equations (unifications) of Herbrand terms.

CHR Syntax. A CHR program is a finite set of rules of the form:

label `@` *Head* `?=>` *Guard* `|` *Body*

The *label* names the rule and may be omitted along with the trailing `@`. The arrow `?=>` denotes the kind of transformation a rule defines, and may be either `<=>` or `==>` (we use `?=>` as a shorthand notation for both forms). There are

```
:- chr_constraint arrow/2, merge/2.

pick @ merge(N,A), merge(N,B) <=> A<B | M is N+1, arrow(A,B), merge(M,A).
join @ arrow(X,A) \ arrow(X,B) <=> A<B | arrow(A,B).
```

Table 1. An example CHR program encoding the merge-sort algorithm

three kinds of CHR rules. The most general are *simpagation* rules of the form: $H_1 \setminus H_2$ `<=>` $G \mid B$, where H_1 and H_2 are sequences of user-defined constraint terms (the d constraint terms. A rule specifies that when constraints in the store match H_1 and H_2 and the guard G holds, the constraints that match H_2 can be *replaced* by the corresponding constraints in B. The literal `true` represents an empty sequence of constraint terms. The guard part, $G \mid$, may be omitted when G is empty.

A *simplification* rule, which has the form: H_2 `<=>` $G \mid B$, specifies that when the stored constraints match the head, and the guard holds, the head constraints can be replaced by the body constraints. A rule of this form can be represented by a simpagation rule: `true` $\setminus H_2$ `<=>` $G \mid B$.

A *propagation* rule, which has the form: H_1 `==>` $G \mid B$, specifies that when the stored constraints match the head, and the guard holds, the body constraints can be *added* to the store. A rule of this form can be represented by a simpagation rule: $H_1 \setminus$ `true` `<=>` $G \mid B$.

Example 1. Consider the CHR program in Table 1. The simplification rule `pick` states that each pair of stored constraints matching `merge(N,A)` and `merge(N,B)` such that `A<B` should be replaced with the pair of constraints `arrow(A,B)` and `merge(M,A)` where `M=N+1`. The simpagation rule `join` states that, in the presence of two constraints `arrow(X,A)` and `arrow(X,B)` such that `A<B`, the constraint `arrow(X,B)` should be replaced by `arrow(A,B)`.

The program, by Thom Frühwirth, encodes the classical merge-sort algorithm. The algorithm is executed in the bottom-up fashion: the `pick` rule selects two sublists of elements at the same level for merging, whereas the `join` rule merges two selected sublists together.

CHR Semantics. CHR has a well-defined declarative as well as operational semantics [4, 3, 9]. The declarative interpretation of a CHR program is given by the set of universally quantified formulas corresponding to the CHR rules, and an underlying consistent constraint theory.

The original operational interpretation of a CHR program [4] is a non-deterministic transition system. The transitions are made when an unsolved constraint is added to the store, or by firing any applicable program rule.

The refined operational semantics [3][3] defines a more deterministic transition system, specifying, among others, that rules are tried in textual order. An ex-

[3] followed by most CHR implementations

	$\langle[\underline{\texttt{merge}(1,80)},\texttt{merge}(1,40),\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \emptyset\rangle$	(1)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(1,40)},\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \{\texttt{merge}(1,80)\}\rangle$	(2)
$\rightarrowtail_{\texttt{pick}}$	$\langle[\underline{\texttt{arrow}(40,80)},\texttt{merge}(2,40),\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \emptyset\rangle$	(3)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(2,40)},\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \{\texttt{arrow}(40,80)\}\rangle$	(4)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(1,50)},\texttt{merge}(1,70)],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40)\}\rangle$	(5)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(1,70)}],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40),\texttt{merge}(1,50)\}\rangle$	(6)
$\rightarrowtail_{\texttt{pick}}$	$\langle[\underline{\texttt{arrow}(50,70)},\texttt{merge}(2,50)],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40)\}\rangle$	(7)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(2,50)}],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40),\texttt{arrow}(50,70)\}\rangle$	(8)
$\rightarrowtail_{\texttt{pick}}$	$\langle[\underline{\texttt{arrow}(40,50)},\texttt{merge}(3,40)],\ \{\texttt{arrow}(40,80),\texttt{arrow}(50,70)\}\rangle$	(9)
$\rightarrowtail_{\texttt{join}}$	$\langle[\underline{\texttt{arrow}(50,80)},\texttt{merge}(3,40)],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50)\}\rangle$	(10)
$\rightarrowtail_{\texttt{join}}$	$\langle[\underline{\texttt{arrow}(70,80)},\texttt{merge}(3,40)],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50)\}\rangle$	(11)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(3,40)}],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50),\texttt{arrow}(70,80)\}\rangle$	(12)
$\rightarrowtail^*$	$\langle[\,],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50),\texttt{arrow}(70,80),\texttt{merge}(3,40)\}\rangle$	(13)

Table 2. An example derivation for the merge-sort program

tended version of the same transition system is used by the set-based operational semantics [9].

Example 2. The merge-sort program from Example 1 constructs a sorted list from a collection of sorted sublists. The head of a sorted sublist is given by means of a `merge(L,N)` constraint, where $2^{\texttt{L}-1}$ is the sublist's length and `N` is the sublist's first element. The `arrow/2` constraints model the edges between the nodes of a sorted sublist.

Table 2 outlines an example derivation for the program under the refined operational semantics. For the clarity of the presentation, the irrelevant transitions and the parts of the execution state not affected by the derivation have been omitted. For each presented derivation step, the table shows the current goal, with the active constraint underlined, and the contents of the constraint store. In the initial goal each sublist consists of a single element, and hence all sublists have the same length (equal to 2^{1-1}). The nodes are collected in the constraint store until two same-length nodes match the head of the `pick` rule. The rule transforms such two nodes into a sorted sublist and increments the length. The `join` rule sorts the nodes within each individual sublist. At the end of the derivation, the constraint store contains a collection of `arrow/2` constraints representing the sorted list.

CHR Indexing. Indexing in CHR facilitates retrieval of suspended constraints to match partner constrains in rule heads. Efficient (constant-time) constraint store indexing has been traditionally implemented by means of attributed variables [6], which 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(`$Index_1$`,...,`$Index_n$`)`, where each $Index_i$ 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.

Constraint store indexing based on attributed variables is efficient, but not always practical—for example, it is not feasible for ground constraints, in which no variables are involved. For that reason, in addition to using variable attributes, 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 store size. The introduction of hash tables [12] facilitated indexing on ground data, yielding amortized constant time complexity for all operations. A hash-table constraint store is defined as an array, in which every element represents a set of colliding constraints (i.e., constraints that evaluate to the same value of the hash function). The table is initialized to a small size, and dynamically expanded whenever the number of constraints exceeds 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 number of colliding constraints, and the resizing operation incur constant, but potentially considerable, overhead on processing the hash tables, which makes them altogether slower than attributed variables.

3 Attributed Data

In this section, we consider constraints containing arguments that are ground terms. If such arguments are matched against each other in rule heads, then constant-time matching is realized by means of a hash-table index on these ground arguments.

As an alternative to hash tables, we propose *attributed data*, which provide $\mathcal{O}(1)$ indexing with constant factors closer to those of attributed variables. The key insight underlying our approach is that the CHR run time can internally use an attributed-variable–like representation for externally provided ground terms.

3.1 Indexing Key Declarations

In our approach, ground arguments of the constraints that are matched against each other in rule heads—and hence serve as indexing keys—are internally represented using a special data type *key type*. The programmers indicate such constraint arguments using the new annotation '`as_chr_key`'. The specifier '`+`*type* `as_chr_key` *keytype*' states that the argument in question is ground (`+`), and uses *type* as its external representation and *keytype* as its internal representation. The abstract key type for a given indexing key in a CHR program is generated automatically by the CHR compiler based on the occurrence pattern of that key in the heads of the program rules.

Example 3. In the merge-sort program from Example 1, since the second argument of `merge/2` as well as both arguments of `arrow/2` are always ground and correspond to the numbers being sorted, a programmer may decide to capture all of them using the same internal representation. Denoted as `elem_key`, this representation is declared as follows:

```
:- chr_constraint
     merge(+int,+int as_chr_key elem_key),
     arrow(+int as_chr_key elem_key,+int as_chr_key elem_key).
```

3.2 Indexing Key Representation

The instances of the new data type resemble the attribute terms of attributed variables. The key type representation, however, does not include the actual variables to avoid unnecessary indirection.

The internal representation $\mathcal{I}$ of a ground indexing key in a CHR program is a term:

$$\mathcal{I} = \texttt{key}(\mathcal{E}, \mathit{Index}_1, \ldots, \mathit{Index}_n)$$

where each Index_i is an index on an argument position of that key in a head constraint of some program rule, and $\mathcal{E}$ is the key's original external value.

The number and form of the indexes in the internal representation for a particular key 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 subset of head constraints available when looking for a matching partner. For a detailed discussion of this issue we refer the reader to Section 3.2 of [8].

For the purpose of this paper we assume that the default representation of 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.

Example 4. Since two of the three argument positions declared as indexing keys in Example 3 are never used to retrieve partner constraints, the CHR compiler decides that only one index—for the first argument of `arrow/2`— will be exploited to speed-up the matching of the `join` rule.

Hence, given the number 80 as the external representation, the corresponding internal representation, assuming that the single index is empty, is `key(80,[])`.

Definition 1 (Conversion Functions). *For a ground indexing key type t, the injective* conversion function ϕ *maps an external value t_ε of t onto the internal representation $t_\mathcal{I}$ of t:*

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

where h is a global hash table relating the external values of ground indexing keys to their known internal representations. The injective conversion function $\psi = \phi^{-1}$ *maps the internal representations $t_\mathcal{I}$ onto the external values $t_\mathcal{E}$:*

$$\psi(\texttt{key}(t_\mathcal{E}, Index_1, \ldots, Index_n)) = t_\mathcal{E}$$

Example 5. The following internal representations are initially computed for the list of numbers given in the query in Example 1: ϕ`(80) = key(80,[])`, ϕ`(40) = key(40,[])`, ϕ`(50) = key(50,[])`, ϕ`(70) = key(70,[])`. Figure 1 depicts these internal representations, as well as the example hash table (with a linked list of buckets) underlying ϕ.

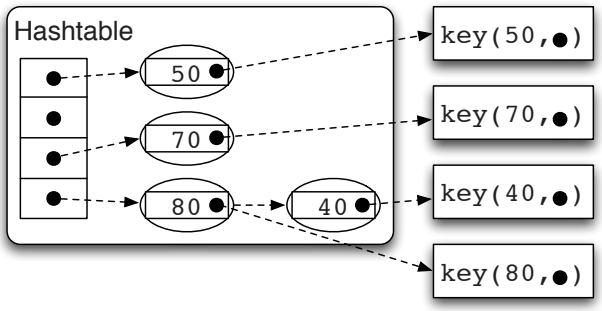


Fig. 1. Internal representation of 80, 40, 50 and 70, and hash table for ϕ.

3.3 Source-to-Source Transformation

In this section we define a source-to-source transformation for mapping between the external and internal representations of ground indexing keys. Without loss of generality, we only formalize the transformation for a single key type. Multiple keys are easily supported by repeated application of the transformation, while making sure to avoid name clashes.

The conversion rule Φ applies the conversion function ϕ at run time:

Definition 2 (Conversion Rule). *The* conversion rule Φ *replaces the external value of a ground indexing key argument t_i in a constraint term c/n with its internal representation $t' = \phi(t)$:*

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

Example 6. The dynamic conversion rule for the `arrow/2` constraint from the merge-sort program is of the form:

`arrow(X,Ne) <=> Ni = `ϕ`(Ne), arrow'(X,Ni).`

Definition 3 (Converted Rule). *The* converted 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 converted 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 `=` $\psi(x_i)$ for each converted argument.*

Example 7. The converted `join` rule from the merge-sort program is of the form:

```
join' @ arrow'(X1,AI) \ arrow'(X2,BI) <=>
            X = ψ(X1), X = ψ(X2),
            A = ψ(AI), B = ψ(BI), A<B |
            arrow(A,B).
```

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

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

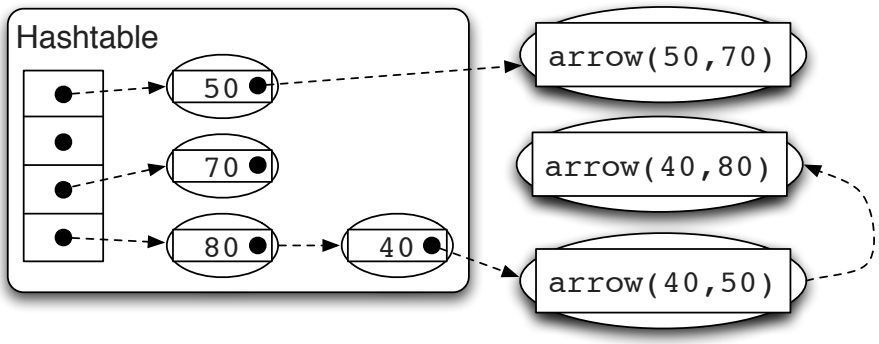
3.4 Elaborated Example

Consider the merge-sort program, and the query

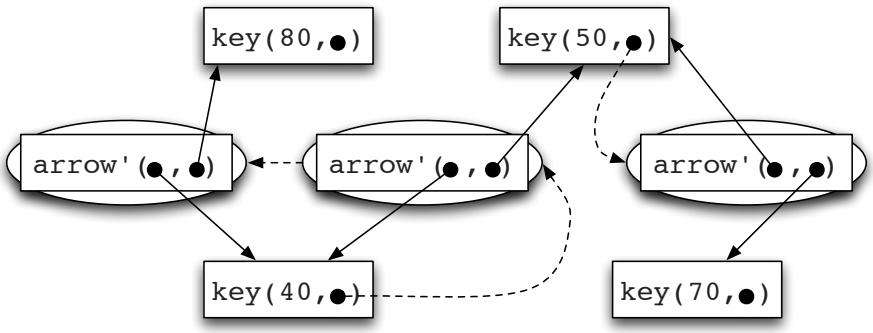
```
?- merge(1,80), merge(1,40), merge(1,50), merge(1,70).
```

evaluated as shown in Table 2. In the execution state (9), `arrow(40,50)` is the active constraint, whereas `arrow(40,80)` and `arrow(50,70)` are suspended in the constraint store. In the following derivation step, the `join` rule is triggered, and `arrow(40,80)` is retrieved from the store to serve as the partner constraint to match the rule's head.

Figure 2 illustrates two instances of this situation: (a) with indexing based on a hash table, and (b) with indexing based on attributed data. In the former case, retrieving the required partner constraint involves hashing the number 40 into the table, traversing the bucket list to find the appropriate bucket, and locating the constraint within the bucket. In the latter case, the internal representation `key(40,L)` provides direct access to the linked list containing `arrow(40,80)`. Clearly, using attributed data avoids the overhead of hashing into the table and of traversing the bucket list.



(a) Hashtable

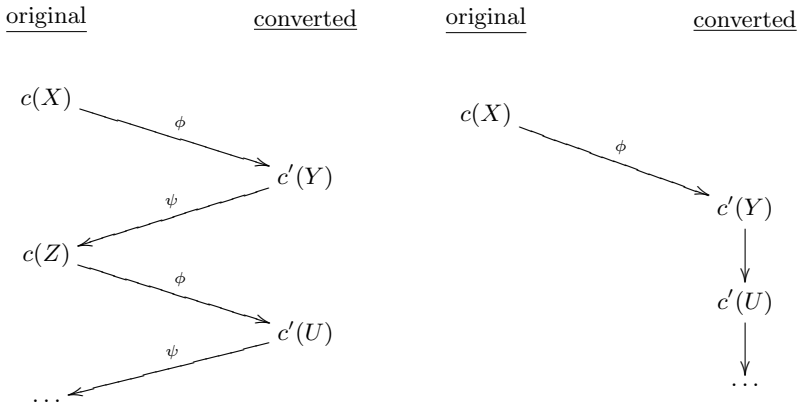


(b) Attributed Data

Fig. 2. Situation during the merge-sorting of 80, 40, 50 and 70.

4 Post-Processing

The experimental results in Section 5 indicate that the performance improvement obtained by better indexing is offset, or in some cases even surpassed, by the run-time overhead of applying the conversion functions. In this section we outline a transformation that statically eliminates most of this overhead; it was previously used to avoid similar performance issues in other transformations based on term flattening [10]. The effectiveness of the transformation is borne out by the benchmarks in Section 5.



(a) Actual Situation (b) Ideal Situation

Fig. 3. Transitions between the original and converted constraints

Alternating the conversions between the internal and external argument representations is a major source of runtime overhead. In a typical scenario (Figure 3(a)), an external value is converted into the internal representation and matched in a head of a rule, then it is converted back in the rule's body for calling a new constraint, converted again to match another rule, and so on. To avoid this overhead, the transformed rules should operate solely on the internal representation of the arguments, whereas the external values should be used only by the queries external to the programs. We propose a four-step rewriting procedure that aims to trigger this ideal scenario (Figure 3(b)). Execution of a program enhanced with the procedure consists of two phases:

(1) conversion of an argument's external value to the internal representation, and
(2) processing of the internal representation.

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

Our rewriting procedure comprises the following steps.

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

Example 8. Consider the `join` rule from the program in Table 1:

```
arrow(X,A) \ arrow(X,B) <=> A < B | arrow(A,B).
```

After conversion, the rule has the form:

```
arrow'(XI1,AI) \ arrow'(XI2,BI) <=>
     X = ψ(XI1), X = ψ(XI2),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow(A,B).
```

By applying Step 1 to the above rule we obtain:

```
arrow'(XI1,AI) \ arrow'(XI2,BI) <=>
     X = ψ(XI1), X = ψ(XI2),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(ϕ(A),ϕ(B)).
```

We refer the reader to the work of Tacchella et al. [15] for the formal definition and correctness proof of unfolding of CHR rules.

Step 2: Eliminate identity conversion.
Apply the following equation from left to right:

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

The transformation is valid based on the property that ϕ is the inverse of ψ.

Example 9. Applying Step 2 to the last rule in Example 8 yields:

```
arrow'(XI1,AI) \ arrow'(XI2,BI) <=>
     X = ψ(XI1), X = ψ(XI2),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(AI,BI).
```

Step 3: Convert external values of matchings to the internal representations.

Apply the equivalence from left to right:

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

The transformation is valid based on the property that ψ is injective.

Example 10. Applying Step 3 to `X` in the rule from Example 9 yields:

```
arrow'(XI,AI) \ arrow'(XI,BI) <=>
     X = ψ(XI),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(AI,BI).
```

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

Example 11. Applying Step 4 to the rule in Example 10 yields:

```
arrow'(XI,AI) \ arrow'(XI,BI) <=>
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(AI,BI).
```

In general, these rewriting steps are not sufficient to enforce the ideal scenario of Figure 3(b). However, as the results in Section 5 show, they have good practical effects.

5 Evaluation

We implemented our approach in K.U.Leuven CHR [11] on SWI-Prolog [16]. The implementation 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. Note that the pre-processor performs the transformations for all keys simultaneously rather than sequentially. In doing so, it avoids generating multiple intermediate conversion rules for constraints involving more than one key type.

We have evaluated our implementation on several standard CHR benchmarks. All run times, given in seconds for the original programs and relative to the original for the transformed versions, were measured on a MacBook Pro Intel Core Duo 1.83 GHz, with 1 GB RAM. Our benchmark suite includes the following programs:

- `chrg`, a CHRg parser with an exponential number of parses
- `dijkstra`, Dijkstra's shortest path algorithm
- `fib`, computation of fibonacci numbers, with effective memoing
- `fib2`, computation of fibonacci numbers, with ineffective memoing
- `mergesort`, mergesort algorithm
- `flat_ram`, RAM machine interpreter, flattened by symbol specialization [10]
- `reverse`, reversing chain of list cells
- `turing`, Turing machine simulator, running the *copy* program
- `uf_opt`, optimal union-find algorithm

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

Table 3. K.U.Leuven CHR run times (in sec.) for attributed data benchmarks

– `wfs`, well-founded semantics algorithm.

For each benchmark, we have manually added the `as_chr_key` annotations for the argument positions according to the following prioritized guidelines:

1. If two head constraints share more than one variable, we do not annotate the corresponding argument positions of those variables, because they are better served by multi-argument indexing. For instance, consider a rule head of the form `c(X,Y), d(Y,X)`. Although indexing on a single argument, i.e., using either `X` or `Y`, does work, indexing on the combination of both arguments is usually more efficient.
2. If two head constraints share exactly one variable, we annotate the corresponding argument positions of that variable with the same key.
3. If no variables are shared, no index is required.

Most benchmarks require a single key type. The exceptions are `ram_flat` and `turing`, each using two key spaces to represent instruction labels/states and data addresses, and `wfs` with separate key spaces for atoms and clause identifiers.

Table 3 lists the run-time results of exploiting attributed data in K.U.Leuven CHR, measured for plain hash tables, plain attributed data, and attributed data with post-processed rule bodies.

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

The second block illustrates two cases of slow-downs incurred by the use of attributed data. The first benchmark, `fib`, performs one hash-table lookup per new constraint, and the initial attributed data conversion preserves that count. Hence, the attributed data manipulation is pure overhead (25%). The second benchmark, `fib2`, modifies the simpagation rule of `fib`:

```
fib(N,F1) \ fib(N,F2) <=> F1 = F2.
```

into a simplification rule:

```
fib(N,F1), fib(N,F2) <=> F1 = F2, fib(N,F1).
```

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 visible speed-up.

The second slow-down, in `dijkstra`, results form a limitation of our current implementation, which does not allow multi-argument indices involving attributed data arguments. For this benchmark, such a multi-argument index would be more efficient than a single-argument attributed data index.

6 Related Work

Several programming languages define features that resemble our concept of attributed data. The `as_chr_key` annotation is related to (primary, secondary and foreign) keys in database tables and indexing declarations in some Prolog systems.

The conversion function ϕ 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.

The *solver types* facility of Mercury [2] 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.

7 Conclusion

We have presented attributed data—a new term representation that facilitates improving the efficiency of CHR indexing at a high level. A complementary post-processing procedure compensates for possible overhead of conversions between the new representation and the standard representation of Prolog terms.

Our technique has been implemented for the K.U.Leuven CHR system on SWI-Prolog. Evaluation on a set of benchmarks shows that using attributed data enables performance improvement, and that post-processing is critical to fully realize this potential.

As a further optimization of the approach, we could directly expose the abstract key types in the situations when there is no preference for the external argument representation. For example, programmers often use variables and integers as identifiers in CHR constraints. The nature of the data type is of no concern, as long as it supports unique value creation and value comparison. The appropriate choice of the abstract key type could eliminate unnecessary indirections of attributed variables or hash tables.

Two other interesting avenues for future work involve introducing support for automated inference of key type annotations, and extending attributed-data indexing to combinations of multiple arguments.

Acknowledgments

We are grateful for the helpful comments of the anonymous reviewers.

References

1. Slim Abdennadher and Michael Marte. University course timetabling using constraint handling rules. *Applied Artificial Intelligence*, 14(4):311–325, 2000.
2. Ralph Becket et al. Adding constraint solving to Mercury. In *8th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2006.
3. 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.
4. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
5. Thom Frühwirth. As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs. *Electronic Notes in Theoretical Computer Science*, 59(3), 2002.
6. 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.
7. 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.
8. 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.
9. 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.
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. 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.
14. Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transactions on Programming Languages and Systems*, 27(6):1216–1269, 2005.
15. 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.
16. Jan Wielemaker. SWI-Prolog release 5.6.0, 2006. http://www.swi-prolog.org/.