A Tale of Histories

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. A distinguishing feature of CHR are propagation rules. To avoid trivial non-termination, CHR implementations ensure a CHR rule is applied at most once with the same combination of constraints by maintaining a so-called propagation history. The performance impact of this history is often significant. We introduce two optimizations to reduce or even eliminate this overhead, and evaluate their implementation in two state-of-the-art CHR systems.

1 Introduction

Constraint Handling Rules (CHR) [4, 12] 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 [5, 7], Haskell, and Java [13].

An important, distinguishing feature of CHR are *propagation rules*. Unlike traditional 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.

The formal study of properties such as confluence and termination, led to the extension of CHR's operational semantics with a *token store* [4]. The token store contains a token for every constraint combination that may match a propagation rule. Each time a propagation rule is applied, the corresponding token is removed. Trivial non-termination is thus avoided by applying a propagation rule at most once with the same combination of constraints.

Practical implementations of CHR use the dual notion of a token store, called a *propagation history*; *history* for short [2, 5, 7, 13]. A history contains a tuple for each constraint combination that already fired a rule. A rule is only applied with some constraint combination, if the history does not contain the corresponding tuple. This is also reflected in more recent CHR operational semantics [3].

The implementation and optimization of propagation histories never received much attention [2, 7]. Our results show however that the propagation history can have a significant impact on both space and time performance. This paper

[⋆] Research Assistant of the Research Foundation– Flanders (FWO-Vlaanderen).

constitutes a first attempt to resolve this apparent discrepancy. We introduce two novel optimization techniques that either reduce or eliminate the overhead associated with propagation history maintenance

Contributions and Overview

- In Section 3 we explore the design space for the implementation of propagation histories. We show why implementing a history efficiently is challenging, and review some approaches taken by existing CHR systems. We then introduce an optimization for two-headed propagation rules.
- Section 4 introduces an innovative optimization that eliminates the need for maintaining a propagation history for all *non-reactive* CHR rules. This important class of CHR rules covers the majority of rules found in general-purpose CHR programs. We prove that the optimization is correct with respect to CHR's refined operational semantics [3].
- We implemented these optimizations in two state of the art CHR implementations, K.U.Leuven CHR [7, 9] for SWI-Prolog, and K.U.Leuven JCHR for Java [13]. Section 5 reports on the significant performance gains.

2 Preliminaries

To make this paper relatively self-contained, this section briefly reviews CHR's basic syntax and operational semantics. Gentler introductions are found for instance in [2, 4, 7].

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; a *constraint* $c(x_1, \ldots, x_n)$ is an atom constructed from these predicate symbols, and a list of arguments x_i, instances of data types offered by $\mathcal{H}$. Two classes of constraints exist: *built-in constraints*, solved by an underlying constraint solver of the host $\mathcal{H}$, and *CHR constraints*, handled by a CHR program. Many CHR systems support type and mode declarations for the arguments of CHR constraints. 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 name ρ uniquely identifies a rule. 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, the rule is a *propagation rule* and the symbol '$\Rightarrow$ ' is used instead of '$\Leftrightarrow$'. If both are non-empty, the rule is a *simpagation* rule. Either H_k or H_r has to be non-empty. The *guard* G is a conjunction of built-in constraints. If '$G \mid$ ' is omitted, it is considered to be '`true` $\mid$ '. The rule's *body* B, finally, is a conjunction of CHR and built-in constraints.

```
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.

Example 1. Fig. 1 shows a classic example CHR program, LEQ. It defines one CHR constraint, a less-than-or-equal constraint, using four CHR rules. All three kinds of rules are present. The constraint arguments are logical variables. The handler uses one built-in constraint, namely equality. If the *antisymmetry* rule is applied, its body adds a new built-in constraint to the built-in equality solver provided by the host environment. The body of the *transitivity* propagation rule adds a new CHR constraint, which is handled by the CHR program itself.

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

```
transitivity @ leq(X, Y), leq(Y1, Z) ⇒ Y = Y1 | leq(X, Z).
```

2.2 The Refined Operational Semantics

The behavior of most current CHR implementations is formally captured by the refined operational semantics [3], 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 [2, 7], and is a slight modification from the original specification [3].

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 follows: $\forall X, Y, Z : X = Y \sqcup Z \Leftrightarrow X = Y \cup Z \wedge Y \cap Z = \emptyset$. For a logical expression X, $vars(X)$ denotes the set of unquantified variables, and $\pi_V(X) \Leftrightarrow \exists v_1, \ldots, v_n : X$ with $\{v_1, \ldots, v_n\} = vars(X) \setminus V$. The meaning of built-in constraints is assumed determined by $\mathcal{D}_\mathcal{H}$, a consistent (first order logic) built-in constraint theory.

Execution States An execution state of ω_r is a tuple $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. The *execution stack* $\mathbb{A}$ is a sequence, used to treat constraints as procedure calls. Its function is explained in more detail below. The CHR constraint *store* $\mathbb{S}$ is a set of *identified* CHR constraints. An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with a unique *constraint identifier* i. The two connated mapping functions, $chr(c\#i) = c$ and $id(c\#i) = i$, are extended to sequences and sets in the obvious manner. The constraint identifiers are used to distinguish otherwise identical constraints ($chr(\mathbb{S})$ is a *multiset* of constraints). The counter n represents the next free CHR constraint identifier. The *built-in constraint store* $\mathbb{B}$ is

1. Solve $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle S \mathbin{++} \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ if b is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\forall c \in S : \mathit{vars}(c) \not\subseteq \mathit{fixed}(\mathbb{B})$ and $\forall H \subseteq \mathbb{S} : (\exists K, R : H = K \mathbin{++} R \wedge \exists \rho \in \mathcal{P} : \neg \mathit{appl}(\rho, K, R, \mathbb{B}) \wedge \mathit{appl}(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset)$.

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 $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle B \mathbin{++} \mathbb{A}, K \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ 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 $\mathit{appl}(\rho, K, R_1 \mathbin{++} [c\#i] \mathbin{++} R_2, \theta, \mathbb{B}) = B$.
Let $t = (\rho, \mathit{id}(K \mathbin{++} R_1) \mathbin{++} [i] \mathbin{++} \mathit{id}(R_2))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

5. Propagate $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle B \mathbin{++} [c\#i : j|\mathbb{A}], \mathbb{S} \setminus R, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ 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 $\mathit{appl}(\rho, K_1 \mathbin{++} [c\#i] \mathbin{++} K_2, R, \theta, \mathbb{B}) = B$.
Let $t = (\rho, \mathit{id}(K_1) \mathbin{++} [i] \mathbin{++} \mathit{id}(K_2 \mathbin{++} R))$, 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.

an abstract logical conjunction of built-in constraints, modeling all constraints passed to the underlying solvers. The *propagation history* $\mathbb{T}$, finally, is a set of tuples, each recording a sequence of constraint identifiers of CHR constraints that fired a rule, and the unique name of that rule.

Given an initial query Q, a sequence (conjunction) of built-in and host language constraints, an *initial execution state* is of the form $\langle Q, \emptyset, \mathtt{true}, \emptyset\rangle_1$.

Transition Rules The transition rules of ω_r are listed in Fig. 2. The top-most element of $\mathbb{A}$ is called the *active constraint.* Each newly added CHR constraint initiates a search for partner constraints that match the heads of the rules in an **Activate** transition. A built-in constraint is passed to the underlying constraint solver in a **Solve** transition. If the newly added built-in constraint may affect the outcome of guards, similar searches for applicable rules are initiated for the affected CHR constraints. Constraints whose variables are all fixed are never reactivated; formally:

Definition 1. *A variable v is* fixed *by a conjunction of constraints B, denoted $v \in \mathit{fixed}(B)$, if and only if $\mathcal{D}_{\mathcal{H}} \models \pi_{\{v\}}(B) \wedge \pi_{\{\theta(v)\}}(B) \rightarrow v = \theta(v)$ for arbitrary substitution θ.*

The order in which occurrences are traversed is fixed by ω_r: an active constraint tries its occurrences in a CHR program in a top-down, right-to-left order. To realize this order in ω_r, identified constraints on the execution stack are *occurrenced* (in **Activate** and **Reactivate** transitions). An *occurrenced* identified

CHR constraint $c\#i:j$ indicates that only matches with the j'th occurrence of c's constraint type are considered when the constraint is active.

Each active constraint traverses its different occurrences by a sequence of **Default** transitions, followed by a **Drop** transition. During this traversal all applicable rules are fired (i.e. **Propagate** and **Simplify** transitions). The applicability of a CHR rule is defined as follows:

Definition 2. *Given a conjunction of built-in constraints $\mathbb{B}$, a rule ρ is* applicable *with sequences of identified CHR constraints K and R, denoted $appl(\rho, K, R, \mathbb{B})$, if and only if a matching substitution θ exists for which $appl(\rho, K, R, \theta, \mathbb{B})$ is defined. The latter partial function is defined as $appl(\rho, K, R, \theta, \mathbb{B}) = B$ if and only if $K \cap R = \emptyset$ and, renamed apart, ρ is of the form (H_k or H_r may be empty):*

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

such that $chr(K) = \theta(H_k)$, $chr(R) = \theta(H_r)$ and $\mathcal{D_H} \models \mathbb{B} \rightarrow \pi_{vars(\mathbb{B})}(\theta \wedge G)$.

As with a procedure, when a rule fires, other constraints (its body) are executed, and execution does not return to the original active constraint *until* after these calls have finished. By putting the body on the activation stack, the different conjuncts of the body are solved (for built-in constraints) or activated (for CHR constraints) in a left-to-right order. This approach corresponds closely to that of the stack-based programming languages to which CHR is compiled.

Derivations Execution proceeds by exhaustively applying transitions. Formally, a derivation D is a sequence of states, with $D[1]$ a valid initial execution state for some query Q, and $D[i] \rightarrowtail_{\mathcal{P}} D[i+1]$ for all subsequent states $D[i]$ and $D[i+1]$. We also say these transitions $D[i] \rightarrowtail_{\mathcal{P}} D[i+1]$ are transitions of D. The common notational abbreviation $\sigma_1 \rightarrowtail^{\star}_{\mathcal{P}} \sigma_n$ denotes a *finite* derivation $[\sigma_1, \ldots, \sigma_n]$.

3 Propagation History Implementation

As stated also in [2, Section 4.3.4], a propagation history is very easy to implement naively, but quite challenging to implement efficiently. Obviously, tuples have to be stored in some efficient data structure, e.g. a balanced tree or a hash table. Naively implemented, tuples are only added to the propagation history, but never removed. Note that this is also the case in the ω_r formalism (cf. Section 2.2). This potentially leads to unbounded memory use.

The main challenge is thus to avoid this memory problem, with minimal overhead. All tuples referring to removed constraints are redundant. Formally, for a state $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$, these are all tuples not in $live(\mathbb{T}, \mathbb{S}) = \{(\rho, I) \in \mathbb{T} \mid I \subseteq id(\mathbb{S})\}$. Practice shows that eagerly removing redundant tuples after each constraint removal is not feasible due to time or space overheads. CHR implementations therefore commonly use ad-hoc garbage collection techniques, which may result in excessive memory use, but perform adequately in practice.

One technique is to lazily remove redundant tuples during history checks (see [2]). A second technique is denoted *distributed propagation history* maintenance (see [7]). With this technique, no global propagation history is maintained. Instead, the runtime representation of each individual CHR constraint contains (a subset of) the history tuples they occur in. When a constraint is removed, the corresponding part of the propagation history is thus removed as well. Both techniques could easily be combined.

We refer to [2, 7] for some more details on the implementation of propagation histories in current CHR systems. Many design choices, however, are not fully covered by these theses:

- Is one global history maintained, or one history per rule?
- Is the distributed history information stored in all constraints of the matching combination, or only in one of the partners? In the latter case, is the active constraint used, or the constraint matching some fixed occurrence?
- In which cases are more eager garbage collection techniques feasible?
- How to exploit functional dependencies?

In the following subsection we introduce an improved technique to maintain the propagation history of two-headed propagation rules.

3.1 Two-headed Propagation Rules

For two-headed propagation rules, a distributed propagation history can be implemented more efficiently. Assume that, if there are multiple propagation rules, a separate history is maintained per rule, as is the case e.g. in the K.U.Leuven JCHR system [13]. By default, history tuples for a two-headed rule contain two constraint identifiers. It is however more efficient to simply store, in each constraint, the identifiers of all partner constraints it fired with whilst active. This avoids the creation of tuples, and allows for more efficient hash tables. We refer to Section 5 for empirical results.

Care must be taken when both heads are occurrences of the same constraint type, as for instance in the *transitivity* rule of Example 1. One possibility is to maintain a separate history per occurrence. Another trick is to use negated constraint identifiers if the the active constraint matches one of the occurrences.

With a similar reasoning, a reduction of the tuple size for all propagation rules is possible. Experiments only showed negligible performance gains though.

4 Non-reactive CHR Rules

In this section we consider *non-reactive CHR rules*, i.e. rules that are never matched by a reactivated CHR constraint. We will show that for this important class of rules no propagation history has to be maintained.

Example 2. Consider the following common CHR pattern to compute the sum of the arguments of `elem/1` constraints:

```
sum, elem(X) ⇒ sum(X).
sum ⇔ true.
sum(X), sum(Y) ⇔ sum(X+Y).
```

If the type or mode declarations of the `elem/1` constraint specify that its argument is always fixed, say a (ground) integer value, then `elem/1` constraints are never reactivated under ω_r. As the `sum/0` constraint is clearly also never reactivated, the first rule is thus never matched by a reactivated CHR constraint.

Formally, non-reactive CHR constraints and rules are defined as follows:

Definition 3. *A CHR constraint type c/n is* non-reactive *in a program $\mathcal{P}$ under a refined operational semantics $\omega_r^\star$ if and only if for all $\omega_r^\star$ derivations D with that program, for all **Solve** transitions in D of the form*

$$\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$$

the set of reconsidered constraints S does not contain constraints of type c/n. A CHR rule $\rho \in \mathcal{P}$ is non-reactive *if and only if all constraint types that occur in its head are non-reactive in $\mathcal{P}$.*

Under the ω_r semantics as defined in Section 2.2, only *fixed*, or *ground*, CHR constraint types are non-reactive. Formally, a CHR constraint type c/n is fixed iff for all CHR constraints c' of type c/n, $vars(c') \subseteq fixed(\emptyset)$ (see Definition 1). A CHR compiler derives which constraints are fixed from their mode declarations, or using static groundness analysis [10]. Both constraints in Example 2, for instance, are fixed.

A substantially larger class of CHR programs, however, can be made non-reactive by a slight modification of the refined operational semantics.

Example 3. Suppose the type or mode information implies the first argument of `fib/2` constraints is always fixed. The second argument on the other hand can be a free (logical) variable:

```
fib(N,M₁) \ fib(N,M₂) ⇔ M₁ = M₂.
fib(N,M) ⇒ N ≤ 1 | M = 1.
fib(N,M) ⇒ N > 1 | fib(N-1,M₁), fib(N-2,M₂), M = M₁ + M₂.
```

For this handler, a `fib/2` constraint does not have to be reactivated when a built-in constraint is added. Indeed: because there are no guards on this argument, no additional rules become applicable by constraining it further.

Using constraints unbound, unguarded arguments to retrieve computation results is very common in CHR. These constraints should not be reactivated. Unfortunately, this is insufficiently specified in the standard ω_r semantics. We therefore propose a semantical refinement, based on the concept of *anti-monotonicity* [8]. Anti-monotonicity generalizes both fixed and unguarded constraint arguments:

Definition 4. *A conjunction of built-in constraints B is* anti-monotone *in a set of variables V if and only if:*

$$\forall B_1, B_2 : (\pi_{vars(B)\setminus V}(B_1 \wedge B_2)) \Leftrightarrow (\pi_{vars(B)\setminus V}(B_1)) \\ \Rightarrow (\mathcal{D_H} \not\models B_1 \rightarrow B) \Rightarrow (\mathcal{D_H} \not\models B_1 \wedge B_2 \rightarrow B)$$

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 $\{x_i\}$.*

Based on these definitions, the *anti-monotony-based delay avoidance* optimization reduces the amount of needlessly reactivated constraints [8]. Concretely, let $delay\_vars_{\mathcal{P}}(c)$ denote the set of variables that occur in the arguments of an (identified) CHR constraint c in which $\mathcal{P}$ is not anti-monotone, then the **Solve** transition of ω_r (cf. Fig. 2) can be replaced with:

> **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 and $S \subseteq \mathbb{S}$ such that $\forall c \in S$: $\mathbf{delay\_vars}_{\mathcal{P}}(c) \not\subset \mathit{fixed}(\mathbb{B})$ and $\forall H \subseteq \mathbb{S} : (\exists K, R : H = K +\!\!+ R \wedge \exists \rho \in \mathcal{P} : \neg appl(\rho, K, R, \mathbb{B}) \wedge appl(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset)$.

The resulting semantics, denoted ω_r', is an instance of ω_r[1].

Clearly, the following properties hold for any CHR program $\mathcal{P}$:

– If the CHR constraint type c/n is fixed, i.e. if c/n is non-reactive in $\mathcal{P}$ under ω_r, then $\mathcal{P}$ is anti-monotone in all n arguments of c/n.
– If $\mathcal{P}$ is anti-monotone in all n arguments of c/n, then that CHR constraint type is non-reactive in $\mathcal{P}$ under ω_r'.

We now show how the maintenance of a propagation history for non-reactive CHR rules can be avoided. The central observation is that when a non-reactive rule is fired, the active constraint is 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 derivation with this program. Then for each* **Simplify** *or* **Propagate** *transition in D of the form*

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

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 2 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 (cf. supra), this includes the application of ρ in (1). By contradiction, our assumption is false, and $i' < i$. □

[1] The **Solve'** transition presented here differs from the one proposed in [8]. As shown in Appendix A, the latter version is not entirely correct. The appendix further provides a correctness proof for our version, and shows that it is stronger than that of [8].

Let ω_r'' denote the semantics obtained by replacing the phrase

> Let $t = (\rho, id(H_1) ++ [i] ++ id(H_2))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

in the **Simplify** and **Propagate** transitions of ω_r' with

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

To avoid trivial non-termination where the same combination of constraints fires a propagation rule infinitely many times, we also assume the following property to hold for ω_r'':

Definition 6 (Duplicate-free Propagation). *For all derivations D of a CHR program $\mathcal{P}$ where the j'th occurrence of c is kept, if the following holds:*
- $\sigma_1 \rightarrowtail_{\mathcal{P}} \sigma_2 \rightarrowtail^{\star}_{\mathcal{P}} \sigma_1' \rightarrowtail_{\mathcal{P}} \sigma_2'$ *is part of D*
- $\sigma_1 = \langle [c\#i : j|\mathbb{A}], \mathbb{S}, \ldots \rangle_{\_}$ *and* $\sigma_1' = \langle [c\#i : j|\mathbb{A}], \mathbb{S}', \ldots \rangle_{\_}$
- $\sigma_1 \rightarrowtail_{\mathcal{P}} \sigma_2$ *is a* ***Propagate*** *transition applied with constraints $H \subseteq \mathbb{S}$*
- $\sigma_1' \rightarrowtail_{\mathcal{P}} \sigma_2'$ *is a* ***Propagate*** *transition applied with constraints $H' \subseteq \mathbb{S}'$*
- *between σ_2 and σ_1' no* ***Default*** *transition occurs of the form*
$$\sigma_2 \rightarrowtail^{\star}_{\mathcal{P}} \langle [c\#i : j|\mathbb{A}], \ldots \rangle_{\_} \rightarrowtail_{\mathcal{P}} \langle [c\#i : j+1|\mathbb{A}], \ldots \rangle_{\_} \rightarrowtail^{\star}_{\mathcal{P}} \sigma_1'$$

then $H \neq H'$.

This property, in combination with Lemma 1, allows us to show that ω_r' and ω_r'' are equivalent:

Theorem 1. *Define the mapping function α as follows:*
$$\alpha(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \{(\rho, I) \in \mathbb{T} \mid \rho \textit{ is a reactive CHR rule}\} \rangle_n$$
If D is an ω_r' derivation, then $\alpha(D)$ is an ω_r'' derivation. Conversely, if D is an ω_r'' derivation, then there exists an ω_r' derivation D' such that $\alpha(D) = D'$.

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

For the reverse direction, let D be an ω_r'' 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

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

the corresponding transition in D' becomes of the form

$$\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle B ++ \mathbb{A}, \mathbb{S}', \mathbb{B}, \mathbb{T} \cup \{(\rho, I)\} \rangle_n \tag{2}$$

We treat the history $\mathbb{T}$ to be a multiset here, because otherwise possible duplicates would disappear unnoticed. All **Propagate** and **Simplify** transitions in D' now have form (2). It suffices 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 (2), if the active constraint matched the k'th occurrence in ρ's head, then $I[k] = max(I)$. By definition of ω_r'', this is true for the tuples

```
procedure up_to(U)#ID : 2
  foreach fib(N,M_2)#ID_2 in ...
    foreach fib(N-1,M_1)#ID_1 in ...
      if N < U
        if ID < ID_1 and ID < ID_2
          ⋱
```

(a) Unoptimized

```
procedure up_to(U)#ID : 2
  foreach fib(N,M_2)#ID_2 in ...
    if ID < ID_2 and N < U
      foreach fib(N-1,M_1)#ID_1 in ...
        if ID < ID_1
          ⋱
```

(b) After *Loop Invariant Code Motion*

Fig. 3. Pseudocode for the second occurrence of the `up_to/1` constraint of Example 4.

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'' and Lemma 1.

Suppose, for some transition of form (2), 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. $\square$

This theorem shows the correctness of replacing the propagation history of non-reactive CHR rules with more efficient constraint identifier comparisons. The next subsection shows how this optimization can be implemented in typical ω_r-based CHR implementations.

4.1 Implementation and Further Optimizations

The standard CHR compilation scheme (see e.g. [2, 5, 7]) generates, for each occurrence, a nested iteration that looks for matching partner constraints for the active constraint. If the active constraint is not removed, all partner constraint iterators are suspended until the body is fully executed. Afterwards, the nested iteration is simply resumed to find more matching combinations.

Example 4. The following handler, called FIBBO, performs a bottom-up computation of all Fibonacci numbers up to a given number (all arguments are fixed):

```
up_to(U) ⇒ fib(0,1), fib(1,1).
up_to(U), fib(N-1,M_1), fib(N,M_2) ⇒ N < U | fib(N+1,M_1+M_2).
```

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. When, finally, the `up_to(U)` constraint reaches its second occurrence, some mechanism is required to prevent the second rule to propagate everything all over again.

A propagation history would require $\mathcal{O}(U)$ space. Because all constraints are non-reactive, however, no propagation history is maintained. Instead, constraint identifiers are simply compared. Fig. 3(a) shows the generated code for the second occurrence of the `up_to/1` constraint.

	SWI		JCHR	
	tree	2-hash	hash	2-hash
EQ(35)	3,465	N/A[2]	47	37 (79%)
LEQ(70)	3,806	2,866 (75%)	85	65 (76%)

Table 1. Benchmark results for the EQ and LEQ benchmarks.

	SWI		JCHR		
	tree	non-react	hash	non-react	non-react+
WFS(200)	2,489	2,143 (86%)	71	67 (94%)	67 (94%)
FIBBO(1000)	15,929	4,454 (28%)	70	67 (95%)	21 (30%)
FIBBO(2000)	61,290	17,704 (29%)	206	275 (133%)	90 (44%)
FIBBO(3000)	*timeout*	*timeout*	542	464 (85%)	153 (28%)

Table 2. Benchmark results for the WFS and FIBBO benchmarks.

If none of the iterators return candidate partner constraints more than once, propagation is guaranteed to be duplicate-free (see Definition 6). Most iterators used by CHR implementations obey this property. If not, a temporary history can be maintained whilst the active constraint is considering an occurrence.

Loop-invariant Code Motion Lemma 1 not only applies to propagation rules, but also to simplification and simpagation rules. Whilst maintaining a history for non-propagation rules is pointless, comparing partner constraint identifiers is not. As shown in Fig. 3(b), the standard *Loop-invariant Code Motion* optimization can be extended to include not only guards (e.g. `N < U`), but also identifier comparisons. For multi-headed CHR rules — including simplification and simpagation rules — this may considerably prune the search space of candidate partner constraints. Moreover, if an iterator returns constraints in ascending order of identifiers, the corresponding (nested) iteration can be stopped early.

5 Evaluation

We implemented the optimizations presented in this paper in the K.U.Leuven CHR system [7, 9] for SWI-Prolog, and in the K.U.Leuven JCHR system [13] for Java. The benchmark results are given in Tables 1 and 2. For each system, the first column gives the reference timings: for SWI this is a tree-based propagation history, for JCHR a hash-based history. Both systems use distributed history maintenance (see Section 3). The *2-hash* and *non-react* columns give timings using the optimization for two-headed propagation rules given in Section 3, and the optimization for non-reactive CHR rules of Section 4 respectively. For the *non-react+* measurements the non-reactiveness optimization was combined with loop invariant code motion (currently only implemented in JCHR).

[2] In the current SWI implementation, the history of a two-headed propagation rule is only optimized if there are no other propagation rules in the program. In JCHR, this is not relevant, as JCHR maintains a separate history per rule.

For both optimizations significant performance gains are measured. Note that the improved timings in Table 1 for the SWI-Prolog system may be due to moving from a tree-based history to a hash-based one. For JCHR, however, this is definitely not the case, showing the relevance of the improved data structure. Table 2 contains one surprising timing for the FIBBO(N) benchmark in JCHR: even though identifier comparisons are cheaper than checking a propagation histories, for $N = 2000$, the performance nevertheless worsened. Detailed profiling showed that this is due to unpredictable behavior of the JVM's garbage collector.

For non-reactive rules, space complexity is furthermore optimal: propagation histories no longer consume space at all. The complexity for the history of the FIBBO handler, for instance, is improved from linear to constant (see Example 4).

6 Related Work, Conclusions and Future Work

Related Work The propagation history contributes to significant performance issues when implementing CHR in a tabling environment [11]. Based on a similar approach explored in [11], an alternative CHR semantics is proposed in [6]. Being set-based, this semantics addresses the trivial non-termination problem without the use of a propagation history. It would be interesting to see whether these results can be transferred to CHR without abandoning its common multiset semantics (see also Future Work).

In [2], a simple analysis is presented to eliminate the propagation histories for certain fixed CHR constraints. Advanced CHR systems such as [9, 13] implement more powerful versions of this analysis, extended towards non-reactive CHR constraints, or made more accurate by abstract interpretation [10]. Our results in Section 4, however, considerably reduce the benefits of these complex analyses, as comparing constraint identifiers is much cheaper than maintaining a history.

Conclusions We showed that maintaining a propagation history comes at a considerable runtime cost, both in time and in space. We introduced two optimizations to reduce or eliminate this overhead. We showed that for two-headed propagation rules more efficient data structures can be used. This is interesting, as rules with more than two heads are relatively rare. We then argued that non-reactive CHR propagation rules do not require the maintenance of a propagation history. Instead, cheap constraint identifier comparisons can be used. Furthermore, these comparisons can be moved early in the generated nested loops, thus pruning the search space of possible partner constraints. We formally proved the correctness of the optimization for non-reactive rules with respect to CHR's refined operational semantics. We implemented both optimizations, and observed significant performance gains.

Future Work For reactive CHR rules a propagation history still has to be maintained. This includes the rules of most true constraint solvers. Most constraint solvers though, such as the archetypal LEQ handler of Example 1, have set semantics. As argued by [6] (see *Related Work* paragraph), if constraints have set semantics, a propagation history is less compelling. Under the refined operational semantics, however, set semantics alone does not suffice to justify the elimination of propagation histories, that is without affecting a program's semantics. A stronger property called *idempotence* is required. We are currently developping an analysis to derive this property, and have already observed promising performance improvements for several programs. For certain programs, an automated confluence analysis (see e.g. [4]) would be useful, as rules that remove duplicate constraints may be moved to the front of a confluent CHR program.

Acknowledgements 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 for their useful comments.

References

1. B. Demoen and V. Lifschitz, editors. *ICLP '04: Proc. 20th Intl. Conf. Logic Programming*, volume 3132 of *LNCS*, Saint-Malo, France, September 2004. Springer.
2. Gregory J. Duck. *Compilation of Constraint Handling Rules.* PhD thesis, University of Melbourne, Australia, December 2005.
3. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In Demoen and Lifschitz [1], pages 90–104.
4. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
5. Christian Holzbaur 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.
6. 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.
7. Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules.* PhD thesis, K.U.Leuven, Belgium, June 2005.
8. Tom Schrijvers and Bart Demoen. Antimonotony-based delay avoidance for CHR. Technical Report CW 385, K.U.Leuven, Dept. Comp. Sc., July 2004.
9. 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.
10. 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.
11. Tom Schrijvers and David S. Warren. Constraint Handling Rules and tabled execution. In Demoen and Lifschitz [1], pages 120–136.
12. 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.
13. 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.

A On Anti-Monotony-based Delay Avoidance

In [8], the following version of the **Solve** transition is proposed:

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 and $S \subseteq \mathbb{S}$ such that $\forall c \in \mathbb{S} \setminus S : \exists V_1, V_2 : \mathit{vars}(c) = V_1 \cup V_2 \wedge V_1 \subseteq \mathit{fixed}(\mathbb{B}) \wedge$ 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‡ $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ if b is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\forall c \in \mathbb{S} \setminus S : \mathit{delay\_vars}_{\mathcal{P}}(c) \subseteq \mathit{fixed}(\mathbb{B})$.

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, $V_2 \subseteq \mathit{vars}(c) \setminus \mathit{delay\_vars}_{\mathcal{P}}(c)$, and thus $V_2 \cap \mathit{delay\_vars}_{\mathcal{P}}(c) = \emptyset$. Therefore, $\mathit{delay\_vars}_{\mathcal{P}}(c) \subseteq V_1 \subseteq \mathit{fixed}(\mathbb{B})$.

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 $V_1 = \mathit{delay\_vars}_{\mathcal{P}}(c)$ and $V_2 = \mathit{vars}(c) \setminus V_1$. □

In [8] the resulting semantics is shown to be correct with respect to the original refined operational semantics ω_r [3], where **Solve** is specified as:

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

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 5. 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 [8] 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 [2, 7] in a different context:

Example 6. 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 $\{\texttt{b}\#3\}$. 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 [2, 7]. 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 4 is correct with respect to this relaxed ω_r semantics:

Theorem 2. *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 $\sigma \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ 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) $\forall c \in S : \mathit{delay\_vars}_{\mathcal{P}}(c) \not\subset \mathit{fixed}(\mathbb{B})$,
(2) $\forall H \subseteq \mathbb{S} : (H = K +\!\!+ R \wedge \exists \rho \in \mathcal{P} :$
$$\neg \mathit{appl}(\rho, K, R, \mathbb{B}) \wedge \mathit{appl}(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset).$$

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

The optimized semantics of [8] 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 7. Consider the following CHR program:

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

For the query '`a, c`' the **Solve** transition of Fig. 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 $\{\texttt{b}\#3\}$. The **Solve**‡ transition, however, allows the `c`'s reactivation. The resulting semantics thus may lead to an incorrect final constraint store $\{\texttt{a}\#1\}$.

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

Theorem 3. *Let $\mathcal{P}$ be a CHR program, and $\sigma = \langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ astate with b a built-in constraint. If $\sigma \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ is a valid **Solve**‡ transition, and $\sigma \rightarrowtail_{\mathcal{P}} \langle S' +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ a valid **Solve'** transition of ω_r', then $S' \subseteq S$.*

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