

Optimization of CHR Propagation Rules: Extended Report

Peter Van Weert

Report CW519, August 2008



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

Optimization of CHR Propagation Rules: Extended Report

Peter Van Weert

Report CW 519, August 2008

Department of Computer Science, K.U.Leuven

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. This extended report contains complete formal proofs of all theoretical results.

Keywords : Constraint Handling Rules, CHR, optimization.

CR Subject Classification : D.3.2 [Programming Languages] Language Classifications — Constraint and logic languages; D.3.4 Processors — Compilers, Optimization.

Optimization of CHR Propagation Rules

Extended Report

Peter Van Weert*

Report CW 519, August 2008

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. This extended report contains complete formal proofs of all theoretical results.

1 Introduction

Constraint Handling Rules (CHR) [1, 5, 7, 16] 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, the CHR programming language is increasingly used for general purposes, in a wide range of applications (see [16] for a recent survey). Efficient implementations exist for several host languages, including Prolog [8, 12], Haskell [17], and Java [19].

An important, distinguishing feature of CHR are *propagation rules*. Unlike most rewrite rules, propagation rules do not remove the constraints matched by their head. They only add extra, implied constraints. Logically, a propagation rule corresponds to an implication. 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 [5], and is reflected in most current CHR implementations [8, 12, 19].

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. [3, 10, 21], maintaining a propagation history remains expensive. Our empirical observations reveal 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.

1.1 Overview

This technical report is an extended version of [18], containing the formal proofs of all theoretical results. A preliminary version of this paper appeared in [21].

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

```

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).

```

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

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 [3, 5, 7, 10].

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* $c(x_1, \dots, x_n)$ 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:

$$\rho @ H_k \setminus H_r \Leftrightarrow G \mid B$$

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** | ’ may be omitted.

Example 1 *Figure 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. The handler uses a built-in equality constraint $=/2$ (solved by 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 redundant CHR constraints.*

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

$$\textit{transitivity} @ \textit{leq}(X, Y), \textit{leq}(Y_1, Z) \Rightarrow Y = Y_1 \mid \textit{leq}(X, Z).$$

2.2 CHR's Refined Operational Semantics

The behavior of most current CHR implementations is captured formally by the refined operational semantics [4], 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 [3, 10], which is a slight modification of the original specification [4].

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 $\pi_V(X) \leftrightarrow \exists v_1, \dots, v_n : X$ with $\{v_1, \dots, v_n\} = vars(X) \setminus V$.

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 sequence \mathbb{A} , called the *execution stack*, is explained below, in the paragraph on ω_r 's transition rules. The ω_r semantics is multiset-based. To distinguish between otherwise identical constraints, the *CHR constraint store* \mathbb{S} is thus a set of *identified* CHR constraints, denoted $c\#i$, where each CHR constraint c is associated with a unique *constraint identifier* $i \in \mathbb{N}$. The projection operators $chr(c\#i) = c$ and $id(c\#i) = i$ are extended to sequences and sets in the obvious manner. The integer n represents the next integer to be used as a 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. [10] for a rigorous definition of this concept). 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.

Given an initial *query* Q , a sequence of built-in and CHR constraints, an *initial execution state* is of the form $\langle Q, \emptyset, \mathbf{true}, \emptyset \rangle_1$.

Transition Rules Figure 2 lists the transition rules of ω_r . The execution stack \mathbb{A} is used to treat CHR constraints as function calls. The top-most element of \mathbb{A} is called the *active constraint*. When active, a CHR constraint performs a search for partner constraints that match the head of a rule. The constraint's occurrences are tried in a top-down, right-to-left order. To realize this order in ω_r , identified constraints on the execution stack are *occurrence*d in **Activate** and **Reactivate** transitions. When an occurrence'd 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 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 " $\rho @ H_k \setminus H_r \leftrightarrow G \mid B$ " (H_k or H_r may be empty) with $chr(K) = \theta(H_k)$, $chr(R) = \theta(H_r)$, 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 .

<p>1. Solve $\langle [b \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle S \text{ ++ } \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ if b is a built-in constraint. For the set of <i>reactivated constraints</i> $S \subseteq \mathbb{S}$, the following bounds hold: lower bound: $\forall H \subseteq \mathbb{S} : (\exists K, R : H = K \text{ ++ } R \wedge \exists \rho \in \mathcal{P} : \neg \text{appl}(\rho, K, R, \mathbb{B}) \wedge \text{appl}(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset)$ and upper bound: $\forall c \in S : \text{vars}(c) \not\subseteq \text{fixed}(\mathbb{B})$.</p>
<p>2. Activate $\langle [c \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle [c\#n:1 \mathbb{A}], \{c\#n\} \sqcup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$ if c is a CHR constraint (which has not yet been active or stored in \mathbb{S}).</p>
<p>3. Reactivate $\langle [c\#i \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle [c\#i:1 \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ if c is a CHR constraint (re-added to \mathbb{A} by a Solve transition but not yet active).</p>
<p>4. Simplify $\langle [c\#i:j \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B \text{ ++ } \mathbb{A}, K \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}' \rangle_n$ 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 $\text{apply}(\rho, K, R_1 \text{ ++ } [c\#i] \text{ ++ } R_2, \mathbb{B}, \theta) = B$. Let $t = (\rho, \text{id}(K \text{ ++ } R_1) \text{ ++ } [i] \text{ ++ } \text{id}(R_2))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.</p>
<p>5. Propagate $\langle [c\#i:j \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B \text{ ++ } [c\#i:j \mathbb{A}], \mathbb{S} \setminus R, \theta \wedge \mathbb{B}, \mathbb{T}' \rangle_n$ 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 $\text{apply}(\rho, K_1 \text{ ++ } [c\#i] \text{ ++ } K_2, R, \mathbb{B}, \theta) = B$. Let $t = (\rho, \text{id}(K_1) \text{ ++ } [i] \text{ ++ } \text{id}(K_2 \text{ ++ } R))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.</p>
<p>6. Drop $\langle [c\#i:j \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ if c has no j-th occurrence in \mathcal{P}.</p>
<p>7. Default $\langle [c\#i:j \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle [c\#i:j+1 \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ if the current state cannot fire any other transition.</p>

Figure 2: The transition rules of the refined operational semantics ω_r .

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 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 Execution proceeds by exhaustively applying transitions. Formally, an ω_r -derivation D is a sequence of ω_r states, with $D[1]$ an initial execution state for some query Q , and $D[i] \mapsto_{\mathcal{P}} D[i+1]$ a valid ω_r transition. We also use the notational abbreviation $\sigma_1 \mapsto_{\mathcal{P}}^* \sigma_n$ to denote a *finite* derivation $[\sigma_1, \dots, \sigma_n]$.

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 ω_r^* (ω_r or any of its refinements: see further) iff for any **Solve** transitions of the form $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle S \text{ ++ } \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ in any ω_r^* -derivation D the set of reactivated

```

:- 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).

```

Figure 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.

```

:- 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.

```

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

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 $\text{vars}(c) \subseteq \text{fixed}(\emptyset)$ (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* [13].

Example 2 The FIBBO handler depicted in Figure 3, performs a bottom-up computation of all Fibonacci numbers up to a given number. The constraint declarations¹ specify that all arguments are fixed instances of the host language’s `int` type (the ‘+’ mode declaration indicates a constraint’s argument is fixed).

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 Figure 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.

Using unbound, unguarded arguments to retrieve the outcome of a computation is very common in CHR. Under ω_r , with all these programs, constraints such as `fib/2` may be reactivated unnecessarily. In general, CHR constraints should only be reactivated if additional built-in constraints may cause more guards to become entailed. This is insufficiently specified in the ω_r semantics. To address this issue, we reintroduce the concept of *anti-monotonicity* [11, 21]:

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

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 $c(x_1, \dots, x_i, \dots, x_n)$ in $\text{HNF}(\mathcal{P})$, the guard of the corresponding rule is anti-monotone in $\{x_i\}$.

¹ The syntax is inspired by that of the K.U.Leuven CHR system [10, 12].

```

:- 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.

```

Figure 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.

A CHR program is always anti-monotone in fixed or 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 $\text{delay_vars}_{\mathcal{P}}(c)$ 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 “ $\forall c \in S : \text{delay_vars}_{\mathcal{P}}(c) \not\subseteq \text{fixed}(\mathbb{B})$ ”.*

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 [12, 19], already implement this refinement of ω_r (see also [11]). In the following two subsections, we prove that for non-reactive CHR rules the expensive maintenance of a propagation history can always be avoided.

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 Figure 5 computes the sum of a client’s account balances using a common CHR programming idiom to compute aggregates (see also [20]). The idiom uses a (typically non-reactive) propagation rule to generate a number of constraints, from which, after simplification to a single constraint, the result can be retrieved.*

Returning to the example: 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.

A common mistake is to assume that the history is thus superfluous for all non-reactive CHR rules. This is not the case, as shown by the following example:

Example 5 *Reconsider the FIBBO handler of Figure 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 with the same active constraint. We say these partner constraints *observe* the corresponding occurrence of the active constraint in ρ (cf. also [13]). Formally:

² See Appendix A for a formal proof that ω'_r is indeed an instance of ω_r . The appendix also specifies the difference between our definition of ω'_r and that of [11], which was wrong.

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 ω_r^* iff for all **Activate** or **Default** transitions of the form³:

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

($\mathbb{A}_0[1] = c\#i$ 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

$$\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_- \mapsto_{\mathcal{P}} \langle B ++ \mathbb{A}, \mathbb{S}', \mathbb{B}', \mathbb{T}' \rangle_- \mapsto_{\mathcal{P}}^* \langle \mathbb{A}, \mathbb{S}'', \mathbb{B}'', \mathbb{T}'' \rangle_-$$

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

Let ω_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 ω_r^\dagger is defined to be *duplicate-free*:

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

$$\text{if } \left\{ \begin{array}{l} \sigma_1 \mapsto_{\mathcal{P}} \sigma_2 \mapsto_{\mathcal{P}}^* \sigma'_1 \mapsto_{\mathcal{P}} \sigma'_2 \text{ is part of } D \\ \sigma_1 = \langle [c\#i:j|\mathbb{A}], \mathbb{S}, \dots \rangle_- \text{ and } \sigma'_1 = \langle [c\#i:j|\mathbb{A}], \mathbb{S}', \dots \rangle_- \\ \sigma_1 \mapsto_{\mathcal{P}} \sigma_2 \text{ is a } \mathbf{Propagate} \text{ transition applied with constraints } H \subseteq \mathbb{S} \\ \sigma'_1 \mapsto_{\mathcal{P}} \sigma'_2 \text{ is a } \mathbf{Propagate} \text{ transition applied with constraints } H' \subseteq \mathbb{S}' \\ \text{between } \sigma_2 \text{ and } \sigma'_1 \text{ no } \mathbf{Default} \text{ transition occurs of the form} \\ \sigma_2 \mapsto_{\mathcal{P}}^* \langle [c\#i:j|\mathbb{A}], \dots \rangle_- \mapsto_{\mathcal{P}} \langle [c\#i:j+1|\mathbb{A}], \dots \rangle_- \mapsto_{\mathcal{P}}^* \sigma'_1 \end{array} \right.$$

then $H \neq H'$.

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

Theorem 1 Define the mapping function α^\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 ω_r^\dagger derivation. Conversely, if D is an ω_r^\dagger derivation, then there exists an ω_r^* derivation D' such that $\alpha^\dagger(D) = D'$.

Proof. If D is an ω_r^* derivation, then $\alpha^\dagger(D)$ is clearly an ω_r^\dagger derivation.

For the reverse direction, let D be an ω_r^\dagger derivation, and D' the derivation obtained from D by adding the necessary tuples to the propagation history. That is, for each **Propagate** or **Simplify** transition in D of the form

$$\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B ++ \mathbb{A}, \mathbb{S}', \mathbb{B}, \mathbb{T} \rangle_n$$

the corresponding transition in D' becomes of the form

$$\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B ++ \mathbb{A}, \mathbb{S}', \mathbb{B}, \mathbb{T} \cup \{(\rho, I)\} \rangle_n \quad (1)$$

All **Propagate** and **Simplify** transitions in D' now have form (1). It suffices to show that for all these transitions $(\rho, I) \notin \mathbb{T}$ (note that we used \cup and not \sqcup , as the disjointness of the union is exactly what still needs to be shown).

Suppose there exist transitions of form (1) in D' for which $(\rho, I) \in \mathbb{T}$. Without loss of generality, we may consider the first such transition. Suppose the active constraint $c\#i:j$ matched the k 'th

³ We use ‘ \cdot ’ to denote that we are not interested in the identifier counter.

occurrence in ρ 's head. Then, clearly, this occurrence must be unobserved, and D' starts with the transition sequence

$$\begin{aligned}
D[1] \quad & \xrightarrow{\star}_{\mathcal{P}} \langle \mathbb{A}_0, \mathbb{S}_0, \mathbb{B}_0, \mathbb{T}_0 \rangle_{n_0} && (\mathbb{A}_0[1] \neq c\#i:j) \\
& \xrightarrow{\mathcal{P}} \langle [c\#i:j|\mathbb{A}'], \mathbb{S}_0, \mathbb{B}_0, \mathbb{T}_0 \rangle_{n_0} \\
& \xrightarrow{\star}_{\mathcal{P}} \langle [c\#i:j|\mathbb{A}'], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \\
& \xrightarrow{\mathcal{P}} \langle B \ ++ [c\#i:j|\mathbb{A}'], \mathbb{S}', \mathbb{B}, \mathbb{T} \cup \{(\rho, I)\} \rangle_n
\end{aligned}$$

with $[c\#i:j|\mathbb{A}'] = \mathbb{A}$, $I[k] = i$ and $(\rho, I) \in \mathbb{T}$. The sequence of transitions without this last invalid **Propagate** transition is the beginning of a valid ω'_r derivation. Therefore, by Definition 7, $(\rho, I) \notin \mathbb{T}_0$ (the j 'th occurrence of c is unobserved, and $I[k] = i$). Prior to the invalid **Propagate** transition, the non-reactive $c\#i:j$ active constraint repeatedly appears on top of the activation stack in a sequence of zero or more **Propagate** transitions:

$$\begin{aligned}
\dots \xrightarrow{\mathcal{P}} \langle [c\#i:j|\mathbb{A}], \mathbb{S}_0, \mathbb{B}_0, \mathbb{T}_0 \rangle_{n_0} & \xrightarrow{\mathcal{P}} \langle B_1 \ ++ [c\#i:j|\mathbb{A}], \mathbb{S}'_0, \mathbb{B}_0, \mathbb{T}_0 \cup \{(\rho, I_1)\} \rangle_{n_0} \\
& \xrightarrow{\star}_{\mathcal{P}} \langle [c\#i:j|\mathbb{A}], \mathbb{S}_1, \mathbb{B}_1, \mathbb{T}_1 \rangle_{n_1} \\
& \xrightarrow{\mathcal{P}} \langle B_2 \ ++ [c\#i:j|\mathbb{A}], \mathbb{S}'_1, \mathbb{B}_1, \mathbb{T}_1 \cup \{(\rho, I_2)\} \rangle_{n_1} \\
& \dots \\
& \xrightarrow{\mathcal{P}} \langle [c\#i:j|\mathbb{A}], \mathbb{S}_m, \mathbb{B}_m, \mathbb{T}_m \rangle_{n_m}
\end{aligned}$$

with $\mathbb{S}_m = \mathbb{S}$, $\mathbb{B}_m = \mathbb{B}$, $\mathbb{T}_m = \mathbb{T}$, and $n_m = n$. By Definition 7:

$$\forall i' \in [1, m] : \forall (\rho, I') \in \mathbb{T}_{i'} \setminus (\mathbb{T}_{i'-1} \cup \{(\rho, I_{i'})\}) : I'[k] \neq i'$$

and consequently (by induction):

$$\Theta = \{(\rho, I') \in \mathbb{T} \setminus \mathbb{T}_0 \mid I'[k] = i\} = \{(\rho, I_1), \dots, (\rho, I_m)\}$$

Because propagation is duplicate-free, $(\rho, I) \notin \Theta$. Consequently, as $I[k] = i$, $(\rho, I) \notin \mathbb{T} \setminus \mathbb{T}_0$. And we already showed that $(\rho, I) \notin \mathbb{T}_0$, so $(\rho, I) \notin \mathbb{T}$. By contradiction, no (first) transition of form (1) can exist with $(\rho, I) \in \mathbb{T}$. \square

3.2.1 Implementation

The main difficulty in the implementation of this optimization is deriving that a rule is unobserved (enforcing duplicate-free propagation is straightforward, as shown in Section 3.3). The abstract interpretation-based late storage analysis of [13], 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 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 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*

$$\langle [c\#i:j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\mathcal{P}} \langle \mathbb{A}', \mathbb{S}', \mathbb{B}', \mathbb{T} \sqcup \{(\rho, I_1 \ ++ [i] \ ++ I_2)\} \rangle_n \quad (2)$$

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 (2) were stored prior to the activation of $c' \# i'$. Also, in (2), $c \# i$ 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 (2). By contradiction, our assumption is false, and $i' < i$. \square

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

Theorem 2 *Define the mapping function α^\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 ω_r^\ddagger derivation. Conversely, if D is an ω_r^\ddagger derivation, then there exists an ω_r' derivation D' such that $\alpha^\ddagger(D) = D'$.

Proof. If D is an ω_r' derivation, then, by Lemma 1, $\alpha^\ddagger(D)$ is an ω_r^\ddagger derivation.

For the reverse direction, let D be an ω_r^\ddagger or ω_r^\ddagger derivation, and D' the derivation obtained from D by adding the necessary tuples to the propagation history, as described in the proof of Theorem 1. That is: all **Propagate** and **Simplify** transitions in D' have form (1). It suffices again to show that for all these transitions $(\rho, I) \notin \mathbb{T}$.

First, we show that Lemma 1 still holds for the derivation D . That is, for all transitions of D of form (1), if the active constraint matched the k 'th occurrence in ρ 's head, then $I[k] = \max(I)$. By definition of ω_r^\ddagger , this is true for the tuples that were not added to the history in the original derivation D . For those added in both D and D' , this also holds by definition of ω_r^\ddagger and Lemma 1.

Suppose, for some transition of form (1), that $(\rho, I) \in \mathbb{T}$, and that the active constraint matched the k 'th occurrence of ρ . Then $I[k] = \max(I)$. Moreover, when the (ρ, I) tuple was first added to the history, by uniqueness of constraint identifiers, the active constraint was the same constraint as active in the considered constraint. As propagation is duplicate-free in D , and the active constraint is non-reactive, this is not possible. \square

3.3.1 Implementation

The standard CHR compilation scheme (see e.g. [3, 10]) 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 *Figure 6(a) shows the generated code for the second occurrence of the `wp_to/1` constraint in Figure 3. For the query `wp_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.

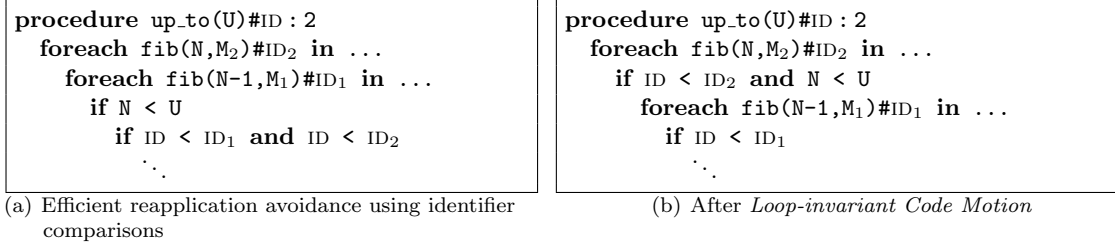


Figure 6: Pseudocode for the second occurrence of the `up_to/1` constraint of Figure 3.

3.3.2 Loop-invariant Code Motion

Most CHR compilers perform a so-called *Loop-invariant Code Motion* optimization to check guard entailment as soon as possible (e.g. ‘ $N < U$ ’ in Figure 6(b)). Contrary to a propagation history check, identifier comparisons enable additional code motion, as illustrated in Figure 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 can 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 are likely to cause reactive propagation rules to fire multiple times with the same combination. For constraint solvers, however, such additional rule applications generally 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 Figure 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 $live(\mathbb{T}, \mathbb{S}) = \{(\rho, I) \in \mathbb{T} \mid I \subseteq id(\mathbb{S})\}$, *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 ω_r^* iff for any state $\sigma = \langle [c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ in a ω_r^* 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 $\sigma \mapsto_{\mathcal{P}}^* \langle \mathbb{A}, \mathbb{S}'', \mathbb{B}'', \mathbb{T}'' \rangle_{n''}$ 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 ‘ $\mapsto_{\mathcal{P}}^*$ ’ denotes a finite derivation, telling duplicate idempotent CHR constraints does not affect termination either.

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 that was already possible prior to b ’s addition may be omitted from S . Many 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} .

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-free⁴ 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 $\mathcal{D}_{\mathcal{H}} \models \pi_{\text{vars}(\mathbb{B}) \cup \text{vars}(D[1])}(\mathbb{B}') \leftrightarrow \mathbb{B}$, $\mathbb{A}' = \mathbb{A}$, $\mathbb{S}' = \mathbb{S}$, and $\text{live}(\mathbb{T}', \mathbb{S}) = \text{live}(\mathbb{T}, \mathbb{S}) \setminus \{(\rho, I) \in \mathbb{T} \mid \rho \text{ is idempotent}\}$.

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 $\sigma_0 = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\mathcal{P}} \langle B ++ \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n = \sigma_1$. Because ρ ’s body B is idempotent, it follows from Definition 9 that the remainder of D' begins with $\sigma_1 \xrightarrow{\mathcal{P}} \sigma'_0 = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}', \mathbb{T}' \rangle_{n'}$, with $\mathcal{D}_{\mathcal{H}} \models \pi_{\text{vars}(\mathbb{B}) \cup \text{vars}(D[1])}(\mathbb{B}') \leftrightarrow \mathbb{B}$, and $\text{live}(\mathbb{T}', \mathbb{S}) = \text{live}(\mathbb{T}, \mathbb{S})$. 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. \square

For multi-headed propagation rules, reapplication is often cheaper than maintaining and checking a propagation history. The experimental results of Section 5 confirm this. 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:

$$\text{in}(X, L_1, U_1) \setminus \text{in}(X, L_2, U_2) \Leftrightarrow L_2 \leq L_1, U_2 \geq U_1 \mid \text{true}.$$

Here, ‘ $\text{in}(X, L, U)$ ’ 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 however an important difference 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.

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

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 $\triangleleft_1, \dots, \triangleleft_n$:*

1. *There exists a rule of the form “ $c(y_1, \dots, y_n) \setminus c(x_1, \dots, x_n) \Leftrightarrow G \mid true$.” in $HNF(\mathcal{P})$ with $\mathcal{D}_{\mathcal{H}} \models (x_1 \triangleleft_1 y_1 \wedge \dots \wedge x_n \triangleleft_n y_n) \rightarrow G$.*

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 $V = \{x_1, \dots, x_n\}$. For all removed occurrences of c/n in $HNF(\mathcal{P})$ that can be renamed to the form

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

(H_k, H_{r_1} , and H_{r_2} may be empty), such that $\neg \exists c(y_1, \dots, y_n) \in H_k \cup chr(B) : \mathcal{D}_{\mathcal{H}} \models G \wedge bi(B) \rightarrow (x_1 \triangleleft_1 y_1 \wedge \dots \wedge x_n \triangleleft_n y_n)$, 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 to the form*

$$“c(x_1, \dots, x_n) \Leftrightarrow G \mid B”, \text{ such that } bi(B) = B \text{ and } \mathcal{D}_{\mathcal{H}} \models \Phi \rightarrow (G \wedge B).$$

Let ρ' be the first such rule occurring in the $HNF(\mathcal{P})$ sequence.

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 $H_k ++ H_r = H_1 ++ [c(x_1, \dots, x_n)] ++ H_2$, such that either*

- $\mathcal{D}_{\mathcal{H}} \models \Phi \rightarrow \neg G$; or
- $H_r \subseteq [c(x_1, \dots, x_n)] \wedge (bi(B) = B) \wedge \mathcal{D}_{\mathcal{H}} \models (\Phi \wedge G) \rightarrow B$; or
- $\exists c(y_1, \dots, y_n) \in H_1 \cup H_2 : \mathcal{D}_{\mathcal{H}} \models (\Phi \wedge G) \rightarrow (x_1 \triangleleft_1 y_1 \wedge \dots \wedge x_n \triangleleft_n y_n)$.

Proof. Let \mathcal{P} be an arbitrary CHR program, and suppose that in this program the four conditions listed in Theorem 4 hold for CHR constraint c/n . Let D be an arbitrary ω_r^{idem} derivation of \mathcal{P} of the form

$$\langle Q, \emptyset, true, \emptyset \rangle_1 \mapsto_{\mathcal{P}}^* \sigma' = \langle [c' \setminus A'], S', B', T' \rangle'_n \mapsto_{\mathcal{P}}^* \sigma = \langle [c \setminus A], S, B, T \rangle_n$$

with c and c' CHR constraint of type c/n such that $\mathcal{D}_{\mathcal{H}} \models B \rightarrow c = c'$. Clearly, $D \ni \sigma' \mapsto_{\mathcal{P}} \langle [c' \setminus n' \setminus A], \{c' \setminus n'\} \cup S', B', T' \rangle_{n'+1}$, that is: at some point earlier in the derivation the duplicate constraint $c' \setminus n'$ was added to the constraint store. Suppose now that $\mathcal{D}_{\mathcal{H}} \models \pi_{\emptyset}(B)$ (the case with $\mathcal{D}_{\mathcal{H}} \models \neg \pi_{\emptyset}(B)$ is straightforward), and that $\sigma \mapsto_{\mathcal{P}} \sigma_0 = \langle [c \setminus n \setminus A], \{c \setminus n\} \cup S, B, T \rangle_{n+1} \mapsto_{\mathcal{P}}^* \langle A, S'', B'', T'' \rangle_{n''}$. Then it remains to be shown that $S'' = S$, $\mathcal{D}_{\mathcal{H}} \models \pi_{vars(B) \cup vars(D[1])}(B'') \leftrightarrow B$, and $live(T, S) = live(T'', S'')$. For the derivation starting from σ_0 , with active constraint $c \setminus n$, we consider two cases. Let \triangleleft denote the preorder on c/n constraints defined as $c(x_1, \dots, x_n) \triangleleft c(y_1, \dots, y_n) \leftrightarrow x_1 \triangleleft_1 y_1 \wedge \dots \wedge x_n \triangleleft_n y_n$, then:

Case A $\exists c'' \setminus n'' \in S : \mathcal{D}_{\mathcal{H}} \models B \rightarrow c \triangleleft c''$

Let ρ be the rule as defined in condition 1. Then $appl(\rho, [c'' \setminus n''], [c \setminus n], B)$, so the active constraint $c \setminus n$ is removed when it reaches rule ρ . By condition 2, none of the rules that are applied with $c \setminus n$ active before it is removed (by an application of ρ or earlier) remove or add any constraints. Therefore, no other CHR constraint becomes active before $c \setminus n$ is removed, and $\sigma_0 \mapsto_{\mathcal{P}}^* \langle A, S, B'', T'' \rangle_{n''}$ with $\pi_{vars(B) \cup vars(D[1])}(B'') \leftrightarrow B$ (only matching substitutions may be added to B), and $live(T, S) = live(T'', S)$ (all rules fired involve $c \setminus n$).

	SWI		JCHR			total # rules	n-headed propagation rules			
	history	non-react	history	non-react	non-react+		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)	<i>timeout</i>	<i>timeout</i>	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.

Case B $\neg\exists c''\#n'' \in \mathbb{S} : \mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow c \triangleleft c''$

Given this assumption, the $c''\#n''$ constraint must have been removed from the store at some point in the derivation. Moreover, this occurred when matching an occurrence that, after renaming, had form:

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

If for this rule the following holds

$$\exists c(y_1, \dots, y_n) \in H_k \cup \text{chr}(B) : \mathcal{D}_{\mathcal{H}} \models G \wedge \text{bi}(B) \rightarrow (x_1 \triangleleft_1 y_1 \wedge \dots \wedge x_n \triangleleft_n y_n) \quad (3)$$

then a constraint $c''\#n''$ with $\mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow c = c' \triangleleft c''$ existed in, or was added to the store \mathbb{S} at the moment $c''\#n''$ was removed. This constraint, in turn, must have been removed, so we apply the above reasoning again, only this time for $c''\#n''$ instead of $c''\#n'$.

After a finite number of times, $c''\#n'$, or one of its derived constraints $c''\#n''$, must have been removed by a rule for which formula (3) does not hold. For $V = \{x_1, \dots, x_n\}$, define $\Phi = \pi_V(G \wedge \text{bi}(B))$ as in the Theorem. The case where $\mathcal{D}_{\mathcal{H}} \models \Phi \leftrightarrow \text{false}$ is trivial, so assume $\mathcal{D}_{\mathcal{H}} \models \Phi \not\leftrightarrow \text{false}$. Then $\mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow \Phi$, and conditions 3 and 4 hold. By condition 3, $\exists \rho' \in \text{HNF}(\mathcal{P}) : \text{appl}(\rho', \emptyset, [c\#n], \Phi)$, and thus $\exists \rho' \in \text{HNF}(\mathcal{P}) : \text{appl}(\rho', \emptyset, [c\#n], \mathbb{B})$. This rule ρ' furthermore does not add CHR constraints, nor does it add built-in constraints that are not already entailed by \mathbb{B} . Condition 4 then ensures that none of the rules applied whilst $c\#n$ is active, and before the c constraint is removed (by an application of ρ' or earlier), remove or add CHR constraints, or add unentailed built-in constraints. Therefore, analogously to Case A, $\sigma_0 \mapsto_{\mathcal{P}}^* \langle \mathbb{A}, \mathbb{S}, \mathbb{B}'', \mathbb{T}'' \rangle_{n''}$ with $\text{live}(\mathbb{T}'', \mathbb{S}) = \text{live}(\mathbb{T}, \mathbb{S})$ and $\mathcal{D}_{\mathcal{H}} \models \pi_{\text{vars}(\mathbb{B}) \cup \text{vars}(D[1])}(\mathbb{B}'') \leftrightarrow \mathbb{B}$. \square

5 Evaluation

We implemented the optimizations introduced in this paper in the K.U.Leuven CHR system [10, 12] for SWI-Prolog, and in the K.U.Leuven JCHR system [19] for Java, and evaluated them using standard CHR benchmarks and constraint solvers. 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 JCHR⁵). If the history was eliminated using the optimization of Section 3.2, code motion is of course not applicable (N/A). Table 2 shows the results for the idempotence-based history elimination of Section 4.

Significant performance gains are measured. The selected benchmarks run about two times faster on average, and scale better as well. Even though no numbers are shown, it is moreover

⁵ 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)	<i>timeout</i>	<i>timeout</i>		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)	<i>timeout</i>	<i>timeout</i>		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.

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 (see e.g. Example 6). Using our optimizations, propagation histories consume no space at all. In extreme cases, this even improves the space complexity of the entire CHR handler.

6 Conclusions

6.1 Related Work

A preliminary version of this paper covering only Section 3.3 of the present paper appeared in [21]. 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 [3]. 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. [14]), [9] 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.

6.2 Conclusions

Whilst there is a vast 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 apparent discrepancy, and introduced 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 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 [4], and implemented them in two state-of-the-art CHR systems [12, 19]. 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] The Constraint Handling Rules (CHR) programming language homepage, 2008. <http://www.cs.kuleuven.be/~dtai/projects/CHR>.
- [2] B. Demoen and V. Lifschitz, editors. *ICLP '04: Proc. 20th Intl. Conf. Logic Programming*, volume 3132 of *LNCS*, Saint-Malo, France, September 2004. Springer.
- [3] Gregory J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Australia, December 2005.
- [4] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaaur. The refined operational semantics of Constraint Handling Rules. In Demoen and Lifschitz [2], pages 90–104.
- [5] Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
- [6] Thom Frühwirth. Programming with a Chinese horse. Invited Talk at 11th Intl. Conf., CP 2005, Sitges, Spain, October 2005. (slides).
- [7] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2008. To appear.
- [8] Christian Holzbaaur and Thom Frühwirth. A Prolog Constraint Handling Rules compiler and runtime system. volume 14(4) of *Journal of Applied Artificial Intelligence*, pages 369–388. Taylor & Francis, April 2000.
- [9] Beata Sarna-Starosta and C.R. Ramakrishnan. Compiling Constraint Handling Rules for efficient tabled evaluation. In M. Hanus, editor, *PADL '07: Proc. 9th Intl. Symp. Practical Aspects of Declarative Languages*, volume 4354 of *LNCS*, pages 170–184, Nice, France, January 2007. Springer.
- [10] Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, June 2005.
- [11] Tom Schrijvers and Bart Demoen. Antimonotony-based delay avoidance for CHR. Technical Report CW 385, K.U.Leuven, Dept. Comp. Sc., July 2004.
- [12] Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Th. Frühwirth and M. Meister, editors, *CHR '04, Selected Contributions*, pages 8–12, Ulm, Germany, May 2004.
- [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] Tom Schrijvers and David S. Warren. Constraint Handling Rules and tabled execution. In Demoen and Lifschitz [2], pages 120–136.
- [15] Jon Sneyers. *Optimizing compilation and computational complexity of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, 2008. To appear.

- [16] Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. 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.
- [17] Martin Sulzmann and Edmund S.L. Lam. Parallel execution of multi-set constraint rewrite rules. In S. Antoy, editor, *PPDP '08: Proc. 10th Intl. Conf. Princ. Pract. Declarative Programming*, pages 20–31, Valencia, Spain, July 2008. ACM Press.
- [18] Peter Van Weert. Optimization of CHR propagation rules. In *ICLP '08: Proc. 24rd Intl. Conf. Logic Programming*, LNCS, Udine, Italy, December 2008. Accepted.
- [19] Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In T. Schrijvers and Th. Frühwirth, editors, *CHR '05*, K.U.Leuven, Dept. Comp. Sc., Technical report CW 421, pages 47–62, Sitges, Spain, 2005.
- [20] Peter Van Weert, Jon Sneyers, and Bart Demoen. Aggregates for CHR through program transformation. In A. King, editor, *LOPSTR '07, Revised Selected Papers*, volume 4915 of *LNCS*, Kongens Lyngby, Denmark, 2008.
- [21] Peter Van Weert. A tale of histories. In T. Schrijvers, F. Raiser, and T. Frühwirth, editors, *CHR '08*, RISC Report Series 08-10, University of Linz, Austria, pages 79–94, Hagenberg, Austria, July 2008.
- [22] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In F. Mesnard and A. Serebenik, editors, *Proc. 13th Intl. Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, December 2003. System's homepage at <http://www.swi-prolog.org/>.

A On Anti-Monotony-based Delay Avoidance

In [11], the following version of the **Solve** transition is proposed:

1. **Solve**[†] $\langle [b|A], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightsquigarrow_{\mathcal{P}} \langle S ++ A, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ if b is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\forall c \in \mathbb{S} \setminus S : \exists V_1, V_2 : vars(c) = V_1 \cup V_2 \wedge V_1 \subseteq fixed(\mathbb{B}) \wedge$ all variables in V_2 appear only in arguments of c that are anti-monotone in \mathcal{P} .

Proposition 1 *Using our notation, this **Solve**[†] transition is equivalent to:*

1. **Solve**[‡] $\langle [b|A], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightsquigarrow_{\mathcal{P}} \langle S ++ A, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ if b is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\forall c \in \mathbb{S} \setminus S : delay_vars_{\mathcal{P}}(c) \subseteq fixed(\mathbb{B})$.

Proof. Let $c \in \mathbb{S} \setminus S$, with S defined as in **Solve**[†]. Then sets V_1 and V_2 exist, as defined in **Solve**[†]. By definition, $V_2 \subseteq vars(c) \setminus delay_vars_{\mathcal{P}}(c)$, and thus $V_2 \cap delay_vars_{\mathcal{P}}(c) = \emptyset$. Therefore, $delay_vars_{\mathcal{P}}(c) \subseteq V_1 \subseteq fixed(\mathbb{B})$.

Conversely, assume $c \in \mathbb{S} \setminus S$, with S defined as in **Solve**[‡]. Then the required sets V_1 and V_2 exist: simply take $V_1 = delay_vars_{\mathcal{P}}(c)$ and $V_2 = vars(c) \setminus V_1$. \square

In [11] the resulting semantics is shown to be correct with respect to the original refined operational semantics ω_r [4], where **Solve** is specified as:

1. **Solve**^{*} $\langle [b|A], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightsquigarrow_{\mathcal{P}} \langle S ++ A, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ where b is a built-in constraint and $S \subseteq \mathbb{S}$ such that $vars(\mathbb{S} \setminus S) \subseteq fixed(\mathbb{B})$.

That is, all constraints with at least one non-fixed argument have to be reactivated. The original specification of the ω_r semantics therefore prohibits any form of delay avoidance for non-fixed arguments, as illustrated by this example:

Example 9 *Consider the following CHR program:*

```

c(X) => X = 2, b.
c(_), a => true.
c(_), b => true.

```

For the query ‘ $a, c(X)$ ’ with X a free logical variable, **Solve**^{*} specifies that the $c(X)$ constraint has to be reactivated when ‘ $X = 2$ ’ is added to the built-in constraint solver, which leads to a final constraint store $\{b\#3\}$. This is the only final store allowed by the original refined semantics. However, as the program is clearly anti-monotone in c ’s argument, the **Solve**[‡] transition might not reactivate c , which then leads to an incorrect final constraint store $\{a\#1\}$.

This counterexample shows the proof in [11] must be wrong. The essential problem is that **Solve**^{*} specifies that constraints with non-fixed arguments have to be reactivated, even if the newly added built-in constraint does not enable any new matchings with them. This problem is not restricted to delay avoidance. It was first noted in [3, 10] in a different context:

Example 10 *Consider the following CHR program:*

```

c(X) => Y = 2, b.
c(_), a => true.
c(_), b => true.

```

For the query ‘ $a, c(X)$ ’ with X a free logical variable the original **Solve**^{*} transition specifies that the $c(X)$ constraint must be reactivated when ‘ $Y = 2$ ’ is added to the built-in constraint solver. The only final store allowed by the original refined semantics is thus $\{b\#3\}$. However, actual CHR implementations will not reactivate the $c(X)$ constraint, as the newly added ‘ $Y = 2$ ’ constraint does not affect X , the only variable occurring in $c(X)$.

Because the original refined operational semantics is thus inconsistent with the behavior of actual (Prolog) CHR implementations, a slightly more relaxed version of the **Solve** transition was defined in [3, 10]. This is also the version of ω_r we presented in Section 2.2. The following theorem shows that our definition of **Solve'** in Section 3 is correct with respect to this relaxed ω_r semantics:

Theorem 5 *Let \mathcal{P} be an arbitrary CHR program, and $\sigma = \langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ an arbitrary state with b a built-in constraint. If $\sigma \mapsto_{\mathcal{P}} \langle S ++ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ is a valid **Solve'** transition of ω'_r , then it is a valid ω_r **Solve** transition as well.*

Proof. By definition of **Solve'**, $S \subseteq \mathbb{S}$, and

- (1) $\forall c \in S : \text{delay_vars}_{\mathcal{P}}(c) \not\subseteq \text{fixed}(\mathbb{B})$,
- (2) $\forall H \subseteq \mathbb{S} : (H = K ++ R \wedge \exists \rho \in \mathcal{P} : \neg \text{appl}(\rho, K, R, \mathbb{B}) \wedge \text{appl}(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset)$.

As the lowerbound of the **Solve** transition in ω_r is also exactly (2), it suffices to prove that $\forall c \in S : \text{vars}(c) \not\subseteq \text{fixed}(\mathbb{B})$. This is obvious given (1), as by definition $\forall c : \text{delay_vars}_{\mathcal{P}}(c) \subseteq \text{vars}(c)$. \square

The optimized semantics of [11] on the other hand remains incorrect with respect to the relaxed ω'_r semantics. The reason is that the **Solve**[‡] transition only restricts the constraints that are not reactivated. The constraints that are reactivated, on the other hand, are not restricted:

Example 11 *Consider the following CHR program:*

```

c ⇒ X = 2, b.
c, a ⇔ true.
c, b ⇔ true.

```

For the query ‘a, c’ the **Solve** transition of Figure 2 specifies that the c constraint may not be reactivated when the ‘X = 2’ constraint is told. This leads to the only final store allowed by the ω_r semantics of Section 2.2, namely {b#3}. The **Solve**[‡] transition, however, allows the c’s reactivation. The resulting semantics thus may lead to an incorrect final constraint store {a#1}.

The final theorem show that our **Solve'** transition is indeed stronger than **Solve**[‡], since it never reactivates more constraints:

Theorem 6 *Let \mathcal{P} be a CHR program, and $\sigma = \langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ a state with b a built-in constraint. If $\sigma \mapsto_{\mathcal{P}} \langle S ++ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ is a valid **Solve**[‡] transition, and $\sigma \mapsto_{\mathcal{P}} \langle S' ++ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ a valid **Solve'** transition of ω'_r , then $S' \subseteq S$.*

Proof. By definition of **Solve**[‡]: $\forall c \in \mathbb{S} \setminus S : \text{delay_vars}_{\mathcal{P}}(c) \subseteq \text{fixed}(\mathbb{B})$, and by definition of **Solve'**: $S' \subseteq \mathbb{S} \wedge \forall c \in S' : \text{delay_vars}_{\mathcal{P}}(c) \not\subseteq \text{fixed}(\mathbb{B})$. Therefore clearly $(\mathbb{S} \setminus S) \cap S' = \emptyset$, and thus $(S' \subseteq \mathbb{S} \wedge S' \cap (\mathbb{S} \setminus S) = \emptyset) \rightarrow S' \subseteq S$. \square

B The Benchmarks

This section contains more information on the benchmarks used for Section 5. The benchmark programs are mostly standard CHR benchmarks or benchmarks created from standard CHR handlers found on the CHR Homepage [1]. Table 3 provides a description of all benchmarks used.

For several benchmarks pseudo-random sparse graphs of $\mathcal{O}(N)$ nodes and edges are generated. These graphs consist of a Hamiltonian cycle of N edges with weight 1 from node i to node $i + 1$ (and node N to node 1), and $3N$ random weight edges, 3 from every node to some randomly chosen other.

All benchmarks were performed on a Intel[®] Pentium[®] 4 (2.80GHz) with 1 GiB of RAM. SWI Prolog version 5.6.55 [22] was used, with a modified version of the K.U.Leuven CHR system [10, 12]. For the JCHR benchmarks, K.U.Leuven JCHR version 1.6.0 was modified. The (generated) Java code was compiled using Sun’s JDK 1.6.0, and executed using Sun’s HotSpot JVM 1.6.0.

<i>Name</i>	<i>Description, origin, author, ...</i>
$EQ(N)$	Classical CHR benchmark that solves a circular list of N equality constraints.
$FIBBO(N)$	Bottom-up computation of the N first Fibonacci numbers (see Figure 3; origin: [6]).
$FLOYD-WARSH(N)$	Finding the shortest path between all pairs of nodes of a sparse graph of $\mathcal{O}(N)$ nodes and edges using the Floyd-Warshall algorithm.
$INTERPOL(N)$	Linear interpolation of some points up to depth N . Author: Paolo Pilozzi.
$INTERVAL(N)$	Using Thom Frühwirth’s interval domain solver over integer and real, available on [1], to solve sequence of $\mathcal{O}(N)$ addition constraints over N variables.
$LEQ(N)$	Classic CHR benchmark that solves a circular list of N less-or-equal constraints. See Figure 1 for the LEQ handler.
$MANNERS(N)$	Port of a classic production rules benchmark called Miss Manners (original version available at http://www.cs.utexas.edu/ftp/pub/ops5-benchmark-suite/). Miss Manners is a constraint optimization problem that finds an optimal, acceptable seating arrangement for N guests at a dinner party, under certain given constraints.
$MINMAX(N)$	This benchmark uses a solver for inequality, minimum and maximum constraints on ground terms, written by Thom Frühwirth and Christian Holzbaaur (available on [1]). The benchmark consists of solving $N - 1$ maximum constraints over N integer variables.
$NSP(N)$	Finding the shortest path between all pairs of nodes of a sparse graph of $\mathcal{O}(N)$ nodes and edges using a naive shortest path algorithm of only three CHR rules. Constraint arguments are free logical variables.
$NSP_GRND(N)$	Variant of the above benchmark where constraint arguments are ground integers.
$SUM(N,M)$	Computes the sum of the balances of the accounts of N clients, where each client has M accounts with pseudo-random balances. See Example 4.
$TIMEPOINT(N)$	Using Frühwirth’s Christian Holzbaaur’s time point constraint handler, available at [1], to solve $1000N$ difference constraints over $1000N$ time points.
$TURING(N)$	Using a Turing machine simulator to execute a Turing program to copy a sequence of N consecutive cells. The simulator was written by Jon Sneyers, and will appear in [15].
$WFS(N)$	Computes the well-founded semantics of a simple 3-valued logic program (origin: [10, 12]).

Table 3: Description of the different benchmarks used for Section 5.