

Guard Reasoning in the Refined Operational Semantics of CHR

Jon Sneyers*, Tom Schrijvers**, and Bart Demoen

Dept. of Computer Science, K.U.Leuven, Belgium
{jon,toms,bmd}@cs.kuleuven.be

Abstract. Constraint Handling Rules (CHR) is a high-level programming language based on multi-headed guarded rules. The original high-level operational semantics of CHR is very nondeterministic. Recently, instantiations of the high-level operational semantics have been proposed and implemented, removing sources of nondeterminism and hence allowing better execution control. Rule guards may be redundant under a more instantiated semantics while being necessary in the general high-level semantics. Expert CHR programmers tend to remove such redundant guards. Although this tends to improve the performance, it also destroys the local logical reading of CHR rules: in order to understand the meaning of a rule, the entire program and the details of the instantiated operational semantics have to be taken into account. As a solution, we propose compiler optimizations that automatically detect and remove redundant guards.

1 Introduction

Constraint Handling Rules (CHR) is a high-level multi-headed rule-based programming language extension commonly used to write constraint solvers. We assume that the reader is familiar with the syntax and semantics of CHR, referring to [6, 17] for an overview. Although examples are given for CHR(Prolog), the optimizations can be applied in any host language.

The (original) theoretical operational semantics (ω_t) of CHRs, as defined in [6], is nondeterministic. For instance, the order in which rules are applied is not specified. All implementations of CHR are an instantiation of the ω_t semantics of CHRs. Any of these instantiations are completely deterministic, since they compile CHR programs to (eventually) instructions of a deterministic RAM machine. Although for any given (version of a) CHR system the execution strategy is completely fixed, only a partial instantiation of ω_t is guaranteed by the developers of CHR compilers. In other words, most CHR systems exclude many

* This work was partly supported by project G.0144.03 funded by the Research Foundation - Flanders (F.W.O.-Vlaanderen). Jon Sneyers is currently funded by Ph.D. grants of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

** Post-Doctoral Researcher of the Research Foundation Flanders (FWO-Vlaanderen).

choices allowed by ω_t but at the same time they do not completely specify their exact deterministic operational semantics since it may change in later versions or with certain compiler optimizations switched on or off.

Most CHR systems instantiate the *refined* operational semantics (ω_r) [5] of CHR. In ω_r , the concept of an *active* constraint is introduced: its occurrences are tried top-to-bottom, removed-before-kept, left-to-right, while rule bodies are executed depth-first and left-to-right. In a sense, this generalizes the standard Prolog execution strategy. Recently, other (partial) instantiations of the ω_t semantics have been proposed; most notably, the priority semantics ω_p [3].

Experienced CHR programmers know the operational semantics specified by the CHR system they use. They take that knowledge into account to improve the performance of their program. However, the resulting CHR programs may well be no longer correct in all ω_t execution strategies. The dilemma experienced CHR programmers face is the following: either they make sure their programs are valid under ω_t semantics, or they write programs that only work correctly under a more instantiated operational semantics. The former may result in a performance penalty, while the latter results in a program for which the logical reading of the rules is no longer clear. CHR rules have a (linear or classical) logic reading which is local: a rule always has the same reading, whatever its context. When the specifics of the operational semantics are implicitly assumed, the locality of the logic reading is lost. For instance, under ω_r semantics, CHR programmers often omit the rule guards that are implicitly entailed by the rule order. In this work we show how to overcome this dilemma.

Automatic code generation and source-to-source transformations are typically implemented by applying a general scheme. Such approaches often introduce many redundant guards or redundant rules. Again, guard simplification and occurrence subsumption can be used to improve the output code. By allowing the user to declare background knowledge about the CHR constraints and host-language predicates that are used, even more redundant code can be avoided.

Our contributions are as follows:

1. We formalize the implicit pre-conditions of a constraint occurrence in the refined operational semantics (see Section 4). Our formalization not only considers the rules in the program, but also user-provided declarations for types, modes and general background knowledge (see also Section 3).
2. We establish the usefulness of these pre-conditions for optimized compilation with two program transformations: *guard simplification* and *occurrence subsumption* (see Sections 3 and 4).
3. We describe our implementation of these optimizations (see Section 6), and the common component for entailment checking (see Section 5). The implementation is available in the K.U.Leuven CHR System.
4. Experimental evaluation shows that our optimizations yield compact and efficient code (see Section 7).
5. We sketch a similar approach for the priority semantics (see Section 8).

This paper is a revised and extended version of [16] and [15]. In the next section we briefly introduce CHR and its formal semantics. The body of this paper is structured as outlined above. Finally, we conclude in Section 9 with a discussion of related and future work.

2 Constraint Handling Rules

We use $[H|T]$ to denote the first (H) and remaining elements (T) of a sequence, $++$ for sequence concatenation, ϵ for empty sequences, \uplus for multiset union, and $\underline{\subseteq}$ for multiset subset. We shall sometimes omit existential quantors to get a lighter notation. Constraints are either CHR constraints or *built-in* constraints in some constraint domain \mathcal{D} . The former are manipulated by the CHR execution mechanism while the latter are handled by an underlying constraint solver. We will assume this underlying solver supports at least equality, **true** and **fail**. We consider all three types of CHR rules to be special cases of simpagation rules:

Definition 1 (CHR program). A CHR program P is a sequence of CHR rules R_i of the form

$$(\text{rulename } @) \ H_i^k \setminus H_i^r \iff g_i \mid B_i$$

where H_i^k (kept head constraints) and H_i^r (removed head constraints) are sequences of CHR constraints with $H_i^k ++ H_i^r \neq \epsilon$, g_i (guard) is a conjunction of built-in constraints, and B_i (body) is a conjunction of constraints. We will write H_i as a shorthand for $H_i^k ++ H_i^r$.

If H_i^k is empty, then the rule R_i is a *simplification* rule. If H_i^r is empty, then R_i is a *propagation* rule. Otherwise the rule is a *simpagation* rule. We assume all arguments of the CHR constraints in H_i to be unique variables, making any head matchings explicit in the guard. This head normalization procedure is explained in [4] and an illustrating example can be found in section 2.1 of [12].

We number the occurrences of each CHR constraint predicate p appearing in the heads of the rules of some CHR program P following the top-down rule order and right-to-left constraint order. The latter is aimed at ordering first the constraints after the backslash (\setminus) and then those before it, since this gives the refined operational semantics a clearer behavior. We number the rules in the same top-down way.

2.1 The Theoretical Operational Semantics ω_t

The operational semantics ω_t of CHR, sometimes also called *theoretical* or *high-level* operational semantics, is highly nondeterministic. It is formulated as a state transition system.

Definition 2 (Identified constraint). An identified CHR constraint $c\#i$ is a CHR constraint c associated with some unique integer i , the constraint identifier. This number serves to differentiate between copies of the same constraint.

1. Solve $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\omega_t} \langle \mathbb{G}, \mathbb{S}, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$ where c is a built-in constraint and $\mathcal{D}_{\mathcal{H}} \models \exists_0 \mathbb{B}$.	
2. Introduce $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\omega_t} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$ where c is a CHR constraint and $\mathcal{D}_{\mathcal{H}} \models \exists_0 \mathbb{B}$.	
3. Apply $\langle \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\omega_t} \langle C \uplus \mathbb{G}, H_1 \uplus \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$ where \mathcal{P} contains a (renamed apart) rule of the form $r @ H'_1 \setminus H'_2 \iff g \mid C$, θ is a matching substitution such that $\text{chr}(H_1) = \theta(H'_1)$ and $\text{chr}(H_2) = \theta(H'_2)$, $h = (r, \text{id}(H_1), \text{id}(H_2)) \notin T$, and $\mathcal{D}_{\mathcal{H}} \models (\exists_0 \mathbb{B}) \wedge (\mathbb{B} \rightarrow \exists_{\mathbb{B}}(\theta \wedge g))$.	

Fig. 1. The transitions of the theoretical operational semantics ω_t .

We introduce the functions $\text{chr}(c\#i) = c$ and $\text{id}(c\#i) = i$, and extend them to sequences and sets of identified CHR constraints in the obvious manner, e.g. $\text{id}(S) = \{c \mid c\#i \in S\}$.

Definition 3 (ω_t execution state). An ω_t execution state σ is a tuple $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. The goal \mathbb{G} is a multiset of constraints to be rewritten to solved form. The CHR constraint store \mathbb{S} is a set of identified CHR constraints that can be matched with rules in the program \mathcal{P} . Note that $\text{chr}(\mathbb{S})$ is a multiset although \mathbb{S} is a set. The built-in constraint store \mathbb{B} is the conjunction of all built-in constraints that have been posted to the underlying solver. These constraints are assumed to be solved (implicitly) by the host language \mathcal{H} . The propagation history \mathbb{T} is a set of tuples, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. The propagation history is used to prevent trivial non-termination for propagation rules: a propagation rule is allowed to fire on a set of constraints only if the constraints have not been used to fire the same rule before. Finally, the counter $n \in \mathbb{N}$ represents the next integer that can be used to number a CHR constraint. We use $\sigma, \sigma_0, \sigma_1, \dots$ to denote execution states.

For a given CHR program \mathcal{P} , the transitions are defined by the binary relation $\xrightarrow{\omega_t}$ shown in Figure 1. Execution proceeds by exhaustively applying the transition rules, starting from an initial state. Given an initial goal G , the *initial state* is: $\langle G, \emptyset, \text{true}, \emptyset \rangle_1$.

2.2 The Refined Operational Semantics ω_r

Duck et al. [5] introduced the refined operational semantics ω_r of CHR. It formally captures the behavior of many CHR implementations.

The refined operational semantics uses a stack of constraints: when a new constraint arrives in the constraint store it is pushed on the stack. The constraint on top of the stack is called the *active* constraint. The active constraint is used to find matching rules, in the order in which this constraint occurs in the program. When all occurrences have been tried, the constraint is popped from the stack. When a rule fires, its body is executed immediately from left to right, thereby potentially suspending the active constraint because of newly arriving

1.	Solve $\langle [c A], S' \uplus S, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\omega_r} \langle S \upuparrows A, S' \uplus S, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$ if c is a built-in constraint and \mathbb{B} fixes the variables of S' .
2.	Activate $\langle [c A], S, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\omega_r} \langle [c\#n:1 A], S', \mathbb{B}, \mathbb{T} \rangle_{(n+1)}$ if c is a CHR constraint, where $S' = \{c\#n\} \uplus S$.
3.	Reactivate $\langle [c\#i A], S, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\omega_r} \langle [c\#i:1 A], S, \mathbb{B}, \mathbb{T} \rangle_n$
4.	Drop $\langle [c\#i:j A], S, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\omega_r} \langle A, S, \mathbb{B}, \mathbb{T} \rangle_n$ if there is no j^{th} occurrence of c in \mathcal{P} .
5.	Simplify $\langle [c\#i:j A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, \mathbb{B}, \mathbb{T} \rangle_n$ $\xrightarrow{\omega_r} \langle C \upuparrows A, H_1 \uplus S, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$ if the j^{th} occurrence of the constraint c is d_j in a rule r in \mathcal{P} of the form $r @ H'_1 \setminus H'_2, d_j, H'_3 \iff g \mid C$ and $\exists \theta : c = \theta(d_j)$, $\text{chr}(H_k) = \theta(H'_k)$ ($k = 1, 2, 3$), $\mathcal{D} \models \mathbb{B} \rightarrow \exists_{\mathbb{B}}(\theta \wedge g)$, and $T \not\exists h = (\text{id}(H_1), \text{id}(H_2 \upuparrows c\#i \upuparrows H_3), r)$.
6.	Propagate $\langle [c\#i:j A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, \mathbb{B}, \mathbb{T} \rangle_n$ $\xrightarrow{\omega_r} \langle C \upuparrows [c\#i:j A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus S, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$ if the j^{th} occurrence of the constraint c is d_j in a rule r in \mathcal{P} of the form $r @ H'_1, d_j, H'_2 \setminus H'_3 \iff g \mid C$ and $\exists \theta : c = \theta(d_j)$, $\text{chr}(H_k) = \theta(H'_k)$ ($k = 1, 2, 3$), $\mathcal{D} \models \mathbb{B} \rightarrow \exists_{\mathbb{B}}(\theta \wedge g)$, and $T \not\exists h = (\text{id}(H_1 \upuparrows c\#i \upuparrows H_2), \text{id}(H_3), r)$.
7.	Default $\langle [c\#i:j A], S, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\omega_r} \langle [c' A], S, \mathbb{B}, \mathbb{T} \rangle_n$ if no other transition applies, where $c' = c\#i:(j+1)$.

Fig. 2. The transitions of the refined operational semantics ω_r .

constraints. When a constraint becomes topmost again, it resumes its search for matching clauses.

Definition 4 (Occurred identified constraint). *An occurred identified CHR constraint $c\#i:j$ is an identified constraint $c\#i$ annotated with an occurrence number j . This annotation indicates that only matches with occurrence j of constraint c are considered at this point in the execution.*

Definition 5 (ω_r execution state). *An ω_r execution state σ is a tuple $\langle A, S, \mathbb{B}, \mathbb{T} \rangle_n$, where $S, \mathbb{B}, \mathbb{T}$, and n represent the CHR store, the built-in store, the propagation history and the next free identity number just like before. The execution stack A is a sequence of constraints, identified CHR constraints and occurred identified CHR constraints, with a strict*

Execution in ω_r proceeds by exhaustively applying transitions from figure 2 to the initial execution state until the built-in store is unsatisfiable or no transitions are applicable. Initial states are defined in the same way as in ω_t .

3 Guard Reasoning

Consider the following example CHR program, which computes the greatest common divisor using Euclid's algorithm.

Example 1 (gcd).

```
gcd(N) <=> N := 0 | true.
gcd(N) \ gcd(M) <=> N =\= 0, M >= N | gcd(M-N).
```

A query containing two (or more) `gcd/1` constraints with positive integer arguments, will eventually result in a constraint store containing one `gcd(k)` constraint where k is their greatest common divisor. For example, the query `gcd(9),gcd(15)` causes the second rule to fire, resulting in `gcd(9),gcd(6)`. This rule keeps firing until the store contains `gcd(3),gcd(0)`. Now the first rule fires, removing `gcd(0)` from the store. The remaining constraint does indeed contain the greatest common divisor of 9 and 15, namely 3. \square

Taking the refined operational semantics into account, the above CHR program can also be written as

```
gcd(N) <=> N := 0 | true.
gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

because if the second rule is tried, the guard of the first rule must have failed – otherwise the active constraint would have been removed. Hence the condition `N =\= 0` is redundant. Under ω_t semantics, this second version of the CHR program is no longer guaranteed to terminate, since applying the second rule indefinitely (which is a valid execution strategy under ω_t semantics) when the constraint store contains e.g. `gcd(0),gcd(3)` results in an infinite loop.

3.1 Guard simplification

When a simpagation rule or a simplification rule fires, some or all of its head constraints are removed. As a result, for every rule R_i , we know that when this rule is tried, any non-propagation rule R_j with $j < i$, where the set of head constraints of rule R_j is a (multiset) subset of that of rule R_i , did not fire for some reason. Either the heads did not match, or the guard failed. Let us illustrate this general principle with some simple examples.

Example 2 (entailed guard).

```
pos @ sign(P,S) <=> P > 0 | S = positive.
zero @ sign(Z,S) <=> Z := 0 | S = zero.
neg @ sign(N,S) <=> N < 0 | S = negative.
```

If the third rule, `neg`, is tried, we know `pos` and `zero` did not fire, because otherwise, the `sign/2` constraint would have been removed. Because the first rule, `pos`, did not fire, its guard must have failed, so we know that $N \leq 0$. From the failing of the second rule, `zero`, we can derive $N \neq 0$. Now we can combine these results to get $N < 0$, which trivially entails the guard of the third rule. Because this guard always succeeds, we can safely remove it. This results in slightly more efficient generated code, and — maybe more importantly — it might also be useful for other analyses. In this example, guard optimization reveals that all `sign/2` constraints are removed after the third rule, allowing the *never-stored* analysis [14] to detect that `sign/2` is never-stored. \square

Example 3 (rule that can never fire).

```
neq @ p(A) \ q(B) <=> A \== B | ...
eq  @ q(C) \ p(D) <=> C == D | ...
prop @ p(X), q(Y) ==> ...
```

In this case, we can detect that the third rule, `prop`, will never fire. Indeed, because the first rule, `neq`, did not fire, we know that `X` and `Y` are equal and because the second rule, `eq`, did not fire, we know `X` and `Y` are not equal. This is a contradiction, so we know the third rule can never fire. \square

Generalizing from the previous examples, we can summarize guard simplification as follows: If (part of) a guard is entailed by knowledge given by the negation of earlier guards, we can replace it by `true`, thus removing it. However, if the *negation* of (part of a) guard is entailed by that knowledge, we know the rule will never fire and we can remove the entire rule.

In handwritten programs, such never firing rules most often indicate bugs in the CHR program – there is no reason to write rules that cannot fire – so it seems appropriate for the CHR compiler to give a warning message when it encounters such rules. Automatic program generation and source-to-source transformations often introduce never firing rules and redundant guards, so it certainly makes sense to apply guard simplification in that context.

3.2 Head matching simplification

Matchings in the arguments of head constraints can be seen as an implicit guard condition that can also be simplified. Consider the following example:

Example 4 (head matching simplification).

```
p(X,Y) <=> X \== Y | ...
p(X,X) <=> ...
```

We can rewrite the second rule to `p(X,Y) <=> ...`, because the (implicit) condition `X == Y` is entailed by the negation of the guard of the first rule. In the refined operational semantics, this does not change the behavior of the program. However, in a sense the second rule has become simpler: it imposes less conditions on the head constraint arguments. As a result, `p/2` can now easily be seen to be never-stored, so more efficient code can be generated by the compiler. \square

3.3 Type and mode declarations

Head matching simplification can be much more effective if the types of constraints arguments are known.

Example 5 (sum).

```

:- chr_type list(T) ---> [] ; [T | list(T)].
:- constraints sum(+list(int), ?int).

sum([],S) <=> S = 0.
sum([X|Xs],S) <=> sum(Xs,S2), S is X + S2.

```

Since we know the first argument of constraint `sum/2` is a (ground) list, these two rules cover all possible cases and thus the constraint is never-stored. \square

Note that the first declaration is a recursive and generic type definition for lists of some type `T`, a variable that can be instantiated with built-in types like `int`, `float`, the general type `any`, or any user-defined type. The constraint declaration on the second line includes mode and type information. It is read as follows: `sum/2` is a CHR constraint which has two arguments: a ground list of integers and an integer, which can be ground or a variable.

Using this knowledge, we can rewrite the second rule of the example program to “`sum(A,S) <=> A = [X|Xs], sum(Xs,S2), S is X + S2.`”. Again, under ω_r semantics this does not affect any computation, but since `sum/2` is now clearly never-stored, the program can be compiled to more efficient Prolog code.

3.4 Domain knowledge declarations

In addition to type and mode information, we have added the possibility to add domain knowledge declarations. Suppose for instance that a Prolog fact `v/1` is used to indicate the verbosity of the program, which can be “`verbose`”, “`normal`”, or “`quiet`”. Consider the following program:

```

foo(X) <=> v(verbose) | writeln(verbose_foo(X)).
foo(X) <=> v(normal) | write(f).
foo(X), bar(X) ==> \+ v(quiet) | writeln(same_foo_bar(X)).

```

Under the refined operational semantics, the last rule can never fire. The following declaration allows the guard reasoning system to detect this:

```

:- chr_declaration v(verbose) ; v(normal) ; v(quiet).

```

In general such a declaration should be ground and always true. We also allow slightly more general declarations of the form

```

:- chr_declaration predicate(X) ---> expression(X).

```

where all variables occurring on the right hand side should also occur on the left hand side. The left hand side should be either a CHR constraint predicate or a Prolog predicate. For example, if the Prolog predicates `male/1` and `female/1` are used in the guards of rules involving `person/1` constraints, the expected behavior of those predicates could be declared as follows:

```

:- chr_declaration person(X) ---> male(X) ; female(X).

```

This declaration ensures that in a program like

```

person(X) <=> male(X) | person(X,m).
person(X) <=> female(X) | person(X,f).
person(X), person(Y) ==> maybe_marry(X,Y).

```

the last rule is automatically removed.

3.5 Occurrence subsumption

If the head of a rule contains multiple occurrences of the same constraint, we can test for *occurrence subsumption*. We know that when a certain occurrence is tried, all earlier occurrences in constraint removing rules must have failed. If the rule containing this occurrence is not a propagation rule, this also holds for earlier occurrences inside that rule.

The K.U.Leuven CHR compiler already had two optimizations that can be considered to be special cases of occurrence subsumption. *Symmetry analysis* checks rules R with a head containing two constraints c_1 and c_2 that are symmetric, in the sense that there is a variable renaming θ such that $\theta(c_1) = c_2$ and $\theta(c_2) = c_1$ and $\theta(R) = R$. In such rules, one of the c_i 's is made **passive**. This means the occurrence can be skipped. In terms of the ω_r semantics: the **Default** transition can be immediately applied for that occurrence, since the **Simplify** or **Propagate** transitions are never applicable for that occurrence. The second optimization looks at rules which make a constraint have *set semantics*, of the form $c_1 \setminus c_2 \Leftrightarrow true|B$, without head matchings, where c_1 and c_2 are identical. In this case, c_1 can be made **passive** (or c_2 , but it is better to keep the occurrence which immediately removes the active constraint). A more efficient constraint store can be used for c if it has set semantics.

In section 4.1 of [8,9], a concept called *continuation optimization* is introduced. *Fail* continuation optimization is essentially the same as occurrence subsumption, while *success* continuation optimization uses similar reasoning to improve the generated code for occurrences in propagation rules. The HAL CHR compiler, discussed in [9], performs a simple fail continuation optimization, which only considers rules without guards and does not use information derived from the failing of earlier guards.

Example 6 (simple case).

```

c(A,B), c(B,A) <=> p(A),p(B) | true.

```

Suppose the active constraint is $c(X,Y)$. For brevity, we use the phrase “occurrence x fires” as a shortcut for “occurrence x of the active constraint causes rule application”. If the first occurrence does not fire, this means that either $c(Y,X)$ is not in the constraint store, or $p(X), p(Y)$ fails. If the second occurrence fires, then $c(Y,X)$ must be in the constraint store, and the guard $p(Y), p(X)$ must succeed. So it is impossible for the second occurrence to fire if the first one did not¹. If the first occurrence did fire, it removes the active constraint so the second occurrence is not even tried. From the above reasoning it follows that the

¹ We assume conjunctions in guards to be commutative: if $p(X), p(Y)$ fails, then $p(Y), p(X)$ must also fail.

second occurrence is redundant, so we could as well change the rule to $c(A,B), c(B,A)\#passive \Leftrightarrow p(A),p(B) \mid true$. \square

In the following examples, the general occurrence subsumption analysis is able to find much more redundant occurrences than the earlier symmetry and set semantics analyses. Underlined occurrences can be made **passive** so they can be skipped (i.e. the compiler does not need to generate a clause for such an occurrence). All these redundant occurrences are detected by the current K.U.Leuven CHR compiler.

Example 7 (more complicated cases).

1. $a(X,Y,Z), \underline{a(Y,Z,X)}, \underline{a(Z,X,Y)} \Leftrightarrow \dots$
2. $b(X,Y,Z), \underline{b(Y,Z,X)}, \underline{b(Z,X,Y)} \Leftrightarrow (p(X); p(Y)) \mid \dots$
3. $c(A,B,C), \underline{c(A,C,B)}, \underline{c(B,A,C)}, \underline{c(B,C,A)}, \underline{c(C,A,B)}, \underline{c(C,B,A)} \Leftrightarrow \dots$
4. $d(A,B,C), \underline{d(A,C,B)}, \underline{d(B,A,C)}, \underline{d(B,C,A)}, \underline{d(C,A,B)}, \underline{d(C,B,A)} \Leftrightarrow p(A),p(B) \mid \dots$
5. $e(A,B,C), \underline{e(A,C,B)}, \underline{e(B,A,C)}, \underline{e(B,C,A)}, \underline{e(C,A,B)}, \underline{e(C,B,A)} \Leftrightarrow p(A) \mid \dots$
6. $f(A,B), f(B,C) \Leftrightarrow A \backslash == C \mid \dots$
 $f(A,B), \underline{f(B,C)} \Leftrightarrow \dots$ \square

A strong occurrence subsumption analysis takes away the need for CHR programmers to write **passive** pragmas to improve efficiency, since the compiler is able to add them automatically if it can prove that making the occurrence passive is justified, i.e. does not change the program's behavior. Because of this, the CHR source code contains much less of these non-declarative operational pragmas, improving the compactness and logical readability.

Of course, not every redundant occurrence can be detected by our analysis. Consider the last rule in this classic CHR version of the Sieve of Eratosthenes:

Example 8 (too complicated case).

```

candidate(1) <=> true.
candidate(N) <=> prime(N), candidate(N-1).
prime(Y) \ prime(X) <=> 0 := X mod Y | true.

```

In this program, the last occurrence of **prime/1** can be declared to be passive, provided that user queries are of the form **candidate(n)**, with $n \geq 1$. Because **prime/1** constraints are added in reverse order, the guard $0 := X \bmod Y$ will always fail if **prime(X)** is the active constraint. Indeed, for all possible partner constraints **prime(Y)** we have $Y > X > 1$ because of the order in which **prime/1** constraints are added, so $X \bmod Y = X \neq 0$. Our implementation of occurrence subsumption lacks the reasoning capability to detect this kind of situations. Not only does the current implementation lack a mechanism for the CHR programmer to indicate which kind of user queries are allowed, it also does not try to investigate rule bodies to derive the kind of information needed in this example. Furthermore, it is far from trivial to automatically detect complicated entailments like $Y > X > 1 \rightarrow X \bmod Y \neq 0$. \square

4 Formalization

In this section we formalize the guard simplification transformation that was intuitively described above. First we introduce some additional notation for the functor/arity of constraints:

Definition 6 (Functor). For every CHR constraint $c = p(t_1, \dots, t_n)$, we define $\text{functor}(c) = p/n$. For every multiset C of CHR constraints we define $\text{functor}(C)$ to be the multiset $\{\text{functor}(c) \mid c \in C\}$.

4.1 Implicit Preconditions

We consider rules that must have been tried (according to the refined operational semantics) before some rule R_i is tried, calling them *earlier subrules* of R_i .

Definition 7 (Earlier subrule). The rule R_j is an earlier subrule of rule R_i (notation: $R_j \prec R_i$) iff $j < i$ and $\text{functor}(H_j) \subseteq \text{functor}(H_i)$.

Now we can define a logical expression $\text{nesr}(R_i)$ (“no earlier subrule (fired)”) stating the implications of the fact that all constraint-removing earlier subrules of rule R_i have been tried unsuccessfully.

Definition 8 (Nesr). For every rule R_i , we define:

$$\text{nesr}(R_i) = \bigwedge \{(\neg(\theta_j \wedge g_j)) \mid R_j \prec R_i \wedge H_j^r \neq \epsilon\}$$

where θ_j is a matching substitution mapping the head constraints of R_j to corresponding head constraints of R_i .

If mode, type or domain knowledge information is available for head constraints of R_i , it can be added to the $\text{nesr}(R_i)$ conjunction without affecting the following definitions and proofs, as long as this information is correct at any given point in any derivation starting from a legal query. This information is encoded as follows:

modes Each mode is translated to its corresponding Prolog built-in: the + mode yields a `ground/1` condition, the - mode a `var/1` condition, and the ? mode a `true/0` precondition. For instance, for the constraint $c(X, Y, Z)$ the mode declaration $c(+, -, ?)$ results in the precondition $\text{ground}(X) \wedge \text{var}(Y) \wedge \text{true}$.

types Each type declaration results in a compound precondition, based on the type definition. Take for instance the type definition for the boolean type:

```
:- chr_type boolean ---> true ; false.
```

The precondition for constraint $p(X)$, whose argument is of type `boolean`, is: $\text{var}(X) \vee (\text{nonvar}(X) \wedge (X = \text{true} \vee X = \text{false}))$. Note that this precondition explicitly distinguishes between different instantiations of the argument.

Type definitions are recursively unfolded into the formula. Unrestrained unfolding is problematic for recursive types like `list(T)`: it leads to an infinite formula. Hence, we stop the unfolding at a fixed depth.

domain knowledge The unconditional and fully ground domain knowledge is added as is. For the conditional form $\text{Pattern} \text{ ---> Formula}$ we consider all predicate occurrences in $\text{nesr}(R_i)$ and all the heads of R_i . For each occurrence that matches Pattern , we add the corresponding instance of Formula .

4.2 Definition of Guard Simplification

Consider a CHR program P with rules R_i which have guards $g_i = \bigwedge_k g_{i,k}$. Applying guard simplification to this program means rewriting some parts of the guards to **true**, if they are entailed by the “no earlier subrule fired” condition (and already evaluated parts of the guard). The entire guard is rewritten to **fail**, if the *negation* of some part of it is entailed by that condition. This effectively removes the rule. Because head matchings are made explicit, head matching simplification (section 3.2) is an implicit part of guard simplification.

Definition 9 (Guard Simplification). *Applying the guard simplification transformation to a CHR program P (with rules $R_i = H_i \Leftrightarrow \bigwedge_k g_{i,k} | B_i$) results in a new CHR program $P' = GS(P)$ which is identical to P except for the guards, i.e. its rules R'_i are of the form $H_i \Leftrightarrow g'_i | B_i$, where*

$$g'_i = \begin{cases} \mathbf{fail} & \text{if } \exists k \mathcal{D} \models \mathbf{nesr}(R_i) \wedge \bigwedge_{m < k} g_{i,m} \rightarrow \neg g_{i,k}; \\ \bigwedge_k g'_{i,k} & \text{otherwise.} \end{cases}$$

In the second case, the $g'_{i,k}$ are defined by

$$g'_{i,k} = \begin{cases} \mathbf{true} & \text{if } \mathcal{D} \models \mathbf{nesr}(R_i) \wedge \bigwedge_{m < k} g_{i,m} \rightarrow g_{i,k}; \\ g_{i,k} & \text{otherwise.} \end{cases}$$

Note that this definition is slightly stronger compared to the definition given in [16], because it takes into account the left-to-right evaluation of the guard. As a result, internally inconsistent guards like $X > Y$, $Y > X$ can be simplified to **fail**, and internally redundant guards can be simplified, e.g. the condition $X >= Y$ can be removed from $X > Y$, $X >= Y$.

Theorem 1 (Guard simplification & transitions). *Given a CHR program P and its guard-simplified version $P' = GS(P)$. Given an execution state $s = \langle A, S, B, T \rangle_n$ occurring in some derivation for the P program under ω_r semantics, exactly the same transitions are possible from s for P and for P' . In other words, $\rightarrow_P \equiv \rightarrow_{P'}$.*

See Appendix A for the proof.

4.3 Definition of Occurrence Subsumption

Although occurrence subsumption can be seen as a source to source transformation (inserting **passive** pragmas), we use a slightly different approach to define occurrence subsumption formally because the common formal definitions of CHR programs and ω_r derivations do not include pragmas. Instead of introducing the concept of **passive** occurrences in the formal refined operational semantics, we define *occurrence subsumable* occurrences and then we show that trying rule application on a subsumed occurrence is redundant. First we define this auxiliary condition:

Definition 10 (Neocc). Given a non-propagation rule R_i containing in its head multiple occurrences c_m, \dots, c_n of the same constraint c and other partner constraints d . We define for every c_k ($m \leq k \leq n$):

$$\text{neocc}(R_i, c_k) = \bigwedge \{ \neg \theta_l(\text{fc}(R_i, c_l)) \mid m \leq l < k, \theta_l(c_l) = c_k \}$$

where $\text{fc}(R_i, c_l) = (g_i \wedge d \wedge c_m \wedge \dots \wedge c_{l-1} \wedge c_{l+1} \wedge \dots \wedge c_n)$.

As the reader can verify, $\text{fc}(R_i, c_l)$ is the firing condition for rule R_i to fire if c_l is the active constraint. The condition $\text{neocc}(R_i, c_k)$ (“no earlier occurrence (fired)”) describes that if the k^{th} occurrence of c is tried, i.e. application of rule R_i is tried, the earlier occurrences inside rule R_i must have failed (since R_i is not a propagation rule). Now we can define formally which occurrences can be made passive.

Definition 11 (Occurrence subsumption). Given a rule R_i as in the previous definition. We say c_k ($m < k \leq n$) is occurrence subsumable iff

$$\mathcal{D} \models \text{nesr}(R_i) \wedge \text{neocc}(R_i, c_k) \rightarrow \neg \text{fc}(R_i, c_k)$$

In the next section we present a formal correctness proof of both the guard simplification transformation from the previous section and occurrence subsumption.

Theorem 2 (Correctness of Occ. Subsumption). Given a CHR program P and an ω_r derivation for P in which an execution state $s = \langle [c\#i : j|A], S, B, T \rangle_n$ occurs. If c_j is occurrence subsumable, **Simplify** and **Propagate** transition cannot (directly) be applied on state s .

See Appendix A for the proof.

5 Entailment checking

The core component for guard reasoning is a logical entailment checker. In this section we discuss our implementation, in CHR, of such an entailment checker. This implementation is used in the guard simplification analysis to test whether one condition B (e.g. $X < Z$) is entailed by another condition A (e.g. $X < Y \wedge Y < Z$), i.e. whether $A \rightarrow B$ holds. The entailment checker only considers (a fragment of the) host-language built-ins. In particular, it does not try to discover implications of user-defined predicates, which would require a complex analysis of the host-language program.

5.1 Overview

As the entailment checking problem is generally undecidable, our entailment checker is incomplete. It tries to prove that B is entailed by A ; if it succeeds, $A \rightarrow B$ must hold, but if it fails, either $A \not\rightarrow B$ holds or $A \rightarrow B$ holds but could not be shown. The core of the entailment checker is written in CHR. When the entailment $A \rightarrow B$ needs to be checked, we call the entailment checker with the query $\text{known}(A)$, $\text{test}(B)$. Schematically, it works as follows:

1. normalize;
e.g. apply De Morgan's laws, convert $\geq, >, <$ to \leq and \neq
2. evaluate ground expressions;
e.g. replace `known(5 ≤ 3)` by `known(fail)`
3. propagate entailed information;
e.g. if you find both `known(X ≤ Y)` and `known(Y ≤ Z)`, then add `known(X ≤ Z)`
4. succeed whenever `known(B)` is added;
5. succeed if B is entailed;
e.g. `test(X ≠ 3)` is entailed by `known(X ≤ 0)`
6. if there is a disjunction `known(A1 ∨ A2)`: check whether $A_1 \rightarrow B$ and also whether $\neg A_1 \wedge A_2 \rightarrow B$, succeed if both tests succeed;
7. otherwise: give up and fail.

We try to postpone the expansion of disjunctions, because (recursively) trying all combinations of conditions in disjunctions can be rather costly: if A is a conjunction containing n disjunctions, each containing m conditions, there are m^n cases that have to be checked. This is why we check entailment of B *before* a disjunction is expanded. Conjunctions in B are dealt with in the obvious way. If B is a disjunction $B_1 \vee B_2$, we add `known(¬B2)` to the store and test B_1 . We can stop (and succeed) if B_1 is entailed, otherwise we backtrack, add `known(¬B1)` to the store and return the result of testing entailment of B_2 .

5.2 Code Details

The negation of a condition is computed in a straightforward way for host-language built-ins. For example, the negation of `X == Y` is `X \== Y`, `\+ Cond` becomes `Cond`, disjunctions become conjunctions of the negated disjuncts, and so on. For user-defined predicates `p` we simply use `(\+ p)`.

Figure 3 shows how the normalization of `known/1` and `test/1` constraints is done. Ground conditions are evaluated using rules like the following:

```
known(X=<Y) <=> number(X), number(Y), X=<Y | true.
known(X=<Y) <=> number(X), number(Y), X>Y | known(fail).
test(X=<Y) <=> number(X), number(Y), X=<Y | true.
```

In Fig. 4 some examples are given of rules that propagate entailed information. The `idempotence` rule and execution under the refined semantics is crucial for termination of this propagation phase.

A simplified version of the rest of the entailment checker is listed in Fig. 5.

Note that Prolog disjunction in the rule body is used to check disjunctions in `test/1` constraints. To deal with disjunctions in `known/1` constraints, a bit of trickery is needed. We want to avoid branching until it is needed. While the propagation rules are already applied before the `test` constraint is activated, the `disjunction` rule can only be applied when the `test` constraint has almost reached its last occurrence. Now we use a double Prolog negation to test both disjuncts. The predicate `try(A, X)` fails if $A \rightarrow X$ can be shown, so its

```

:- chr_constraint known/1, test/1.

known(G) <=> normal_form(G,N) | known(N).
test(G) <=> normal_form(G,N) | test(N).

normal_form(X>Y, (Y=<X, X=\=Y)).
normal_form(X>=Y, Y=<X).
normal_form(X<Y, (X=<Y, X=\=Y)).
normal_form(X is Y, X:=Y).
normal_form(\+ G, NotG) :- negation(G,NotG).

negation(X = Y, X \= Y).          negation(X \= Y, X = Y).
negation(X < Y, Y =< X).          negation(X > Y, X =< Y).
negation(X =< Y, Y < X).          negation(X >= Y, X < Y).
negation(X == Y, X \== Y).        negation(X \== Y, X == Y).
negation(X \= Y, X := Y).         negation(X := Y, X =\= Y).
negation(var(X), nonvar(X)).      negation(nonvar(X), var(X)).
negation((A;B), (\+ A, \+ B)).    negation((A,B), (\+ A; \+ B)).
negation(true, fail).            negation(fail, true).
negation(\+ G, G).              % double negation

```

Fig. 3. Conversion to normal form.

negation succeeds if $A \rightarrow X$ holds. By using a double negation, all propagated consequences of A are automatically undone.

Disjunctions in the antecedent are the main bottleneck of the entailment checker: every disjunction potentially doubles the amount of work to be done, so the checking is potentially exponential in the input size. In the case of guard simplification, the antecedents consist of negations of guards, and guards are typically conjunctions. As a result, after normalization the antecedent consists of disjunctions (of negated conjuncts). Hence, for efficiency reasons it is important to avoid disjunction branching if possible. In addition to the above strategy of delaying disjunctions, we have added rules to simplify some common cases of redundant disjunctions. Examples of such rules are the following:

```

known((fail; B)) <=> known(B).
known((true ; A)) <=> true.
known(A \ known((\+ A; B)) <=> known(B).
known(A \ known((\+ A, C; B)) <=> known(B).

```

5.3 Flattening

The generic constraints `known/1` and `test/1` provide a conceptual simplicity in formulating and maintaining the rules of the entailment checker. However, this genericity incurs a runtime penalty: the CHR compiler fails to efficiently

idempotence	@ known(G) \ known(G) <=> true.
inconsistency	@ known(X), known(\+ X) <=> known(fail).
conjunction	@ known((A,B)) <=> known(A), known(B).
eq_neq_inconsistency	@ known(X\=Y), known(X==Y) <=> known(fail).
eq_transitivity	@ known(X==Y), known(Y==Z) ==> known(X==Z).
neq_substitution	@ known(X==Y), known(Y\=Z) ==> known(X\=Z).
eq_symmetry	@ known(X==Y) ==> known(Y==X).
neq_symmetry	@ known(X\=Y) ==> known(Y\=X).
neq_inconsistency	@ known(X\=X) ==> known(fail).
leq_antisymmetry	@ known(X<Y), known(Y<X) <=> known(X=:Y).
leq_transitivity	@ known(X<Y), known(Y<Z) ==> known(X<Z).
leq_substitution1	@ known(X=:Y), known(X < Z) ==> known(Y < Z).
leq_substitution2	@ known(X=:Y), known(Z < X) ==> known(Z < Y).
strict_lt_transitivity	@ known(X<Y), known(X\=Y), known(Y<Z), known(Y=\=Z) ==> known(X\=Z).
aneq_inconsistency	@ known(X=\X) <=> known(fail).
aeq_aneq_inconsistency	@ known(X=:Y), known(X=\Y) <=> known(fail).
aeq_transitivity	@ known(X=:Y), known(Y=:Z) ==> X \== Z known(X=:Z).
aeq_symmetry	@ known(X=:Y) ==> known(Y=:X).
aneq_symmetry	@ known(X=\Y) ==> known(Y=\X).

Fig. 4. Propagation of known/1 constraints.

index the constraints, and each active constraint has to consider (almost) all occurrences.

Because the entailment checker is one of the main performance bottlenecks in the K.U.Leuven CHR compiler, the above inefficiency is unacceptable. Fortunately, it can be mitigated with little effort by automated rule specialization. Sarna-Starosta and Schrijvers [11] propose a technique for specializing constraints with respect to the different toplevel function symbols in their arguments that rules try to match. In the current version of the compiler, this specialization leads to 20 versions of `test/1` and 26 versions of `known/1`, e.g. `known_==/2`, `test_true/0`, ...

This specialization provides indexing on the toplevel function symbol for free. The CHR compiler always allocates separate indexing datastructures for distinct constraint symbols.

Of course, the specialization of constraints leads to the specialization of rules, and, because many rules only apply to one specialized form, fewer occurrences for each specialized constraint. For instance, we obtain only two occurrences for `test_true/0`:

```
known_fail \ test_true <=> true.
test_true <=> true.
```

This fully automatic specialization makes the entailment checker roughly twice as fast.

```

fail_implies_everything @ known(fail) \ test(X) <=> true.
trivial_ entailment      @ known(G) \ test(G) <=> true.
eq_implies_leq1         @ known(X:=Y) \ test(X<Y) <=> true.
eq_implies_leq2         @ known(X:=Z) \ test(X<Y) <=> number(Y), number(Z), Z<Y | true.
eq_implies_leq3         @ known(X:=Z) \ test(Y<X) <=> number(Y), number(Z), Y<Z | true.
leq_implies_leq1        @ known(X<Z) \ test(X<Y) <=> number(Y), number(Z), Z<Y | true.
leq_implies_leq2        @ known(X<Y) \ test(Z<Y) <=> number(X), number(Z), Z<X | true.
leq_implies_neq1        @ known(X<Z) \ test(X=\=Y) <=> number(Y), number(Z), Y>Z | true.
leq_implies_neq2        @ known(X<Y) \ test(Y=\=Z) <=> number(X), number(Z), Z<X | true.
leq_implies_neq2        @ known(X<Y) \ test(Z=\=Y) <=> number(X), number(Z), Z<X | true.
true_is_true            @ test(true) <=> true.

test_conjunction         @ test((A,B)) <=> test(A), known(A), test(B).
test_disjunction         @ test((A;B)) <=> known(\+ B),test(A) ; known(\+ A),test(B).

disjunction              @ test(X), known((A;B))
                        <=> \+ try(A,X), !, known(\+ A), \+ try(B,X).
give_up                  @ test(_) <=> fail.

try(A,X) :- known(A), (test(X) -> fail ; true).

```

Fig. 5. Part of the program to check entailments.

6 Implementation

We have implemented guard simplification and occurrence subsumption in the K.U.Leuven CHR compiler [13], which can be found in recent releases of SWI-Prolog [21]. In this section we give a brief overview of our implementation of guard simplification and occurrence subsumption, which depends heavily on the entailment checker discussed in the previous section.

The guard simplification / occurrence subsumption compilation phase rewrites every rule in the CHR program. In the rewritten rules, the redundant parts of the guard have been removed, the head matchings (an implicit part of the guard) are made as general as possible and subsumed occurrences are declared to be passive. As a result, the generated code is more efficient because redundant checks are removed, and also the next compilation phases – like storage analysis – are more effective.

Our implementation works as follows. For every rule R_i , we first compute a conjunction of inferred information. Then we use this information to transform the rule to a simpler and more efficient form.

6.1 Inferring information

First we make the head matchings explicit, inserting fresh variables in the arguments of head constraints as needed. For example, the rule

$$c([X|Xs], Y, Y) \Leftrightarrow \dots \mid \dots$$

would be rewritten to the equivalent rule in head normal form:

$$c(A, Y, B) \Leftrightarrow A = [X|Xs], B == Y, \dots \mid \dots$$

Next we iteratively construct a conjunction similar to $\text{nesr}(R_i)$ from section 4, containing the negations of the guards of the earlier subrules $R_j \prec R_i$. All possible substitutions have to be considered. As an example, consider the program:

$$\begin{aligned} c(X) &\Leftrightarrow p(X) \mid \dots \\ c(2) &\Leftrightarrow q \mid \dots \\ c(A), c(B) &\Leftrightarrow \dots \mid \dots \end{aligned}$$

For the third rule, the following conjunction would be computed:

$$(A \text{ \textbackslash == } 2; \text{ \textbackslash + } q), (B \text{ \textbackslash == } 2; \text{ \textbackslash + } q), \text{ \textbackslash + } p(A), \text{ \textbackslash + } p(B)$$

Finally we add type and mode information by looking up the type and mode declarations for the head constraints of R_i , unfolding the type definitions to the nesting depth needed (see Section ??). We also add the relevant information from the domain knowledge declarations.

6.2 Using the information

Now we can use the derived information to transform the rule. Schematically, our implementation works as follows:

1. for every part of the guard of R_i (the $g_{i,k}$'s from section 4): check if it is entailed by the derived information and remove it if it is (i.e. replace it with **true**); if its negation is entailed, replace it with **fail**;
2. move every entailed head matching to the body if the variables in the right hand side of the matching do not occur in the guard; if they also do not occur in the body, remove the head matching;
3. produce a warning message if the guard now entails **fail**, or if the head matchings entail **fail**. This means that rule R_i will never fire, which probably indicates a bug in the CHR program;
4. for every occurrence c_k of a constraint that occurs more than once in R_i , compute $\text{neocc}(R_i, c_k)$ and do occurrence subsumption by checking whether $\neg \text{fc}(R_i, c_k)$ is entailed by $\text{nesr}(R_i) \wedge \text{neocc}(R_i, c_k)$, i.e. check whether occurrence c_k can be safely set to 'passive'.

As an example of the second step, consider the rule

$$c([X|Xs], [], A, A, [B|Bs]) \Leftrightarrow B > 0 \mid d(X, A).$$

and assume the derived information entails that the first arguments of $c/5$ is a non-empty list, the second argument is an empty list and the third and fourth argument are identical. The rule would be rewritten to

$$c(Z, _, A, _, [B|_]) \Leftrightarrow B > 0 \mid Z = [X|_], d(X, A).$$

7 Experimental results

In order to get an idea of the efficiency gain obtained by guard simplification and occurrence subsumption, we have measured the performance of several CHR benchmarks, both with and without the optimization. All benchmarks were performed in SWI-Prolog [21] Pentium 4 machine running Debian GNU/Linux with a low load. Before looking at the runtimes, we first take a look at the code the compiler generates for an example CHR program, and how this code is improved by guard simplification.

7.1 Generated code comparison

Consider this fragment from a prime number generating program from the CHR web site [19]:

```
filter([X|In],P,Out) <=> 0 =\= X mod P |
                        Out = [X|Out1],
                        filter(In,P,Out1).
filter([X|In],P,Out) <=> 0 := X mod P |
                        filter(In,P,Out).
filter([],P,Out) <=> Out = [].
```

The CHR compiler (without guard simplification) generates general code for the `filter/3` constraint. Because no information is known about the arguments of `filter/3`, the compiled code has to take into account variable triggering and the possibility that none of the rules apply and the constraint has to be stored. Following the compilation scheme explained in [10], we get this generated code:

```
filter(List,P,Out) :- filter(List,P,Out, _ ).

% first occurrence
filter(List,P,Out,C) :-
    nonvar(List), List = [X|In],
    0 =\= X mod P, !,
    ... % removecode
    Out = [E|Out1], filter(In,P,Out1).

% second occurrence
filter(List,P,Out,C) :-
    nonvar(List), List = [X|In],
    0 := X mod P, !,
    ... % removecode
    filter(In,P,Out).

% third occurrence
filter(List, _ ,Out,C) :-
```

```

List == [], !,
... % removecode
Out = [].

% insert into store if no rule applied
filter(List,P,Out,C) :-
... % insertcode

```

If we enable guard simplification, the guard in the second rule is removed, but this alone does not considerably improve efficiency. However, we can add type and mode information and then use the guard simplification analysis to transform the program to an equivalent and more efficient form.

In this example, the programmer intends to call `filter/3` with the first two arguments ground, while the third one can have any instantiation. The first and the third argument are lists of integers, while the second argument is an integer. So we add the following type and mode declaration:

```
:- constraints filter(+list(int),+int,?list(int)).
```

Using this type and mode information, guard simplification now detects that all possibilities are covered by the three rules. The guard in the second rule can be removed, so the `filter/3` constraint with the first argument being a non-empty list is always removed after the second rule. Thus in order to reach the third rule, the first argument has to be the empty list – it cannot be a variable because it is ground and it cannot be anything else because of its type. As a result, we can drop the head matching in the third rule:

```

filter([X|In],P,Out) <=> 0 =\= X mod P |
                        Out = [X|Out1],
                        filter(In,P,Out1).
filter([_|In],P,Out) <=> filter(In,P,Out).
filter(_,P,Out) <=> Out = [].

```

This transformed program is compiled to more efficient Prolog-code, because never-stored analysis detects `filter/3` to be never-stored after the third rule. The generated code for the guard simplified program is considerably simpler:

```

filter([X|In],P,Out) :- 0 =\= X mod P, !,
                        Out = [X|Out1],
                        filter(In,P,Out1).
filter([_|In],P,Out) :- !, filter(In,P,Out).
filter(_,_,[ ]).

```

7.2 Guard simplification results

Figure 6 gives an overview of our results. The first column indicates the benchmark name and the parameters that were used. These benchmarks are available

at [20]. The second and third column indicate whether mode and type declarations were provided, respectively. The fourth column indicates whether guard simplification was enabled. In all these columns, an empty cell means the choice has no influence on the resulting compiled code (so it can be “yes” or “no”). The fifth column shows the size of the resulting compiled Prolog code as a pair of the form (*#Clauses ; #Lines*), not including auxiliary predicates. The last column shows the runtime in seconds and a percentage comparing the runtime to that of the version with mode information but without guard simplification. If a cell contains an equality sign (“=”), we could not measure any performance difference compared to the version in the row just above that cell. If a cell contains an equivalence sign (“≡”), the Prolog code for that row is identical to the one in the row just above. For every benchmark, the results for hand-optimized Prolog code are included, representing the ideal target code.

We have measured similar results [16] in hProlog [18]. The only significant difference with the results presented here, is the amount of run time improvement caused by adding mode information. In hProlog, this improvement is typically 20 to 30 percent, while in SWI-Prolog, it can be 50 to 70 percent. The reason is that the `nonvar/1`-test and other redundant code – which is removed when the argument is declared to be ground – is handled much more efficiently by hProlog.

Discussion. The first benchmark, `sum`, computes the sum of the elements of a list of 10000 numbers (all 1), and is repeated 500 times (see example 5 page 7):

```
sum([],S) <=> S = 0.
sum([A|R],S) <=> sum(R,T), S is A+T.
```

If type and mode declarations are provided, guard simplification moves the head matching to the body, enabling never-stored analysis to remove redundant code to add `sum/2` to the constraint store. As in the other benchmarks, no significant performance difference could be measured between the resulting compiled program²

```
sum([],S) :- !, S = 0.
sum([A|R],S) :- sum(R,T), S is A+T.
```

and the handwritten Prolog code

```
sum([],S) :- S = 0.
sum([A|R],S) :- sum(R,T), S is A+T.
```

The second benchmark is an example of how guard simplification can in some way make mode information redundant. The CHR-program looks like this:

² For readability, variables have been renamed in the generated code shown here. The results are similar for a tail-recursive version of `sum/2`.

<i>Benchmark</i>	<i>Mode</i>	<i>Type</i>	<i>Guard simplification</i>	<i>Program size</i>	<i>Runtime (%)</i>
sum (10000,500)	no	no		4 ; 46	12.23 (243)
	yes	no		3 ; 10	5.03 (100)
	yes	yes	yes	2 ; 6	4.49 (89)
	hand-optimized code			2 ; 5	= =
Takeuchi (1000)	no		no	4 ; 50	136.11 (173)
	yes		no	3 ; 17	78.62 (100)
			yes	2 ; 12	72.88 (93)
	hand-optimized code			≡	≡ ≡
nrev (30,50000)	no	no		8 ; 92	47.83 (342)
	yes	no		6 ; 20	13.97 (100)
	yes	yes	yes	4 ; 11	8.44 (60)
	hand-optimized code			4 ; 7	= =
cprimes (100000)	no		no	14 ; 160	196.48 (245)
	no		yes	12 ; 120	= =
	yes	no	no	11 ; 42	80.20 (100)
	yes	no	yes	10 ; 35	= =
	yes	yes	yes	8 ; 25	79.25 (99)
	hand-optimized code			8 ; 23	= =
dfsearch (16,500)	no		no	5 ; 67	149.02 (397)
	no		yes	5 ; 66	141.75 (377)
	yes	no	no	4 ; 16	37.58 (100)
	yes	no	yes	4 ; 15	31.63 (84)
	yes	yes	yes	3 ; 11	29.97 (80)
	hand-optimized code			3 ; 8	= =

Fig. 6. Benchmark results for guard simplification.

```
tak(X,Y,Z,A) <=> X =< Y | ...
tak(X,Y,Z,A) <=> X > Y | ...
```

The first three arguments are supposed to be ground integers. If this mode information is given, the possibility of variable triggering can be excluded. However, even without mode information, guard simplification removes the guard in the second rule. As a result, the constraint is detected as being never-stored, also excluding the possibility of variable triggering. In this case, the generated code is identical to the handwritten Prolog code. The guard $X > Y$ is removed because it is (entailed by) the negation of $X =< Y$. When $X =< Y$ fails, we know X and Y are ground terms evaluating to numbers, and $X > Y$. If in some other host language, $X =< Y$ would fail if its arguments are invalid – instead of resulting in some fatal error message or exception – then it would have a different negation, for instance $(X > Y ; \backslash + \text{number}(X) ; \backslash + \text{number}(Y))$. In that case, guard simplification would not remove the guard of the second rule, except when mode and type information is given.

In the third benchmark, `nrev`, a list of length 30 is reversed 50000 times using the classic naive algorithm. Except for some redundant cuts, the generated code:

```

nrev([],Ans) :- !, Ans = [].
nrev([X|Xs],Ans) :- nrev(Xs,L), app(L,[X],Ans).
app([],L,M) :- !, L = M.
app([X|L1],L2,[X|L3]) :- app(L1,L2,L3).

```

is essentially identical to the handwritten Prolog program:

```

nrev([],[]).
nrev([X|Xs],Ans):- nrev(Xs,L), app(L,[X],Ans).
app([],L,L).
app([X|L1],L2,[X|L3]):- app(L1,L2,L3).

```

The example in section 7.1 is a fragment from the fourth benchmark, `cprimes`, which computes the first 100,000 prime numbers. The last benchmark, `dfsearch`, performs a depth-first search on a large tree. In both cases, the generated code for the guard simplified version with mode and type information is again essentially identical to the handwritten Prolog code.

Conclusion. Overall, for these benchmarks, the net effect of the guard simplification transformation – together with never-stored analysis and use of mode information to remove redundant variable triggering code – is cleaner generated code which is much closer to what a Prolog programmer would write. As a result, a major performance improvement is observed in these benchmarks, which are CHR programs that basically implement a deterministic algorithm.

Naive compilation causes CHR programs to have a relatively low performance compared to native host-language (Prolog) alternatives. As a result, CHR programmers usually write auxiliary predicates in Prolog instead of formulating them directly in CHR. Thanks to guard simplification and other analyses, the programmer can now simply implement everything as CHR rules, relying on the compiler to generate efficient code. Mixed-language programs often use inelegant interface constructs, like rules of the form `foo(X) \ getFoo(Y) <=> Y = X`, to read information from the constraint store in the host-language parts when this information is needed. Host-language interface constraints like `getFoo/1` can be avoided by writing the entire program in CHR. Thanks to (amongst others) guard simplification, this can be done without performance penalty.

7.3 Occurrence subsumption results

Figure 7 shows the results of occurrence subsumption for four benchmarks. Symmetry and set semantics analyses were disabled in both cases because they are special cases of occurrence subsumption. The second column indicates whether occurrence subsumption was enabled. The third column indicates the number of non-passive occurrences. The runtime column is as in Fig. 6. The benchmarks correspond to the rules in Example 7 (page 10).

Occurrence subsumption seems to result in a substantial performance improvement, if there are subsumable occurrences (which is of course true for

<i>Benchmark</i>	<i>Occ. subsumption</i>	<i># occurrences</i>	<i>Runtime (%)</i>
a	no	3	52.8 (100)
(5000)	yes	1	17.1 (32)
b	no	3	52.1 (100)
(5000)	yes	2	34.3 (66)
c	no	6	86.5 (100)
(5000)	yes	1	17.2 (20)
d	no	6	84.7 (100)
(5000)	yes	3	50.3 (59)
e	no	6	86.4 (100)
(5000)	yes	3	49.9 (58)
f	no	4	64.4 (100)
(5000)	yes	3	47.8 (74)

Fig. 7. Benchmark results for occurrence subsumption.

these benchmarks). Occurrence subsumption also reduces the size of the generated code, by eliminating entire clauses. Compared to guard simplification, this size reduction is more visible, unless of course – as in the case of benchmarks from the previous section – guard simplification reveals the never-stored property, which also allows substantial simplification of the generated code.

8 Guard reasoning under ω_p semantics

Guard reasoning can also be applied in the context of different operational semantics. In this section we consider the priority semantics ω_p introduced by De Koninck et al. [3]. The programmer assigns a priority to every rule. The ω_p semantics is an instantiation of ω_t which ensures that of all applicable rules, the one with the highest priority is applied first. Priorities are strictly positive integer numbers, where smaller numbers indicate higher priority.

Consider the `gcd` program of example 1, executed under ω_p semantics, and annotated with the following (dynamic) priorities:

```

1   :: gcd(N) <=> N := 0 | true.
N+2 :: gcd(N) \ gcd(M) <=> N =\= 0, M >= N | gcd(M-N).

```

In this case, the entire guard of the second rule is redundant. The reasoning is as follows. The first rule takes priority over the second rule, so we can derive that if the second rule is applicable, the arguments of both head constraints must be different from zero. Now suppose we have the constraints `gcd(A)` and `gcd(B)` and the second rule is applicable for some matching $\theta = \{N/A, M/B\}$, with priority $A + 2$. Suppose that $M < N$, so $B < A$. The matching $\theta' = \{N/B, M/A\}$ has a lower priority $B + 2 < A + 2$, so the second rule cannot be applicable with matching θ . From this contradiction we can derive that $M \geq N$ should always hold when the priority semantics allows the rule to be applicable. So under

the priority semantics ω_p , the following simplified program is equivalent to the original program:

```
1  :: gcd(N) <=> N == 0 | true.
N+2 :: gcd(N) \ gcd(M) <=> gcd(M-N).
```

We give two more examples to illustrate how we can reason about guards under the ω_p semantics.

Example 9 (static priorities). Consider the following rules:

```
1  :: domain(A,L:U) <=> L > U | fail.
2  :: domain(A,L:U), domain(A,U:L) <=> L = U | A = U.
3  :: domain(A,L:U), domain(A,U:V) <=> L < V | A = U.
```

The first rule removes `domain/2` constraints with an empty domain (lower bound strictly larger than upper bound). When the second rule is tried, we know the first rule is not applicable because of the priorities. So for the second rule, we know that $\neg(L > U)$ and also that $\neg(U > L)$, because otherwise one of the head constraints would have been removed by the first rule. Now we have $\neg(L > U) \wedge \neg(U > L) \leftrightarrow L \leq U \wedge U \leq L \leftrightarrow L = U$, so the guard of the second rule is redundant. Now for the third rule, we know that $L \leq U$ and $U \leq V$ because of the first rule, and also that $L \neq V$ because of the second rule. Hence $L < V$ and the guard of the third rule is also redundant.

Example 10 (dynamic priorities). Consider the following rules:

```
X  :: a(X,Y,Z) <=> Y > Z | true.
Y  :: a(X,Y,Z) <=> X < Z | true.
Z  :: a(X,Y,Z) <=> Z > X | true.
```

We can derive that the last rule can never fire. The reasoning is as follows. When we try the last rule for a given `a/3` constraint, the first rule was not applied earlier because it would have removed the constraint. Either the first rule was not applied because the priorities allow non-application (so $Z \leq X$), or it was not applied because the guard failed (so $\neg Y > Z$). So from inspecting the first rule, assuming the last rule can be applied, we can derive that $Z \leq X \vee Y < Z$. Similarly, from inspection of the second rule, we can derive that $Z \leq Y \vee Z < X$. Now if the last rule is applicable, its guard should hold, so $Z > X$. It is easy to see that this is inconsistent with the two derived formulae, so we can conclude that the last rule is redundant and may be removed.

In future work we plan to formalize and implement guard reasoning under ω_p semantics.

9 Conclusion

By reasoning about guards and the operational semantics under which the program will be executed, we can automatically identify redundant guards and

redundant rules. As a result, a CHR programmer can write a correct program under the general ω_t semantics, and the compiler will convert it to a more efficient program which is only correct under a particular instance of ω_t (for example ω_r or ω_p). Type and mode declarations can also be taken into account.

In order to achieve higher efficiency, CHR programmers often write parts of their program in Prolog if they do not require the additional power of CHR. They no longer need to write mixed-language programs for efficiency: they can simply write the entire program in CHR. Non-declarative auxiliary “host-language interface” constraints like `getFoo/1` (see section 7.2) can be avoided.

9.1 Related work

Guard simplification is somewhat similar to *switch detection* in Mercury [7]. In Mercury, disjunctions – explicit or implicit (multiple clauses) – are examined for determinism analysis. In general, disjunctions cause a predicate to have multiple solutions. However, if for any given combination of input values, only one of the disjuncts can succeed, the disjunction does not affect determinism. Because they superficially resemble switches in the C programming language, such disjunctions are called *switches*. Switch detection checks unifications involving variables that are bound on entry to the disjunction and occurring in the different branches. In a sense, this is a special case of guard simplification, since guard simplification considers other tests as well, using a more general entailment checking mechanism. Guard simplification analysis can be used to remove redundant guard conditions on the source level, because CHR rules are committed-choice. It is harder to express the switch detection optimization as a source to source transformation for Mercury programs.

Guard simplification and occurrence subsumption can be combined into one analysis. In some intermediate representation, there can be a separate copy of each rule for every constraint occurrence c , where all heads except c are passive. This representation is closer to the generated Prolog code, where each occurrence gets a separate clause in which (after matching the partner constraints) the rule guard and body are duplicated. From this angle, guard simplification is simplifying the guards of all copies of a certain rule at once, while occurrence subsumption is simplifying the guard of one specific copy to `fail`, removing that copy. A stronger and more general optimization can be obtained by simplifying the guard of each copy separately. This optimization can no longer be expressed as a pure source to source transformation. We have elaborated that approach in [15]. While reasoning on the level of constraint occurrences is stronger, it is also computationally more expensive and specific to the refined semantics, which has the concept of active occurrences.

Occurrence subsumption is essentially the same as *fail continuation optimization* [8, 9], although our implementation performs much more complex implication reasoning, resulting in a stronger optimization compared to [8, 9]. The related concept of *success continuation optimization* [9] was explored in [15]. The K.U.Leuven CHR system currently implements a weak form of success continuation optimization: head matchings are taken into account to skip non-matching

occurrences in the continuation of propagation rules. This could be generalized by taking into account all information that can be derived by guard reasoning.

9.2 Future work

Our current entailment checker can only deal with a limited number of Prolog built-ins. Using domain knowledge declarations, properties of user-defined Prolog predicates can be declared to enhance the capabilities of the entailment checker. The expressivity of such declarations is still fairly limited, and such declarations have to be added manually by the programmer. We see two ways for substantial further improvement. Firstly, the entailment checker could statically evaluate a call to a Prolog predicate to determine its success or failure. Here a conservative approach is essential as the pitfalls of side effects and non-termination must be avoided. Secondly, we may derive a solver for the Prolog predicate from its logic program definition with the techniques of [2]. We conjecture that the latter leads to stronger results than meta-interpretation, but at a greater computational cost.

It would be interesting to explore a generalization of guard simplification that not just removes redundant conjuncts, but also replaces computationally expensive conditions by cheaper ones. For example, consider this program:

$$\begin{aligned} p(X) &\Leftarrow X \geq 0, g(X) \mid \dots \\ p(X) &\Leftarrow X < 0, \text{\textbackslash}+ g(X) \mid \dots \\ p(X) &\Leftarrow g(X) \mid \dots \end{aligned}$$

If $g/1$ is a predicate that takes a very long time to evaluate, we could change the guard of the last rule to $X < 0$, because $\neg(X \geq 0 \wedge g(X)) \wedge \neg(X < 0 \wedge \neg g(X))$ entails $g(X) \leftrightarrow X < 0$.

When there are many earlier subrules to consider in the guard simplification analysis, the performance of our current implementation may become an issue. Rules with many shared head constraints are an even bigger performance issue, because of the combinatorial blowup caused by constructing all possible mappings from the head of an earlier subrule to the current rule head. For example, if some constraint c occurs n times in the head of an earlier subrule, and m ($\geq n$) times in the current head, there are $\frac{m!}{(m-n)!}$ conditions to be added to the `nesr` conjunction. In future work we hope to further improve the scalability of our implementation.

The information entailed by the failure and success of guards, used here to eliminate redundant guards and rules, would also be useful in other program analyses and transformations. Of particular interest is the generation of specialized code for individual constraint calls in rule bodies. Taking into account the success and failure leading up to this call, stronger guard simplification may be performed than in the general case.

Finally, an interesting area for future work is the formalization and implementation guard reasoning for the priority semantics, as we mentioned in Section 8. The relation between guards and rule priorities needs further investigation: perhaps a sort of reverse reasoning can be used to simplify priorities given the

guards. In this way, it could be possible to replace dynamic priorities by static priorities or to execute part of a program under the more efficient refined semantics, perhaps by adding some guards.

References

1. Slim Abdennadher and Thom Frühwirth. Operational equivalence of CHR programs and constraints. In J. Jaffar, editor, *CP '99: Proc. 5th Intl. Conf. Princ. Pract. Constraint Programming*, LNCS 1713, pages 43–57, Alexandria, USA, 1999.
2. Slim Abdennadher and Christophe Rigotti. Automatic generation of CHR constraint solvers. In S. Abdennadher, T. Frühwirth, and C. Holzbaaur, editors, *Special Issue on Constraint Handling Rules*, volume 5(4–5) of *TPLP*, pages 403–418, 2005.
3. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In M. Leuschel and A. Podelski, editors, *PPDP '07: Proc. 9th Intl. Conf. Princ. Pract. Declarative Programming*, pages 25–36, Wroclaw, Poland, July 2007.
4. Gregory Duck, Peter Stuckey, María García de la Banda, and Christian Holzbaaur. Extending arbitrary solvers with Constraint Handling Rules. In *PPDP '03: Proc. 5th Intl. Conf. Princ. Pract. Declarative Programming*, Uppsala, Sweden, 2003.
5. Gregory Duck, Peter Stuckey, María García de la Banda, and Christian Holzbaaur. The refined operational semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *ICLP '04: Proc. 20th Intl. Conf. Logic Programming*, LNCS 3132, pages 90–104, Saint-Malo, France, 2004.
6. Thom Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming, Journal of Logic Programming*, volume 37 (1–3), October 1998.
7. Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the 19th Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.
8. Christian Holzbaaur, María García de la Banda, David Jeffery, and Peter Stuckey. Optimizing compilation of Constraint Handling Rules. In P. Codognet, editor, *ICLP '01: Proc. 17th Intl. Conf. Logic Programming*, LNCS, pages 74–89, 2001.
9. Christian Holzbaaur, María García de la Banda, Peter Stuckey, and Gregory Duck. Optimizing compilation of Constraint Handling Rules in HAL. In S. Abdennadher, T. Frühwirth, and C. Holzbaaur, editors, *Special Issue on Constraint Handling Rules*, volume 5(4–5) of *TPLP*, pages 503–531, 2005.
10. Christian Holzbaaur and Thom Frühwirth. Compiling Constraint Handling Rules into Prolog with attributed variables. In G. Nadathur, editor, *PPDP '99: Proc. 1st Intl. Conf. Princ. Pract. Declarative Programming*, LNCS 1702, 1999.
11. Beata Sarna-Starosta and Tom Schrijvers. Indexing techniques for CHR based on program transformation. Technical Report CW 500, K.U.Leuven, Dept. Computer Science, August 2007.
12. Tom Schrijvers and Bart Demoen. Antimonotony-based Delay Avoidance for CHR. Technical Report CW 385, K.U.Leuven, Department of Computer Science, July 2004.
13. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In T. Frühwirth and M. Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, number 2004-01, 2004.
14. Tom Schrijvers, Peter Stuckey, and Gregory Duck. Abstract Interpretation for Constraint Handling Rules. In *Proceedings of the 7th Intl. Conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

15. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and continuation optimization for occurrence representations of CHR. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming (ICLP'05)*, volume 3668 of *LNCS*, pages 83–97, Sitges, Spain, October 2005.
16. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard simplification in CHR programs. In A. Wolf, T. Frühwirth, and M. Meister, editors, *Proceedings of the 19th Workshop on (Constraint) Logic Programming (W(C)LP'05)*, number 2005-01 in *Ulmer Informatik-Berichte*, pages 123–134, Ulm, Germany, February 2005.
17. Jon Sneyers, Peter Van Weert, Leslie De Koninck, and Tom Schrijvers. As time goes by: Constraint Handling Rules — A survey of CHR research between 1998 and 2007. *TPLP*, 2008. To be submitted.
18. Bart Demoen. hProlog home page. <http://www.cs.kuleuven.be/~bmd/hProlog>.
19. Martin Kaeser et al. WebCHR. <http://chr.informatik.uni-ulm.de/~webchr/>.
20. Tom Schrijvers. CHR benchmarks and programs. Available at the K.U.Leuven CHR home page at <http://www.cs.kuleuven.be/~toms/Research/CHR/>.
21. Jan Wielemaker. SWI-Prolog home page. <http://www.swi-prolog.org>.

A Correctness proofs

Detailed definitions of execution state, transition and derivation can be found in [5]. The summary from section 2 should suffice to understand the theorems and proofs below.

First we prove a lemma which will be useful later. Intuitively it says that for every point in a derivation (under ω_r semantics) where a rule can directly be applied with c being the active constraint, there must be an earlier execution state in which the first occurrence of c is about to be checked and where all preconditions for that rule to fire are also fulfilled.

Lemma 1. *If in a derivation $s_0 \mapsto^* s_k$ for P under ω_r semantics, the execution state s_k is of the form $s_k = \langle [c\#i : j|A_k], S_k, B_k, T_k \rangle_{n_k}$, and transitions $s_k \mapsto_{\text{simplify}} s_{k+1}$ or $s_k \mapsto_{\text{propagate}} s_{k+1}$ are applicable, applying rule R_x , then the derivation contains an intermediate execution state $s_l = \langle [c\#i : 1|A_l], S_l, B_l, T_l \rangle_{n_l}$, such that $s_0 \mapsto^* s_l \mapsto^* s_k$ and for every execution state s_m with $l \leq m \leq k$, the CHR store contains all partner constraints needed for the application of rule R_x and the built-in store entails the guard of rule R_x .*

Proof. Consider the execution state

$$s_{l'} = \langle [c\#i : 1|A_{l'}], S_{l'}, B_{l'}, T_{l'} \rangle_{n_{l'}} \quad (s_0 \mapsto^* s_{l'} \mapsto^* s_k)$$

just after the last **Reactivate** transition that put $c\#i : 1$ at the top of the execution stack; if there was no such transition, consider $s_{l'}$ to be the execution state just after the **Activate** transition that put $c\#i : 1$ at the top of the execution stack.

Suppose at some point in the derivation $s_{l'} \mapsto^* s_k$, the built-in store does not entail the guard g_x of R_x . Then the built-in store has to change between that point and s_k , so that after the change it does entail g_x . This will possibly trigger some constraints:

- If c is triggered, then c is reactivated *after* $s_{l'}$, which is a contradiction given the way we defined $s_{l'}$.
- If another constraint d from the head of R_x is triggered, it becomes the active constraint. Now there are two possibilities:
 - (a) All constraints from the head of R_x are in the CHR store. This means eventually, either rule R_x will be tried with d as the active constraint, or another partner constraint gets triggered (but not c , because of how we defined $s_{l'}$), in turn maybe triggering other partner constraints, but any way R_x will be tried with one of the partner constraints as the active constraint. Because the built-in store now does entail g_x , the rule fires and a tuple is added to the propagation history. In execution state s_k , this tuple will still be in the propagation history, preventing the application of rule R_x . This is of course a contradiction.
 - (b) Not all constraints from the head of R_x are in the CHR store, so some have to be added before s_k is reached, and a similar early-firing happens at the moment the last partner constraint is added, also leading to a contradiction.

- If none of the constraints from the head of R_x are triggered, some of them are not in the CHR store yet, because if they are all there, at least one of them should be triggered, otherwise the change in the built-in store would not affect the entailment of g_x . As a result, some of the constraints occurring in the head of R_x have to be added before s_k is reached so we get a similar early-firing situation as above, again leading to a contradiction.

All these cases lead to a contradiction, so our assumption was wrong. This shows that during the derivation $s_{l'} \rightsquigarrow^* s_k$, the built-in store always entails the guard of R_x .

Suppose at some point in the derivation $s_{l'} \rightsquigarrow^* s_k$, the CHR store does not contain all partner constraints needed for rule R_x . Then somewhere in that derivation the last of these partner constraints (d) is added to the CHR store, so all constraints needed for R_x are in the CHR store. However, the only transition that could have added d to the CHR store is **Activate**, which also makes d the active constraint. We get an early-firing situation like above because the guard of R_x is entailed and every partner constraint (including c) is now in the CHR store. So we get a contradiction, proving that during the derivation $s_{l'} \rightsquigarrow^* s_k$, the CHR store always contains all constraints needed for rule R_x .

To conclude our proof: we have found an execution state s_l with the required properties, namely $s_l = s_{l'}$. \square

Using the previous lemma we now show that the “no earlier subrule fired” formula $\text{nesr}(R_i)$ is logically implied by the built-in store at the moment the rule R_i is applied.

Lemma 2. *If for a given CHR program P , the rule containing the j^{th} occurrence of the CHR predicate c is $R_{c,j}$, and if there is a derivation $s_0 \rightsquigarrow^* s_k = \langle [c\#i : j|A], S, B, T \rangle_n$ for P under ω_r semantics, and rule $R_{c,j}$ can be applied in execution state s_k , then we have $\mathcal{D} \models B \rightarrow \exists_B \text{nesr}(R_{c,j})$.*

Proof. From the previous lemma follows the existence of an intermediate execution state s_l ($0 \leq l \leq k$), such that for every execution state s_m with $l \leq m \leq k$, the CHR store contains all partner constraints needed for the application of rule $R_{c,j}$ and its guard is entailed by the built-in store.

To prove $\mathcal{D} \models B \rightarrow \exists_B \text{nesr}(R_{c,j})$, we show that

$$\forall R_a \in P : (R_a \prec R_{c,j} \wedge H_a^r \neq \epsilon) \Rightarrow (\mathcal{D} \models B \rightarrow \exists_B \neg(\theta_a \wedge g_a))$$

Suppose this is not the case, so assume there exists a non-propagation rule R_a such that $R_a \prec R_{c,j}$ and $\mathcal{D} \models B \wedge \theta_a \wedge g_a$. Since $R_{c,j}$ can be applied in execution state s_k , there exists a matching substitution σ matching c and constraints from S to corresponding head constraints of the rule $R_{c,j}$. Because $R_a \prec R_{c,j}$, there exists a number $o_a < j$ such that the o_a^{th} occurrence of c is in rule R_a . There exists an execution state $s_m = \langle [c\#i : o_a|A_m], S_m, B_m, T_m \rangle_{n_m}$ with $l \leq m < k$. From this state, a **Simplify** or **Propagate** transition can fire, applying rule R_a , because:

- all partner constraints are present in S_m ;
- there exists a matching substitution θ that matches c and partner constraints from the CHR store to the head constraints of R_a , namely $\theta = \theta_a \wedge \sigma$;
- the guard g_a is entailed because of our assumption;
- the history does not already contain a tuple for this instance, because R_a removes some of the constraints in its head.

But this application of R_a removes constraints needed for the rule application in s_k , because every head constraint of R_a also appears in $R_{c,j}$. This results in a contradiction. So our assumption was false, and $\mathcal{D} \models B \rightarrow \exists_B \text{nesr}(R_{c,j})$. \square

Now we are ready for a theorem stating that guard simplification does not affect the applicability of transitions. Correctness of guard simplification with respect to operational equivalence [1] is a trivial corollary of this theorem.

Theorem 3 (Guard simplification & transitions). *Given a CHR program P and its guard-simplified version $P' = GS(P)$. Given an execution state $s = \langle A, S, B, T \rangle_n$ occurring in some derivation for the P program under ω_r semantics, exactly the same transitions are possible from s for P and for P' . In other words, $\succrightarrow_P \equiv \succrightarrow_{P'}$.*

Proof. The **Solve**, **Activate** and **Reactivate** transitions do not depend on the actual CHR program, so obviously their applicability is identical for P and P' . The applicability of **Drop** only depends on the heads of the rules in the program, so again it is identical for P and P' .

If a **Simplify** or **Propagation** transition is possible for P , this means $A = [c\#i : j|A']$ and $\mathcal{D} \models B \rightarrow \exists_B g_k$, where k is the rule number of the j^{th} occurrence of c . According to lemma 2, we now know that $\mathcal{D} \models B \rightarrow \exists_B \text{nesr}(R_k)$. The rule R'_k is identical to R_k except for its guard g'_k , so the same transition is possible for P' unless the guard g'_k fails (while g_k succeeds). This can only happen if for some part $g_{k,x}$ of the conjunction g_k we have $\mathcal{D} \models \exists_B \text{nesr}(R_k) \wedge \bigwedge_{m < x} g_{k,m} \rightarrow \neg g_{k,x}$. Now we can derive a contradiction: $\mathcal{D} \models B \rightarrow \exists_B \text{nesr}(R_k)$ and $\mathcal{D} \models B \rightarrow \exists_B g_k$ combined with the previous statement gives $\mathcal{D} \models B \rightarrow \neg \exists_B g_k$ because of course $\forall m \models g_k \rightarrow g_{k,m}$.

If a **Simplify** or **Propagation** transition is possible for P' , this means $A = [c\#i : j|A']$ and $\mathcal{D} \models B \rightarrow \exists_B \text{nesr}(R_k)$. Again, assume the j^{th} occurrence of c is in the k^{th} rule. The same transition is also possible for P , unless for some x , $\mathcal{D} \models B \rightarrow \neg \exists_B g_{k,x}$. If there is more than one of such x 's, choose the smallest one, i.e. let $g_{k,x}$ be the first part of the guard conjunction that fails. Note that $\mathcal{D} \models B \rightarrow \exists_B \bigwedge_{m < x} g_{k,m}$. Because $\mathcal{D} \models B \rightarrow \exists_B g'_{k,x}$, we know that $g_{k,x} \neq g'_{k,x}$, and because of the definition of guard simplification, this can only be the case if $\mathcal{D} \models \text{nesr}(R_k) \wedge \bigwedge_{m < x} g_{k,m} \rightarrow g_{k,x}$. Again, this results in a contradiction, so the applicability of **Simplify** and **Propagation** is identical for P and P' .

Since the applicability of **Default** only depends on the applicability of the other transitions, it is also identical for P and P' . We showed that the applicability of any of the seven possible transitions is unchanged by guard simplification, concluding our proof. \square

Corollary 1 (Correctness of GS). *Under the refined operational semantics, any CHR program P and its guard-simplified version P' are operationally equivalent.*

Proof. According to the previous theorem, $\succrightarrow_P \equiv \succrightarrow_{P'}$, so all states are trivially P, P' -joinable. \square

For definitions of operational equivalence and joinable states we refer the reader to [1]. Now we can show that subsumable occurrences may be skipped (i.e. can be made passive). More formally:

Theorem 4 (Correctness of Occ. Subsumption). *Given a CHR program P and an ω_r derivation for P in which an execution state $s = \langle [c\#i : j|A], S, B, T \rangle_n$ occurs. If c_j is occurrence subsumable, **Simplify** and **Propagate** transition cannot (directly) be applied on state s .*

Proof. Suppose the **Simplify** or **Propagate** transition can be applied, firing rule R . Using the notation from definition 10, this means that $\mathcal{D} \models B \rightarrow \exists_B \text{fc}(R, c_j)$. Also, lemma 2 tells us that $\mathcal{D} \models B \rightarrow \exists_B \text{nesr}(R)$. Because of lemma 1 we know that rule R has been tried for the earlier occurrences of c in that rule. These tries must have failed, because R is a constraint-removing rule (c_j is occurrence subsumable) which cannot be applied twice on the same constraints. So

$$\forall k : m \leq k < j \Rightarrow \mathcal{D} \models B \rightarrow \neg \exists_B \theta_k(\text{fc}(R, c_k))$$

where θ_k is a renaming such that $\theta_k(c_k) = c_j$. This is equivalent to $\mathcal{D} \models B \rightarrow \exists_B \text{neocc}(R, c_j)$. Because c_j is occurrence subsumable, we have $\mathcal{D} \models B \rightarrow \neg \exists_B \text{fc}(R, c_j)$, which results in a contradiction. So the **Simplify** and **Propagate** transitions are indeed not applicable in state s . \square