Optimization of CHR Propagation Rules

Peter Van Weert*

Department of Computer Science, K.U.Leuven, Belgium
Peter.VanWeert@cs.kuleuven.be

Abstract. Constraint Handling Rules (CHR) is an elegant, high-level programming language based on multi-headed, forward chaining rules. To ensure CHR propagation rules are applied at most once with the same combination of constraints, CHR implementations maintain a so-called *propagation history*. The performance impact of this history can be significant. We introduce several optimizations that, for the majority of CHR rules, eliminate this overhead. We formally prove their correctness, and evaluate their implementation in two state-of-the-art CHR systems.

1 Introduction

Constraint Handling Rules (CHR) [1, 2] is a high-level committed-choice CLP language, based on multi-headed, guarded multiset rewrite rules. Originally designed for the declarative specification of constraint solvers, it is increasingly used for general purposes, in a wide range of applications. Efficient implementations exist for several host languages, including Prolog [3], Haskell, and Java [4].

An important, distinguishing feature of CHR are *propagation rules*. Unlike most rewrite rules, propagation rules do not remove the constraints matched by their head. To avoid trivial non-termination, each CHR rule is therefore applied at most once with the same combination of constraints. This requirement stems from the formal study of properties such as termination and confluence [1], and is reflected in most current CHR implementations.

To prevent reapplication, a CHR runtime system maintains a so-called *propagation history*, containing a tuple for each constraint combination that fired a rule. Efficiently implementing a propagation history is challenging. Even with the implementation techniques proposed in e.g. [5–7], maintaining a propagation history remains expensive. Our empirical observations reveal that the history often has a significant impact on both space and time performance. Existing literature on CHR compilation nevertheless pays only scant attention to history-related optimizations. This paper resolves this discrepancy by introducing several novel approaches to resolve history-related performance issues. We show that, for almost all CHR rules, the propagation history can be eliminated completely. We either use innovative, alternate techniques to prevent rule reapplication, or prove that reapplication has no observable effect. Experimental results confirm the relevance and effectiveness of our optimizations.

* Research Assistant of the Research Foundation – Flanders (FWO-Vlaanderen).

Overview Section 3 discusses *non-reactive* CHR rules—rules that are not reconsidered when built-in constraints are added—and shows that their history can always be eliminated without affecting the program's operational semantics. More precisely, we prove that reapplication of non-reactive rules is either impossible, or that it can be prevented using a novel, more efficient technique.

Section 4 introduces the notion of *idempotence*. We prove that reapplying idempotent rules has no observable effect, and thus that their history can be eliminated as well, even if the rule is reactive. Together, the optimizations of Sections 3 and 4 cover the majority of the rules found in existing CHR programs.

We implemented the proposed optimizations in two state-of-the-art CHR implementations. Section 5 reports on the significant performance gains obtained. Section 6, finally, reviews some related work and concludes.

For self-containedness, we first briefly review CHR's syntax and operational semantics in Section 2. Gentler introductions are found for instance in [1, 5, 6].

2 Preliminaries

2.1 CHR Syntax

CHR is embedded in a host language $\mathcal{H}$. A *constraint type* c/n is denoted by a functor/arity pair; *constraints* $c(x_1, \ldots, x_n)$ are atoms constructed from these symbols. Their arguments x_i are instances of data types offered by $\mathcal{H}$. Many CHR systems support type and mode declarations for constraint arguments.

There are two classes of constraints: *built-in constraints*, solved by an underlying constraint solver of the host language $\mathcal{H}$, and *CHR constraints*, handled by a CHR program. A CHR program $\mathcal{P}$, also called a *CHR handler*, is a sequence of CHR rules. The generic syntactic form of a CHR rule is:

$$\rho \mathbin{@} H_k \setminus H_r \Leftrightarrow G \mid B$$

The rule's unique *name* ρ is optional; if omitted a name is assigned implicitly. The *head* consists of two conjunctions of CHR constraints, H_k and H_r. Their conjuncts are called *occurrences* (*kept* and *removed occurrences* resp.). If H_k is empty, the rule is a *simplification rule*. If H_r is empty, it is a *propagation rule*, and '$\Rightarrow$' is used instead of '$\Leftrightarrow$'. If both are non-empty, the rule is a *simpagation* rule. The *guard* G is a conjunction of built-in constraints, the *body* B a conjunction of CHR and built-in constraints. A trivial guard '`true` $\mid$ ' may be omitted.

Example 1. Fig. 1 shows a classic CHR handler, called LEQ. It defines a single CHR constraint, a less-than-or-equal constraint, using four CHR rules. All three kinds of rules are present. All constraint arguments are logical variables.

```
reflexivity  @ leq(X, X) ⇔ true.
idempotence  @ leq(X, Y) \ leq(X, Y) ⇔ true.
antisymmetry @ leq(X, Y), leq(Y, X) ⇔ X = Y.
transitivity @ leq(X, Y), leq(Y, Z) ⇒ leq(X, Z).
```

Fig. 1. LEQ, a CHR program for the less-than-or-equal constraint.

The handler uses a built-in equality constraint `=/2` (e.g. Prolog's built-in unification). The first two rules remove redundant constraints. The ***antisymmetry*** rule replaces the CHR constraints matched by its head with a built-in equality constraint. The ***transitivity*** propagation rule adds implied CHR constraints.

Head Normal Form In the Head Normal Form of a CHR program $\mathcal{P}$, denoted HNF($\mathcal{P}$), variables occur at most once in a rule's head. For instance in HNF(LEQ), the normalized form of the ***transitivity*** rule from Fig. 1 is:

transitivity `@ leq(X, Y), leq(`Y_1`, Z)` $\Rightarrow$ `Y = `Y_1 `| leq(X, Z).`

2.2 CHR's Refined Operational Semantics

The behavior of most current CHR implementations is captured formally by the *refined operational semantics* [8], commonly denoted as ω_r. The ω_r semantics is formulated as a state transition system, in which *transition rules* define the relation between subsequent *execution states*. The version presented here follows [5, 6], which is a slight refinement of the original specification [8].

Notation Sets, multisets and sequences (ordered multisets) are defined as usual. We use $S[i]$ to denote the i'th element of a sequence S, $+\!\!+$ for sequence *concatenation*, and $[e|S]$ to denote $[e] +\!\!+ S$. The *disjoint union* of sets is defined as: $\forall X, Y, Z : X = Y \sqcup Z \leftrightarrow X = Y \cup Z \wedge Y \cap Z = \emptyset$. For a logical expression X and a set V of variables, $vars(X)$ denotes the set of *free variables*, and *constraint projection* is defined as $\pi_V(X) \leftrightarrow \exists v_1, \ldots, v_n : X$ with $\{v_1, \ldots, v_n\} = vars(X) \setminus V$.

Execution States An execution state of ω_r is a tuple $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. The role of the *execution stack* $\mathbb{A}$ is explained below. The ω_r semantics is multiset-based. To distinguish between otherwise identical constraints, the *CHR constraint store* $\mathbb{S}$ is a set of *identified* CHR constraints, denoted $c\#i$, where each CHR constraint c is associated with a unique integer number i, called a *constraint identifier*. The projection operators $\text{CHR}(c\#i) = c$ and $\text{ID}(c\#i) = i$ are extended to sequences and sets in the obvious manner. The integer n represents the next available constraint identifier. The *built-in constraint store* $\mathbb{B}$ is a conjunction containing all built-in constraints passed to the built-in solver. Their meaning is determined by the built-in constraint theory $\mathcal{D}_\mathcal{H}$ (see e.g. [6] for a rigorous definition of $\mathcal{D}_\mathcal{H}$). The *propagation history* $\mathbb{T}$, finally, is a set of tuples, each recording a sequence of identifiers of CHR constraints that fired a rule, and the name of that rule.

Transition Rules Fig. 2 lists the transition rules of ω_r. Execution proceeds by exhaustively applying these transitions, starting from an *initial execution state* $\langle Q, \emptyset, \texttt{true}, \emptyset \rangle_1$. The constraint sequence Q is called the initial *query* Q.

CHR constraints are assigned unique identifiers and added to $\mathbb{S}$ in **Activate** transitions. The execution stack $\mathbb{A}$ is a sequence used to treat constraints as procedure calls. The top-most element of $\mathbb{A}$ is called the *active constraint*. When active, a CHR constraint performs a search for applicable rules. The ω_r semantics specifies that occurrences in a handler are tried in a top-down, right-to-left order. To realize this order in ω_r, identified constraints on the execution stack

1. Solve $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ if b is a built-in constraint. For the set of *reactivated constraints* $S \subseteq \mathbb{S}$, the following bounds hold: lower bound: $\forall H \subseteq \mathbb{S} : (\exists K, R : H = K +\!\!+ R \wedge \exists \rho \in \mathcal{P} : \neg appl(\rho, K, R, \mathbb{B}) \wedge appl(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset)$ and upper bound: $\forall c \in S : vars(c) \not\subset fixed(\mathbb{B})$.

2. Activate $\langle [c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle [c\#n : 1|\mathbb{A}], \{c\#n\} \sqcup \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_{n+1}$ if c is a CHR constraint (which has not yet been active or stored in $\mathbb{S}$).

3. Reactivate $\langle [c\#i|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle [c\#i : 1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if c is a CHR constraint (re-added to $\mathbb{A}$ by a **Solve** transition but not yet active).

4. Simplify $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle B +\!\!+ \mathbb{A}, K \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ with $\mathbb{S} = \{c\#i\} \sqcup K \sqcup R_1 \sqcup R_2 \sqcup S$, if the j-th occurrence of c in $\mathcal{P}$ occurs in rule ρ, and θ is a matching substitution such that $apply(\rho, K, R_1 +\!\!+ [c\#i] +\!\!+ R_2, \mathbb{B}, \theta) = B$.
Let $t = (\rho, \text{ID}(K +\!\!+ R_1) +\!\!+ [i] +\!\!+ \text{ID}(R_2))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

5. Propagate $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle B +\!\!+ [c\#i : j|\mathbb{A}], \mathbb{S} \setminus R, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ with $\mathbb{S} = \{c\#i\} \sqcup K_1 \sqcup K_2 \sqcup R \sqcup S$, if the j-th occurrence of c in $\mathcal{P}$ occurs in rule ρ, and θ is a matching substitution such that $apply(\rho, K_1 +\!\!+ [c\#i] +\!\!+ K_2, R, \mathbb{B}, \theta) = B$.
Let $t = (\rho, \text{ID}(K_1) +\!\!+ [i] +\!\!+ \text{ID}(K_2 +\!\!+ R))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

6. Drop $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if c has no j-th occurrence in $\mathcal{P}$.

7. Default $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle [c\#i : j+1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if the current state cannot fire any other transition.

Fig. 2. The transition rules of the refined operational semantics ω_r.

are *occurrenced*. If an occurrenced identified CHR constraint $c\#i : j$ is active, only matches with the j'th occurrence of c's constraint type are considered. Interleaving a sequence of **Default** transitions, all applicable rules are thus fired in **Propagate** and **Simplify** transitions. A rule is applicable if the store contains matching *partner constraints* for all remaining occurrences in its head. Formally:

Definition 1. *Given a conjunction of built-in constraints $\mathbb{B}$, a rule ρ is* applicable *with sequences of identified CHR constraints K and R, denoted $appl(\rho, K, R, \mathbb{B})$, iff a matching substitution θ exists for which $apply(\rho, K, R, \mathbb{B}, \theta)$ is defined. The latter partial function is defined as $apply(\rho, K, R, \mathbb{B}, \theta) = B$ iff $K \cap R = \emptyset$ and, renamed apart, ρ is of form "ρ @ $H_k \setminus H_r \Leftrightarrow G \mid B$" ($H_k$ or H_r may be empty) with $\text{CHR}(K) = \theta(H_k)$, $\text{CHR}(R) = \theta(H_r)$, and $\mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow \pi_{vars(\mathbb{B})}(\theta \wedge G)$.*

If the top-most element of $\mathbb{A}$ is a built-in constraint, this constraint is passed to the built-in solver in a **Solve** transition. As this may affect the entailment of guards, all CHR constraints for which additional rules might have become applicable have to be put back on the execution stack. These then cause **Reactivate** transitions to reinitiate searches for applicable rules. Constraints with fixed arguments are not reactivated, as no additional guards can become entailed.

Definition 2. *A variable v is* fixed *by constraint conjunction B, or $v \in fixed(B)$, iff $\mathcal{D}_{\mathcal{H}} \models \forall\theta((\pi_{\{v\}}(B) \wedge \pi_{\{\theta(v)\}}(\theta(B))) \rightarrow v = \theta(v))$ for any variable renaming θ.*

When a rule fires, its body is executed. By putting the body on the activation stack, the different conjuncts of the body are activated (for CHR constraints) or

```
:- chr_constraint up_to(+int), fib(+int,+int).

up_to(U) ⇒ fib(0,1), fib(1,1).
up_to(U), fib(N-1,M1), fib(N,M2) ⇒ N < U | fib(N+1,M1+M2).
```

Fig. 3. This handler, referred to as FIBBO, performs a bottom-up computation of all Fibonacci numbers up to a given number. All constraint arguments are fixed integers.

solved (for built-in constraints) in a left-to-right order. Control only returns to the original active constraint *after* the body is completely executed.

Derivations For a CHR operational semantics ω, an ω-derivation D is a (possibly infinite) sequence of ω_r states, with $D[1]$ an initial execution state for some query Q, and $D[i] \rightarrowtail_{\mathcal{P}} D[i+1]$ valid ω transitions. We use the notational abbreviation $\sigma_1 \rightarrowtail^{\star}_{\mathcal{P}} \sigma_n$ to denote a *finite* derivation $[\sigma_1, \ldots, \sigma_n]$.

3 Non-reactive Propagation Rules

Section 3.1 introduces *non-reactive CHR rules*, rules that are never matched by a reactivated constraint, and illustrates that a substantial portion of CHR rules is non-reactive. In Section 3.2, we prove that the history of certain non-reactive propagation rules can be eliminated, as CHR's operational semantics ensures these rules are never matched by the same constraint combination. For the remaining non-reactive rules, we introduce an innovative, more efficient technique to prevent rule reapplication in Section 3.3, and prove its soundness.

3.1 Introduction: From Fixed to Non-reactive CHR

Non-reactive CHR constraints are never reactivated when built-in constraints are added. Formally:

Definition 3. *A CHR constraint type c/n is* non-reactive *in a program $\mathcal{P}$ under a refined operational semantics $\omega_r^{\star}$ (ω_r or any of its refinements: see further) iff for any* **Solve** *transitions of the form $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ in any $\omega_r^{\star}$-derivation D the set of reactivated constraints $S \subseteq \mathbb{S}$ does not contain constraints of type c/n. A rule $\rho \in \mathcal{P}$ is* non-reactive *iff all constraint types that occur in its head are non-reactive in $\mathcal{P}$.*

The simplest instances are so-called *fixed* constraints. A CHR constraint type c/n is fixed iff $vars(c) \subseteq fixed(\emptyset)$ (see Definition 2) for all constraints c of this type. Clearly, if all constraint arguments are fixed, no additional rule becomes applicable when adding built-in constraints. Which CHR constraints are fixed is derived from their mode declarations, or using static *groundness analysis* [9].

Example 2. The FIBBO handler depicted in Fig. 3, performs a bottom-up computation of all Fibonacci numbers up to a given number. The constraint declarations[1] specify that all arguments are fixed instances of the host language's **int** type (the '+' mode declaration indicates a constraint's argument is fixed).

[1] The syntax is inspired by that of the K.U.Leuven CHR system [3, 6].

```
:- chr_constraint fib(+int,?int).
memoization @ fib(N,M1) \ fib(N,M2) ⇔ M1 = M2.
base_case   @ fib(N,M) ⇒ N ≤ 1 | M = 1.
recursion   @ fib(N,M) ⇒ N > 1 | fib(N-1,M1), fib(N-2,M2), M = M1 + M2.
```

Fig. 4. A CHR handler that computes Fibonacci numbers using a top-down computation strategy with memoization.

Under ω_r, a CHR constraint type is non-reactive iff it is fixed. The following example though shows why the class of non-reactive constraints should be larger:

Example 3. Fig. 4 contains an alternative Fibonacci handler, this time using a top-down computation strategy with memoization. The `fib/2` constraint is not fixed, and is typically called with a free (logical) variable as second argument—hence also the '`?`' mode declaration. Reactivating `fib/2` constraints is nevertheless pointless, as there are *no guards* constraining its second argument. Additional built-in constraints therefore never result in additional applicable rules.

All theoretical results in this section apply to non-reactive rules only. Under ω_r, however, constraints such as `fib/2` are not non-reactive. As using unbound, unguarded arguments to retrieve results is very common in CHR, a minor refinement ω_r is required to increase the practical relevance of our results.

In general, CHR constraints should only be reactivated if extra built-in constraints may cause more guards to become entailed. We therefore reintroduce the concept of *anti-monotonicity* [7, 10]:

Definition 4. *A conjunction of built-in constraints B is* anti-monotone *in a set of variables V iff* $\forall B_1, B_2((\pi_{vars(B)\setminus V}(B_1 \wedge B_2) \leftrightarrow \pi_{vars(B)\setminus V}(B_1)) \rightarrow ((\mathcal{D}_\mathcal{H} \not\models B_1 \rightarrow B) \rightarrow (\mathcal{D}_\mathcal{H} \not\models B_1 \wedge B_2 \rightarrow B)))$

Definition 5. *A CHR program $\mathcal{P}$ is* anti-monotone *in the i'th argument of a CHR constraint type c/n, if and only if for every occurrence $c(x_1, \ldots, x_i, \ldots, x_n)$ in HNF($\mathcal{P}$), the guard of the corresponding rule is anti-monotone in $\{x_i\}$.*

Any CHR program is anti-monotone in both fixed and unguarded constraint arguments. Moreover, several typical built-ins are anti-monotone in their arguments. In Prolog, for instance, `var(X)` is anti-monotone in `{X}`. Using anti-monotonicity, we now define ω_r', a slight refinement of ω_r[2]:

Definition 6. *Let $delay\_vars_\mathcal{P}(c)$ denote the set of variables in which $\mathcal{P}$ is not anti-monotone that occur in an (identified) CHR constraint c. Then ω_r' is obtained from ω_r by replacing the upper bound on the set of reactivated constraints S in its* **Solve** *transition with "$\forall c \in S : delay\_vars_\mathcal{P}(c) \not\subset fixed(\mathbb{B})$".*

Most rules in general-purpose CHR programs are non-reactive under ω_r'. Several CHR systems, including the K.U.Leuven CHR and JCHR systems [3, 4], implement ω_r'. Doing so, may already improve performance considerably (see [10]). In the following two subsections, we prove that for non-reactive CHR rules the expensive maintenance of a propagation history can always be avoided.

[2] We refer to [7, Appendix A] for a formal proof that ω_r' is indeed an instance of ω_r.

```
:- chr_constraint account(+client_id, +float), sum(+client_id, ?float).
:- chr_constraint gen(+client_id), sum(+float), get(?float).

sum_balances @  sum(C, Sum) ⇔ gen(C), get(Sum).
generate @  gen(C), account(C,B) ⇒ sum(B).
simplify @  sum(B1), sum(B2) ⇔ sum(B1 + B2).
retrieve @  get(Q), gen(_), sum(Sum) ⇔ Q = Sum.
```

Fig. 5. CHR rules computing the sum of the account balances of a given client. These rules may be part of some larger CHR handler modeling a banking application.

3.2 Propagation History Elimination

Because non-reactive CHR constraints are only active once, non-reactive propagation rules often do not require a history:

Example 4. The `sum/2` constraint in Fig. 5 computes the sum of a client's account balances using a common CHR programming idiom to compute *aggregates*: a (typically non-reactive) propagation rule generates a number of constraints, from which, after simplification to a single constraint, the result can be retrieved.

When the active `gen/1` constraint considers the *`generate`* rule, it iterates over candidate `account/2` partner constraints. Assuming this iteration does not contain duplicates (a property formalized shortly in Definition 8), the *`generate`* rule never fires with the same constraint combination under ω_r, even if no propagation history is maintained. Indeed, the *`generate`* rule only adds `sum/1` constraints, which, as there is no `get/1` constraint yet in the store (the body of the *`sum_balances`* rule is executed from left to right), only fire the *`simplify`* rule.

The history, however, is not superfluous for all non-reactive CHR rules, as shown by the following example:

Example 5. Reconsider the FIBBO handler of Fig. 3. If an `up_to(U)` constraint is told, the first rule propagates two `fib/2` constraints. After this, the second rule propagates all required `fib/2` constraints, each time with a `fib/2` constraint as the active constraint. Next, control returns to the `up_to(U)` constraint, and advances to its second occurrence. Some mechanism is then required to prevent the second (non-reactive) propagation rule to add erroneous `fib/2` constraints.

So, non-reactive propagation rules can match the same constraint combination more than once. This occurs if one or more partner constraints for an active constraint in rule ρ were added by firing ρ or some earlier rule, whilst the same constraint was already active. We say these partner constraints *observe* the corresponding occurrence of the active constraint in ρ (cf. also [9]). Formally:

Definition 7. *Let the k'th occurrence of a rule ρ's head be the j'th occurrence of constraint type c/n. Then this occurrence is* unobserved *under a refined operational semantics $\omega_r^\star$ iff for all* **Activate** *or* **Default** *transitions of the form*[3]*:*

$$\langle \mathbb{A}_0, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_\_ \rightarrowtail_{\mathcal{P}} \langle [c\#i : j | \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_\_$$

[3] We use '_' to denote that we are not interested in the identifier counter.

($\mathbb{A}_0[1] = c\#i$ *or* $\mathbb{A}_0[1] = c\#i:j-1$) *the following holds:* $\forall(\rho, I) \in \mathbb{T} : I[k] \neq i$, *and similarly for all transition sequences starting with a* **Propagate** *transition*

$$\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{\_} \rightarrowtail_{\mathcal{P}} \langle B +\!\!+ \mathbb{A}, \mathbb{S}', \mathbb{B}', \mathbb{T}' \rangle_{\_} \rightarrowtail^{\star}_{\mathcal{P}} \langle \mathbb{A}, \mathbb{S}'', \mathbb{B}'', \mathbb{T}'' \rangle_{\_}$$

with $\mathbb{A}[1] = c\#i:j$, $\forall(\rho, I) \in \mathbb{T}''\backslash\mathbb{T}' : I[k] \neq i$.

Let $\omega_r^\dagger$ denote the semantics obtained from ω_r' by adding the following condition to its **Propagate** and **Simplification** transitions: "*If the* j*'th occurrence of* c *is unobserved under* ω_r'*, then* $\mathbb{T}' = \mathbb{T}$ ". Also, to prevent trivial reapplication in a consecutive sequence of **Propagate** transitions (see e.g. Example 4), propagation in $\omega_r^\dagger$ is defined to be *duplicate-free*:

Definition 8 (Duplicate-free Propagation). *Propagation in a refined operational semantics* $\omega_r^\star$ *is* duplicate-free *iff for all* $\omega_r^\star$*-derivations* D *of a CHR program* $\mathcal{P}$ *where the* j*'th occurrence of* c *is kept, the following holds:*

if $\begin{cases} \sigma_1 \rightarrowtail_{\mathcal{P}} \sigma_2 \rightarrowtail^{\star}_{\mathcal{P}} \sigma_1' \rightarrowtail_{\mathcal{P}} \sigma_2' \text{ is part of } D \\ \sigma_1 = \langle [c\#i:j|\mathbb{A}], \mathbb{S}, \ldots \rangle_{\_} \text{ and } \sigma_1' = \langle [c\#i:j|\mathbb{A}], \mathbb{S}', \ldots \rangle_{\_} \\ \sigma_1 \rightarrowtail_{\mathcal{P}} \sigma_2 \text{ is a } \textbf{Propagate} \text{ transition applied with constraints } H \subseteq \mathbb{S} \\ \sigma_1' \rightarrowtail_{\mathcal{P}} \sigma_2' \text{ is a } \textbf{Propagate} \text{ transition applied with constraints } H' \subseteq \mathbb{S}' \\ \text{between } \sigma_2 \text{ and } \sigma_1' \text{ no } \textbf{Default} \text{ transition occurs of the form} \\ \qquad \sigma_2 \rightarrowtail^{\star}_{\mathcal{P}} \langle [c\#i:j|\mathbb{A}], \ldots \rangle_{\_} \rightarrowtail_{\mathcal{P}} \langle [c\#i:j+1|\mathbb{A}], \ldots \rangle_{\_} \rightarrowtail^{\star}_{\mathcal{P}} \sigma_1' \end{cases}$

then $H \neq H'$.

The following theorem establishes the equivalence of $\omega_r^\dagger$ and ω_r', thus proving the soundness of eliminating the history of unobserved CHR rules:

Theorem 1. *Define the mapping function* $\alpha^\dagger$ *as follows:*

$$\alpha^\dagger(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \{(\rho, I) \in \mathbb{T} \mid \rho \text{ is not unobserved}\} \rangle_n$$

If D *is an* ω_r' *derivation, then* $\alpha^\dagger(D)$ *is an* $\omega_r^\dagger$ *derivation. Conversely, if* D *is an* $\omega_r^\dagger$ *derivation, then there exists an* ω_r' *derivation* D' *such that* $\alpha^\dagger(D) = D'$.

Proof. See [11]. □

Implementation The main difficulty in the implementation of this optimization is deriving that a rule is unobserved (enforcing duplicate-free propagation is typically straightforward, as shown in Section 3.3). The abstract interpretation-based late storage analysis of [9], which derives a similar observation property, can be adapted for this purpose. The details are beyond the scope of this paper.

3.3 Optimized Reapplication Avoidance

Non-reactive CHR rules that are not unobserved, such as the second rule in the FIBBO handler of Example 5, do require some mechanism to prevent reapplication. Moreover, even if a rule is unobserved, this does not mean the compiler's analysis is capable of deriving it. In this section we therefore present a novel, very efficient technique that prevents the reapplication of any non-reactive propagation rule without maintaining a costly propagation history.

The central observation is that, when a non-reactive rule is applied, the active constraint is always more recent than its partner constraints:

Lemma 1. *Let $\mathcal{P}$ be an arbitrary CHR program, with $\rho \in \mathcal{P}$ a non-reactive rule, and D an arbitrary ω_r' derivation with this program. Then for each* **Simplify** *or* **Propagate** *transition in D of the form*

$$\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{A}', \mathbb{S}', \mathbb{B}', \mathbb{T} \sqcup \{(\rho, I_1 +\!\!+ [i] +\!\!+ I_2)\}\rangle_n \qquad (1)$$

the following holds: $\forall i' \in I_1 \cup I_2 : i' < i$.

Proof. Assume $i' = \max(I_1 \sqcup I_2)$ with $i' \geq i$. By Definition 1 of rule applicability, $i' \neq i$, and $\exists c'\#i' \in \mathbb{S}$. This $c'\#i'$ partner constraint must have been stored in an **Activate** transition. Since $i' = \max(I_1 \sqcup \{i\} \sqcup I_2)$, in D, this transition came *after* the **Activate** transitions of all other partners, including $c\#i$. In other words, all constraints in the matching combination of transition (1) were stored prior to the activation of $c'\#i'$. Also, in (1), $c\#i$ is back on top of the activation stack. Because c is non-reactive, and thus never put back on top by a **Reactivate** transition, the later activated $c'\#i'$ must have been removed from the stack in a **Drop** transition. This implies that all applicable rules matching c' must have fired. As all required constraints were stored (see earlier), this includes the application of ρ in (1). By contradiction, our assumption is false, and $i' < i$. □

Let $\omega_r^\ddagger$ denote the semantics obtained from ω_r' by replacing the propagation history condition in its **Simplify** and **Propagate** transitions with the following:

> *If ρ is non-reactive, then $\forall i' \in \text{ID}(H_1 \cup H_2) : i' < i$ and $\mathbb{T}' = \mathbb{T}$. Otherwise, let $t = (\rho, \text{ID}(H_1) +\!\!+ [i] +\!\!+ \text{ID}(H_2))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.*

Propagation in $\omega_r^\ddagger$ is again duplicate-free, as defined by Definition 8. Similarly to Theorem 1, the following theorem proves that ω_r' and $\omega_r^\ddagger$ are equivalent:

Theorem 2. *Define the mapping function $\alpha^\ddagger$ as follows:*

$$\alpha^\ddagger(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n) = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \{(\rho, I) \in \mathbb{T} \mid \rho \text{ is a reactive CHR rule}\}\rangle_n$$

If D is an ω_r' derivation, then $\alpha^\ddagger(D)$ is an $\omega_r^\ddagger$ derivation. Conversely, if D is an $\omega_r^\ddagger$ derivation, then there exists an ω_r' derivation D' such that $\alpha^\ddagger(D) = D'$.

Proof. See [7] or [11]. □

Implementation The standard CHR compilation scheme (see e.g. [5, 6]) generates for each occurrence a nested iteration that looks for matching partner constraints for the active constraint. If a matching combination is found, and the active constraint is not removed, the constraint iterators are suspended and the rule's body is executed. Afterwards, the nested iteration is simply resumed.

Example 6. Fig. 6(a) shows the generated code for the second occurrence of the `up_to/1` constraint in Fig. 3. For the query `up_to(U)`, the propagation history for the corresponding rule would require $\mathcal{O}(U)$ space. Because all constraints are non-reactive, however, no propagation history has to be maintained. Simply comparing constraint identifiers suffices.

If all iterators return candidate partner constraints at most once, propagation is guaranteed to be duplicate-free (see Definition 8). Most iterators used by CHR implementations have this property. If not, a temporary history can for instance be maintained whilst the active constraint is considering an occurrence.

```
procedure up_to(U)#ID : 2
  foreach fib(N,M2)#ID2 in ...
    foreach fib(N-1,M1)#ID1 in ...
      if N < U
        if ID < ID1 and ID < ID2
          ⋱
```

(a) Efficient reapplication avoidance using identifier comparisons

```
procedure up_to(U)#ID : 2
  foreach fib(N,M2)#ID2 in ...
    if ID < ID2 and N < U
      foreach fib(N-1,M1)#ID1 in ...
        if ID < ID1
          ⋱
```

(b) After *Loop-invariant Code Motion*

Fig. 6. Pseudocode for the second occurrence of the `up_to/1` constraint of Fig. 3.

Loop-invariant Code Motion Most CHR compilers perform so-called *Loop-invariant Code Motion* optimization to check guard entailment as soon as possible (e.g. '`N < U`' in Fig. 6(b)). Contrary to a propagation history check, identifier comparisons enable additional code motion, as illustrated in Fig. 6(b). This may prune the search space of candidate partner constraints considerably.

Note furthermore that Lemma 1 does not only apply to propagation rules, but also to simplification and simpagation rules. Whilst maintaining a history for non-propagation rules is pointless, comparing partner constraint identifiers in outer loops is not, as they may avoid redundant iterations of nested loops.

4 Idempotence

Constraints in CHR handlers that specify traditional constraint solvers, such as the `leq/2` constraint of Example 1, typically range over unbound variables, and are thus highly reactive. Without a history, constraint reactivations may cause reactive propagation rules to fire multiple times with the same combination. For constraint solvers, however, such additional rule applications typically have no effect, as they only add redundant constraints that are immediately removed. For such rules, the propagation history may be eliminated as well.

Example 7. Suppose the reactive ***transitivity*** propagation rule of Fig. 1 is allowed to fire a second time with the same constraint combination matching its head, thus adding a `leq(X,Z)` constraint for the second time. If the earlier told duplicate is still in the store, this redundant `leq(X,Z)` constraint is immediately removed by the ***idempotence*** rule. Otherwise, the former duplicate must have been removed by either the ***reflexivity*** or the ***antisymmetry*** rule. It is easy to see that in this case `X = Z`, and thus that the new, redundant `leq(X,Z)` constraint is again removed immediately by the ***reflexivity*** rule.

We say the `leq/2` constraint of the above example is *idempotent*. With $live(\mathbb{T}, \mathbb{S}) = \{(\rho, I) \in \mathbb{T} \mid I \subseteq \text{ID}(\mathbb{S})\}$, *idempotence* is defined formally as:

Definition 9. *A CHR constraint type c/n is* idempotent *in a CHR program $\mathcal{P}$ under a refined semantics $\omega_r^\star$ iff for any state $\sigma = \langle [c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ in a $\omega_r^\star$ derivation D with c a CHR constraint, the following holds: if earlier in D a state $\langle [c'|\mathbb{A}'], \mathbb{S}', \mathbb{B}', \mathbb{T}' \rangle_{n'}$ occurs with $\mathcal{D}_\mathcal{H} \models \mathbb{B} \rightarrow c = c'$, then $\sigma \rightarrowtail_\mathcal{P}^\star \langle \mathbb{A}, \mathbb{S}'', \mathbb{B}'', \mathbb{T}'' \rangle_{n''}$ with $\mathbb{S}'' = \mathbb{S}$, $live(\mathbb{T}'', \mathbb{S}) = live(\mathbb{T}, \mathbb{S})$, and $\mathcal{D}_\mathcal{H} \models \pi_{vars(\mathbb{B}) \cup vars(D[1])}(\mathbb{B}'') \leftrightarrow \mathbb{B}$.*

In other words, an idempotent constraint c for which a syntactically equal constraint c' was told earlier in the same derivation, is removed without making any observable state change. Since '$\rightarrowtail^\star_{\mathcal{P}}$' denotes a finite derivation, telling duplicate idempotent CHR constraints also does not affect termination.

We do not consider arbitrary, extra-logical host language statements here, and assume all built-in constraints b are idempotent, that is: $\forall b : \mathcal{D}_\mathcal{H} \models b \wedge b \leftrightarrow b$. By adding "*If* $\mathcal{D}_\mathcal{H} \models (\mathbb{B} \wedge b) \leftrightarrow \mathbb{B}$, *then* $S = \emptyset$" to the **Solve** transition of ω_r (or any of its refinements from Section 3), we avoid redundant constraint reactivations when idempotent built-in constraints are told. This is correct, as **Solve**'s upper bound on S already specifies that any matching already possible prior to b's addition may be omitted from S. Most CHR systems already implement this optimization. Denote the resulting semantics ω_r^{idem}.

Definition 10. *A CHR rule $\rho \in \mathcal{P}$ is* idempotent *under ω_r^{idem} iff all CHR constraint types that occur in its body are idempotent in $\mathcal{P}$ under ω_r^{idem}.*

We now prove that an idempotent propagation rule may be fired more than once with the same combination of constraints, without affecting a program's operational semantics. Let $\omega_r^{idem'}$ denote the semantics obtained by adding the following phrase to the **Simplify** and **Propagate** transitions of ω_r^{idem}:

If the rule ρ is idempotent, then $\mathbb{T}' = \mathbb{T}$; otherwise, ... (as before)

Assuming furthermore that propagation for $\omega_r^{idem'}$ is duplicate-free[4] in the sense of Definition 8, the $\omega_r^{idem'}$ semantics is equivalent to ω_r^{idem}. More precisely:

Theorem 3. *If D' is an $\omega_r^{idem'}$ derivation, then there exists an ω_r^{idem} derivation D with $D[1] = D'[1]$ such that a monotonic function α can be defined from the states in D to states in D' for which*

- *$\alpha(D[1]) = D'[1]$*
- *if $\alpha(D[i]) = D'[k]$ and $\alpha(D[j]) = D'[l]$ with $i < j$, then $k < l$*
- *if $\alpha(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle \mathbb{A}', \mathbb{S}', \mathbb{B}', \mathbb{T}' \rangle_{n'}$, then $\mathcal{D}_\mathcal{H} \models \pi_{vars(\mathbb{B}) \cup vars(D[1])}(\mathbb{B}') \leftrightarrow \mathbb{B}$, $\mathbb{A}' = \mathbb{A}$, $\mathbb{S}' = \mathbb{S}$, and $live(\mathbb{T}', \mathbb{S}) = live(\mathbb{T}, \mathbb{S}) \setminus \{(\rho, I) \in \mathbb{T} \mid \rho$ is idempotent$\}$.*

Conversely, if D is an ω_r^{idem} derivation, then an $\omega_r^{idem'}$ derivation D' exists with $D'[1] = D[1]$ for which a function with these same properties can be defined.

Proof Sketch. An $\omega_r^{idem'}$ derivation D' only differs from the corresponding ω_r^{idem} derivation D when a **Propagate** transition fires an idempotent propagation rule ρ using a combination of constraints that fired ρ before. This $\omega_r^{idem'}$ transition has form $\sigma_0 = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_\mathcal{P} \langle B +\!\!+ \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n = \sigma_1$. Because ρ's body B is idempotent, it follows from Definition 9 that the remainder of D' begins with $\sigma_1 \rightarrowtail^\star_\mathcal{P} \sigma'_0 = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}', \mathbb{T}' \rangle_n$, with $\mathcal{D}_\mathcal{H} \models \pi_{vars(\mathbb{B}) \cup vars(D[1])}(\mathbb{B}') \leftrightarrow \mathbb{B}$, and $live(\mathbb{T}', \mathbb{S}) = live(\mathbb{T}, \mathbb{S})$. Because σ'_0 is thus essentially equivalent to σ_0, we simply omit states σ_1 to σ'_0 in the corresponding ω_r^{idem} derivation D.

Given above observations it is straightforward to construct the mapping function α and the required derivations for both directions of the proof. □

[4] In this case a *finite* number of duplicate propagations would also not be a problem.

For multi-headed propagation rules, reapplication is often cheaper than maintaining and checking a history. The experimental results of Section 5 confirm this. Of course, reapplying a body can be arbitrarily expensive. To estimate the cost of reapplication versus the cost of maintaining a history, heuristics can be used.

4.1 Deriving Idempotence

The main challenge lies in automatically deriving that a CHR constraint is idempotent. A wide class of idempotent CHR constraints should be covered:

Example 8. Many constraint solvers contain a rule such as:

$$\texttt{in(X,L}_1\texttt{,U}_1\texttt{)} \setminus \texttt{in(X,L}_2\texttt{,U}_2\texttt{)} \Leftrightarrow \texttt{L}_2 \leq \texttt{L}_1\texttt{, U}_2 \geq \texttt{U}_1 \mid \texttt{true.}$$

Here, '`in(X,L,U)`' denotes that the variable `X` lies in the interval [`L`, `U`]. The `in/3` constraint is probably idempotent (it depends on the preceding rules). There is an important difference though with the `leq/2` constraint in Example 7: by the time the constraint is told for the second time, the earlier told duplicate may now be replaced with a *syntactically different* constraint—in this case: a constraint representing a smaller interval domain.

Theorem 4 provides a sufficiently strong syntactic condition for determining the idempotence of a CHR constraint. It uses arbitrary preorders on the constraint's arguments. For the three arguments of the `in/3` constraint in Example 8 for instance, the preorders $=$, $\leq$ and $\geq$ can be used respectively.

Let $bi(B)$ and $chr(B)$ denote the conjunction of built-in respectively CHR constraints that occur in a constraint conjunction B. Then:

Theorem 4. *A CHR constraint type c/n is idempotent in $\mathcal{P}$ under ω_r^{idem} if for preorders $\lhd_1, \ldots, \lhd_n$:*

1. *There exists a rule of the form "$c(y_1, \ldots, y_n) \setminus c(x_1, \ldots, x_n) \Leftrightarrow G \mid true$." in HNF($\mathcal{P}$) with $\mathcal{D}_\mathcal{H} \models (x_1 \lhd_1 y_1 \wedge \ldots \wedge x_n \lhd_n y_n) \rightarrow G$.*
   Let ρ be the first such rule occurring in the HNF($\mathcal{P}$) sequence.
2. *All rules in HNF($\mathcal{P}$) prior to ρ that contain an occurrence of c/n have a trivial body '`true`', and do not contain any removed occurrences apart from possibly that c/n occurrence.*

Consider a set of n mutually distinct variables $V = \{x_1, \ldots, x_n\}$. For all removed occurrences of c/n in HNF($\mathcal{P}$) that can be renamed to the form

$$H_k \setminus H_{r_1}, c(x_1, \ldots, x_n), H_{r_2} \Leftrightarrow G \mid B$$

(H_k, H_{r_1}, and H_{r_2} may be empty), such that $\neg\exists c(y_1, \ldots, y_n) \in H_k \cup chr(B) : \mathcal{D}_\mathcal{H} \models G \wedge bi(B) \rightarrow (x_1 \lhd_1 y_1 \wedge \ldots \wedge x_n \lhd_n y_n)$, define $\Phi = \pi_V(G \wedge bi(B))$. For each of these occurrences, either $\mathcal{D}_\mathcal{H} \models \Phi \leftrightarrow false$, or conditions 3 and 4 hold:

3. *There exists a rule in HNF($\mathcal{P}$) that can be renamed such that it has form "$c(x_1, \ldots, x_n) \Leftrightarrow G \mid B$", with $bi(B) = B$ and $\mathcal{D}_\mathcal{H} \models \Phi \rightarrow (G \wedge B)$.*
   Let ρ' be the first such rule occurring in the HNF($\mathcal{P}$) sequence.

	SWI		JCHR			total #	n-headed propagation rules			
	history	non-react	history	non-react	non-react+	rules	$n=1$	$n=2$	$n=3$	$n>3$
FIBBO(1000)	15,929	4,454 (28%)	70	67 (95%)	21 (30%)	3	1	-	1	-
FIBBO(3000)	*timeout*	*timeout*	542	464 (85%)	153 (28%)	3	1	-	1	-
FLOYD-WARSH(30)	11,631	9,706 (83%)	368	188 (51%)	186 (51%)	21	3	2	1	-
INTERPOL(8)	5,110	1,527 (30%)	43	41 (95%)	37 (86%)	5	-	2	-	-
MANNERS(128)	849	561 (66%)	328	322 (98%)	317 (97%)	8	-	-	1	-
NSP_GRND(12)	547	169 (31%)	10	6 (60%)	5 (50%)	3	1	1	-	-
NSP_GRND(36)	81,835	10,683 (13%)	1,434	502 (35%)	494 (34%)	3	1	1	-	-
SUM(1000,100)	6,773	3,488 (51%)	215	135 (63%)	*N/A*	4	-	1	-	-
TURING(20)	10,372	7,387 (71%)	761	280 (37%)	276 (36%)	11	1	4	1	5
WFS(200)	2,489	2,143 (86%)	71	67 (94%)	67 (94%)	44	-	4	-	-

Table 1. Benchmark results (in average milliseconds) for non-reactive CHR rules.

4. *All rules in HNF($\mathcal{P}$) prior to ρ' that contain an occurrence of c/n can be renamed to "$H_k \setminus H_r \Leftrightarrow G \mid B$" with $H_k +\!\!+ H_r = H_1 +\!\!+ [c(x_1,\ldots,x_n)] +\!\!+ H_2$, such that either*
   - $\mathcal{D}_\mathcal{H} \models \Phi \rightarrow \neg G$; *or*
   - $H_r \subseteq [c(x_1,\ldots,x_n)] \wedge (bi(B) = B) \wedge \mathcal{D}_\mathcal{H} \models (\Phi \wedge G) \rightarrow B$; *or*
   - $\exists c(y_1,\ldots,y_n) \in H_1 \cup H_2 : \mathcal{D}_\mathcal{H} \models (\Phi \wedge G) \rightarrow (x_1 \lhd_1 y_1 \wedge \ldots \wedge x_n \lhd_n y_n)$.

Proof Sketch. By Definition 9, we have to show that adding a c/n constraint makes no essential changes to the execution state if a duplicate constraint was added earlier in the same derivation. The proof considers two cases: either the duplicate constraint, or a constraint derived from it, is still in the store, or it has been removed. We show that, if the theorem's conditions hold, in both these cases the newly told duplicate is removed, and that it only makes idempotent state changes before that. The complete, formal proof can be found in [11]. □

5 Evaluation

We implemented the optimizations introduced in this paper in the K.U.Leuven CHR system [3, 6] for SWI-Prolog, and in the K.U.Leuven JCHR system [4] for Java, and evaluated them using typical CHR benchmarks and constraint solvers[5]. Benchmark timings are given in Tables 1 and 2. The *history* columns give the reference timings (in milliseconds) when using a propagation history.

The *non-react* columns in Table 1 contain the results when the optimizations of Section 3 are used. For the *non-react+* measurements, loop-invariant code motion was applied to the identifier comparisons (see Section 3.3; currently only implemented in JCHR[6]). Only for the SUM benchmark the the history was eliminated using the optimization of Section 3.2 (code motion is of course not applicable (*N/A*) in this case). Table 2 shows the results for the idempotence-based history elimination of Section 4.

Significant performance gains are measured all optimizations. The selected benchmarks run about two times faster on average, and scale better as well. Even

[5] Information on the benchmarks and the platform used is found in [11, Appendix B].

[6] In JCHR, after code motion, identifier comparisons are integrated in the constraint iterators themselves. These iterators moreover exploit the fact that the stored constraints are often sorted on their identifiers. This can further improve performance.

	SWI		JCHR		#		SWI		JCHR		#
	history	idempotence	hist.	idempot.			history	idempotence	hist.	idempot.	
INTERVAL(21)	22,622	17,611 (78%)	8	5 (62%)	15/27	EQ(35)	3,465	1,931 (56%)	47	19 (40%)	1/4
INTERVAL(42)	*timeout*	*timeout*	54	28 (52%)	15/27	LEQ(70)	3,806	1,236 (32%)	85	35 (41%)	1/4
NSP_GRND(12)	547	164 (30%)	10	6 (60%)	2/3	NSP(12)	1,454	1,036 (71%)	12	8 (67%)	2/3
NSP_GRND(36)	81,835	10,485 (13%)	1,365	496 (36%)	2/3	NSP(36)	*timeout*	*timeout*	1,434	621 (43%)	2/3
TIMEPOINT(16)	1,684	1,312 (78%)	404	317 (78%)	2/7	MINMAX(15)	4,826	3,631 (75%)	133	82 (61%)	6/54

Table 2. Benchmark results (in average milliseconds) for idempotent propagation rules. The '#' columns give the number of propagation rules over the total number of rules.

though no numbers are shown, it is moreover clear that the space complexity of the propagation histories has become optimal. Unoptimized, the worst-case space consumption of a propagation history is linear in the number of rule applications (cf. Example 6). Using our optimizations, histories consume no space at all. In extreme cases, this even improves the space complexity of the entire handler.

6 Conclusions

Related Work A preliminary version of this paper covering only Section 3.3 of the present paper appeared in [7]. The present paper completes this earlier work by introducing propagation history elimination based on unobservedness and idempotence, and by providing a more extensive experimental evaluation.

Section 3.2 can be seen as an extension and formalization of an optimization briefly presented in [5]. This ad-hoc optimization was restricted to fixed CHR constraints, and lacked a formal correctness proof.

Since the propagation history contributes to significant performance issues when implementing CHR in a tabling environment (see e.g. [12]), [13] proposes an alternative set-based CHR semantics, and argues that it does not need a propagation history. Our results, however, show that abandoning CHR's familiar multiset-based semantics is not necessary: indeed, our optimizations eliminate the history-related performance issues whilst preserving the ω_r-semantics.

Conclusions Whilst there is a vast research literature on CHR compilation and optimization, propagation histories never received much attention. Maintaining a propagation history, however, comes at a considerable runtime cost, both in time and in space. In this work, we resolved this discrepancy by introducing several innovative optimization techniques that circumvent the maintenance of a history for the majority of CHR propagation rules:

- For *non-reactive* CHR propagation rules, we showed that very cheap constraint identifier comparisons can be used. These comparisons can moreover be moved early in the generated nested iterations, thus pruning the search space of possible partner constraints. We also formally identified the class of non-reactive rules for which the history can simply be eliminated.
- Whilst rules in general-purpose CHR programs are mostly non-reactive, CHR handlers that specify a constraint solver are typically highly reactive. We therefore introduced the concept of *idempotence*, and found that most rules in the latter handlers are idempotent. We showed that if a propagation

rule is idempotent, the rule may safely be applied more than once matching the same combination of constraints. Interestingly, reapplication is mostly cheaper than maintaining and checking a history. We also presented a sufficient syntactic condition for the idempotence of a CHR constraint.

We proved the correctness of all our optimizations and analyses in the formal framework of CHR's refined operational semantics [8], and implemented them in two state-of-the-art CHR systems [3, 4]. Our experimental results show significant performance gains for all benchmarks containing propagation rules.

Acknowledgments The author thanks Tom Schrijvers for his invaluable aid in the implementation of the optimizations in the K.U.Leuven CHR system. Thanks also to Bart Demoen and the anonymous referees of CHR 2008 and ICLP 2008 for their useful comments on earlier versions of this paper.

References

1. Frühwirth, T.: Theory and practice of Constraint Handling Rules. J. Logic Programming, Special Issue on Constraint Logic Programming **37**(1–3) (1998) 95–138
2. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. Submitted to *Journal of Theory and Practice of Logic Programming* (2008)
3. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: Implementation and application. In: CHR 2004: Selected Contributions, Ulm, Germany (2004) 8–12
4. Van Weert, P., Schrijvers, T., Demoen, B.: K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In: CHR 2005: Proc. 2nd Workshop on Constraint Handling Rules, Sitges, Spain (2005) 47–62
5. Duck, G.J.: Compilation of Constraint Handling Rules. PhD thesis, University of Melbourne, Australia (December 2005)
6. Schrijvers, T.: Analyses, optimizations and extensions of Constraint Handling Rules. PhD thesis, K.U.Leuven, Belgium (June 2005)
7. Van Weert, P.: A tale of histories. In: CHR 2008: Proc. 5th Workshop on Constraint Handling Rules, Hagenberg, Austria (2008) 79–94
8. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. [14] 90–104
9. Schrijvers, T., Stuckey, P.J., Duck, G.J.: Abstract interpretation for Constraint Handling Rules. In Barahona, P., Felty, A., eds.: PPDP '05: Proc. 7th Intl. Conf. Princ. Pract. Declarative Programming, Lisbon, Portugal, ACM (2005) 218–229
10. Schrijvers, T., Demoen, B.: Antimonotony-based delay avoidance for CHR. Technical Report CW 385, K.U.Leuven, Dept. Computer Science (July 2004)
11. Van Weert, P.: Optimization of CHR propagation rules: Extended report. Technical Report CW 519, K.U.Leuven, Dept. Computer Science (August 2008)
12. Schrijvers, T., Warren, D.S.: Constraint Handling Rules and tabled execution. [14] 120–136
13. Sarna-Starosta, B., Ramakrishnan, C.: Compiling Constraint Handling Rules for efficient tabled evaluation. In Hanus, M., ed.: PADL 2007. Volume 4354 of LNCS., Nice, France, Springer (January 2007) 170–184
14. Demoen, B., Lifschitz, V., eds.: ICLP 2004. Volume 3132 of LNCS. Springer, Saint-Malo, France (September 2004)