Guard and Continuation Optimization for Occurrence Representations of CHR

Jon Sneyers*, Tom Schrijvers**, Bart Demoen

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

Abstract. Constraint Handling Rules (CHR) is a high-level rule-based language extension, commonly embedded in Prolog. We introduce a new occurrence representation of CHR programs, and a new operational semantics for occurrence representations, equivalent to the widely implemented refined operational semantics. The occurrence representation allows in a natural way to express guard and continuation optimizations, which remove redundant guards and eliminate redundant code for subsumed occurrences. These optimizations allow CHR programmers to write self-documented rules with a clear logical reading. We show correctness of both optimizations, present an implementation in the K.U.Leuven CHR compiler, and discuss speedup measurements.

1 Introduction

Constraint Handling Rules (CHR) is a high-level multi-headed rule-based programming language extension originally designed for writing constraint solvers. We assume that the reader is familiar with CHR, referring to [5, 4] for an overview. Our optimizations are formulated independent of host language, but the implementation and examples described in this paper are in Prolog.

The original theoretical operational semantics for CHR (ω_t), defined in [5], is nondeterministic since it does not specify the order in which rules are tried. However, all recent implementations use a more specific operational semantics, called the *refined* operational semantics (ω_r) [4]. In ω_r, the rules are tried in textual order. In practice, CHR programmers use the ω_r semantics. Their programs possibly are non-terminating or produce unintended results under ω_t semantics.

The dilemma CHR programmers face is the following: either they make sure their programs are valid under ω_t semantics, or they write programs that behave correctly only under ω_r semantics. Sticking to ω_t semantics results in more declarative code with a clear logical reading. Using ω_r semantics can result in more efficient code and allows easier implementation of programming idioms like key lookup. However, source code readability decreases significantly since rules not necessarily contain all preconditions for applying it: preconditions that are implicitly entailed by the rule order are often omitted by the programmer.

* This work was partly supported by project G.0144.03 funded by F.W.O.-Vlaanderen.
** Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

In this paper, we propose compiler optimizations that are a major step towards allowing CHR programmers to write more readable and declarative programs without sacrificing efficiency. They automatically remove redundant guard conditions, on the occurrence level. This paper extends [11, 13], which introduced guard simplification, a special case of guard optimization for rules.

The optimizations presented here are mainly based on reasoning about the guards of the CHR rules. This work is orthogonal to other optimization techniques, and they can be combined effortlessly, as we have done in the K.U.Leuven CHR compiler [8].

The next section intuitively describes the optimizations by discussing some examples. Section 3 introduces a new operational semantics for occurrence representations of CHR programs, equivalent to the refined operational semantics. Then, in Section 4, the new guard and continuation optimizations are defined formally, and their correctness is showed w.r.t. this new semantics. Section 5 briefly discusses an implementation of the optimizations in the K.U.Leuven CHR system, and the speedups we have measured. Finally, we conclude in Section 6.

2 Motivating Examples

Example 1 (guard optimization).

```
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. Hence this guard will always succeed, and 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 [10] to detect that `sign/2` is never-stored. □

Example 2 (types and modes).

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

We consider head matchings to be an implicit part of the guard: the last rule can be written as "`sum(A,S) <=> A = [X|Xs] | sum(Xs,T), S is X + T.`". Guard optimization can be much more effective if the types (and modes) of constraint arguments are known. If we know the first argument of constraint `sum/2` is an instantiated list, these two simplification rules cover all possible cases and thus the constraint is never-stored. In [9], optional mode declarations

were introduced to specify the mode of constraint arguments: ground (+) or unknown (?). Inspired by the Mercury type system [14], we have added optional type declarations to define types and specify the type of constraint arguments. For the above example, the CHR programmer would add:

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

The first line is a recursive and generic type definition for lists of type `T`, where `T` can be instantiated with built-in types like `int`, `float`, the general type `any`, or a user-defined type. The constraint declaration on the second line includes mode and type information. Using this knowledge, we can rewrite the last rule to "`sum(A,S) <=> true | A = [X|Xs], sum(Xs,S2), S is X + S2.`", keeping its behavior intact while again helping never-stored analysis. □

Note that it is often crucial to provide type and mode information to get the optimization results presented in this paper.

Example 3 (occurrence subsumption).

(a) `a(X,A,B), a(X,C,D) <=> A < B, C < D |` Body.
(b) `b(X,Y,Z), b(Y,Z,X), b(Z,X,Y) <=>` Body.
(c) `c(X,Y,Z), c(Y,Z,X), c(Z,X,Y) <=> (p(X); p(Y)) |` Body.
(d) `d(A,B), d(B,C) <=> A \== C |` Body.
`d(A,B), d(B,C) <=>` Body.

In examples (a) to (d) above, underlined occurrences are *subsumed* by earlier occurrences: the head constraints and guard are symmetric. Because a constraint is removed by a rule application the subsumed occurrences are redundant. The underlined occurrences are derived to be `passive`, meaning they can be skipped in the execution of the program. In the occurrence representation (introduced in the next section) we can express this by setting the guard for these occurrences to `fail`.

A strong occurrence subsumption analysis takes away the need for CHR programmers to write `pragma passive` declarations to improve efficiency, since the compiler automatically adds such declarations. As a result, the CHR source code contains less non-declarative operational pragmas, improving compactness and readability. □

Examples 1 and 2 may seem trivial, and similar optimizations have been proposed in the context of Prolog, but example 3 is very specific for CHR. They are all covered by the optimizations introduced in Section 4.

3 Semantics for Occurrence Representations

We will use $[H|T]$ to denote a list where the first element is H and remaining elements are T; $+\!\!+$ for list concatenation and $\square$ or `[]` for the empty list; $\uplus$ for multiset union, $\sqcap$ for multiset intersection, and $\sqsubseteq$ for multiset subset. We will

sometimes abuse notation by implicitly converting between lists and multisets, sets and multisets, and lists and conjunctions (where the conjunction is evaluated in the same order as the order of the list elements). We use $vars(E)$ to denote the variables of a syntactic expression E.

3.1 CHR Programs

CHR constraint symbols are drawn from the set of predicate symbols, denoted by a functor/arity pair. CHR constraints are atoms constructed from these symbols. To improve readability, we will often omit the arguments of CHR constraints. 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 the underlying constraint solver of the host language. We will assume this underlying solver supports at least equality, `true` and `fail`.

Definition 1 (CHR program). *A CHR program P is a list of CHR rules R_i of the form $H_i^k \setminus H_i^r \iff g_i \mid B_i$ where H_i^k and H_i^r (kept/removed heads) are lists of CHR constraints ($H_i = H_i^k +\!\!+ H_i^r \neq \square$); g_i (guard) is a list of built-in constraints; B_i (body) is a list of constraints.*

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 it is a *simpagation* rule. The guard and body of a rule are often treated as conjunctions. 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 [3].

Note that built-in constraints used in the guard are always *ask*-constraints when variables occurring in the head are involved. For example, the rule `p(X) <=> X = foo | B` is identical to the rule `p(X) <=> X == foo | B`. In other words, guards cannot modify variable bindings of constraint arguments.

3.2 Occurrence Representation

Most CHR compilers generate one block of host-language code (e.g. one Prolog clause) for every occurrence of a constraint in the head of a rule. Therefore it makes sense to represent a CHR program on the occurrence-level instead of the rule-level. This *occurrence representation* corresponds more closely to the generated code of current compilers. Its finer granularity allows the formulation of more powerful optimizations.

The head constraint occurrences are numbered from top to bottom and from left to right (but first the removed constraints, then the kept constraints). The i-th occurrence of a constraint c is denoted as $c : i$, the number of the rule in which it occurs as $\mathrm{rnum}(c : i)$. We will write $\mathrm{Occ} = \mathrm{Occ}^k \cup \mathrm{Occ}^r$ for the set of all occurrences of a given CHR program, where Occ^k are occurrences from the kept heads H_i^k and Occ^r are occurrences from the removed heads H_i^r.

Definition 2 (Occurrence representation). *An occurrence representation $O = (g, b, p_r, p_k, n_s, n_f)$ of a CHR program P is a 6-tuple of functions:*

– *g maps occurrences to lists of built-in constraints;*
– *b maps occurrences to lists of any constraints;*
– *p_r (p_k) map occurrences to subsets of* Occ^r *(*Occ^k*);*
– *n_s and n_f map occurrences to occurrences.*

We say g returns the *guard* for an occurrence, b returns the *body*, p_r returns the *removed partner* constraint occurrences and p_k returns the *kept partner* constraint occurrences, n_s returns the *success continuation* occurrence and n_f returns the *fail continuation* occurrence. For an occurrence $o \in \mathrm{Occ}$, we will write $p(o)$ to denote $p_k(o) +\!\!+ p_r(o)$ and $h(o)$ to denote $[o|p(o)]$.

3.3 The ω_o Semantics for Occurrence Representations

In this section we will introduce the call-based refined operational semantics for occurrence representations, referred to as ω_o semantics. It is a variant of the call-based refined operational semantics ω_c [10], formulated in terms of occurrence representations. The ω_c semantics is an equivalent variant of the refined operational semantics ω_r [4]. The difference between these two semantics lies in their formulation. The transition system of ω_r linearizes the dynamic call-graph of CHR constraints into the execution stack of its execution states. However, in ω_c (and ω_o), constraints are treated as procedure calls: each newly added *active* constraint searches for possible matching rules in order, until all matching rules have been executed or the constraint is deleted from the store. As with a procedure, when a matching rule fires, other CHR constraints may be executed as subcomputations and, when they finish, the execution returns to finding rules for the current active constraint. The latter semantics are much closer to the procedure-based target languages of current CHR compilers, like Prolog and HAL. This makes the ω_c (and ω_o) semantics much more suitable for reasoning about optimizations.

Execution State of ω_o. An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with some unique integer i. This number serves to differentiate among copies of the same constraint. An *occurrenced* identified CHR constraint $c : j\#i$ is an identified CHR constraint associated with an integer j, indicating that only matches with occurrence $c : j$ should be considered (in other work, the notation $c\#i : j$ is used). We introduce functions $\xi(c\#i : j) = \xi(c\#i) = \xi(c) = c$ and $\mathrm{id}(c\#i : j) = \mathrm{id}(c\#i) = i$, and extend them to lists, sets and multisets of (identified) CHR constraints in the obvious manner.

The execution state of the ω_o semantics is identical to the execution state of the ω_c semantics: it is a tuple $\langle G, A, S, B, T\rangle_n$ where G, A, S, B, T and n, represent the goal, call stack, CHR store, built-in store, propagation history and next free identifier respectively. We use σ_i to denote execution states.

The *goal* G is a list of CHR constraints and built-in constraints. The *execution stack* A is a list of identified CHR constraints, with a strict ordering where the top-most constraint is called *active*. The *CHR constraint store* S is a

multiset of identified CHR constraints. The *built-in constraint store* B is a conjunction of built-in constraints that have been passed to the underlying solver. The *propagation history* T is a set of lists, each recording the identities of the CHR constraints which fired a rule, and the number of the rule itself. This is necessary to prevent trivial non-termination for propagation rules: a propagation rule is allowed to fire on a set of constraints only once. Finally, the *next free identifier* n represents the next integer which can be used to identify a CHR constraint. Given an initial goal G, the initial state is $\langle G, \Box, \emptyset, \emptyset, \emptyset \rangle_1$.

Definition 3 (Matching conditions). *Given an occurrence representation O. For every occurrence $o \in \mathrm{Occ}$ and multisets S, K and R of CHR constraints (possibly identified and/or occurrenced), we define the following two conditions:*

$$\mathsf{sat}_{kr}(o, S) \triangleq g(o) \wedge \big(\xi(h(o)) \trianglelefteq \xi(S)\big)$$

$$\mathsf{sat}_{h}(o, K, R) \triangleq g(o) \wedge \big(\xi(p_k(o)) = \xi(K)\big) \wedge \big(\xi(p_r(o)) = \xi(R)\big)$$

If S is the CHR store and the built-in store entails $\mathsf{sat}_{kr}(o, S)$, then the rule of occurrence o can be applied. The condition $\mathsf{sat}_{h}(o, K, R)$ is used when we need to distinguish between the kept and removed partner constraints. Note that $\mathsf{sat}_{kr}(o, S) \Leftrightarrow \exists K, R\ \big(\mathsf{sat}_{h}(o, K, R) \wedge \xi(K \uplus R \uplus \{o\}) \trianglelefteq \xi(S)\big)$.

Transition Rules of ω_o. Execution proceeds by exhaustively applying transitions to the initial execution state until the built-in solver state is unsatisfiable or no transitions are applicable. We denote transitions from state σ_0 to σ_1 with $\sigma_0 \rightarrowtail_N \sigma_1$ where N is the (shorthand) name of the transition. We define $\rightarrowtail^*$ to be the reflexive transitive closure of $\rightarrowtail$.

We define $\mathit{solutions}_V(B)$ to be the set of all substitutions (unordered assignments to all variables of V) satisfying B. We say a set of variables $I \subseteq V$ is independent for c w.r.t. B iff $\mathit{solutions}_I(B) = \mathit{solutions}_I(B \wedge c) = Y$ and X_1 and X_2 exist such that $X_1 \times Y = \mathit{solutions}_V(B)$ and $X_2 \times Y = \mathit{solutions}_V(B \wedge c)$ where $\times$ denotes the Cartesian product of two sets (ignoring order). We define $\mathit{affected\_vars}_B(c) \triangleq \mathit{vars}(B \wedge c) \setminus I$, where I a maximal independent (for c w.r.t. B) subset of $\mathit{vars}(B \wedge c)$. Intuitively, when adding a built-in constraint c to the built-in store B, we have to trigger at least the CHR constraints containing one or more affected variables (variables from $\mathit{affected\_vars}_B(c)$).

The possible transitions are defined in Figure 1. The actual definition of the *solve* function will depend on the built-in solver. The lower bound of the ω_c semantics is a subset of the lower bound defined here. However, for the Herbrand solver (the built-in solver of Prolog), this lower bound corresponds to current implementations: it boils down to triggering the constraints containing a variable that is touched (instantiated or bound to another variable) by adding c. For other host languages, it might not be feasible (or worth the overhead) to implement this lower bound. The new lower bound for Solve is closer to current implementations, and avoids references to guards. Because of this change, ω_o semantics may demand more constraints to be triggered (without causing any

additional rule applications). However, the lower bound of ω_c is much harder to compute, possibly causing more overhead than what is gained by avoiding redundant constraint triggering.

The transitions in Figure 1 are a formulation of the ω_c semantics in terms of occurrences, except for the Solve transition. In the following, we will consider the original definitions of ω_c semantics [10] and ω_r semantics [4], where the definition for Solve has been replaced by the one described above. We will use $\overset{\omega_o}{\longmapsto}_O$ to denote $\rightarrowtail^*$ under ω_o semantics for an occurrence representation O, and $\overset{\omega_c}{\longmapsto}_P$ ($\overset{\omega_r}{\longmapsto}_P$) to denote $\rightarrowtail^*$ under ω_c (ω_r) semantics for a CHR program P.

3.4 Properties of ω_o semantics

Definition 4 (Refined Occurrence Representation). *The refined occurrence representation $O_{\text{ref}}(P) = (g, b, p_r, p_k, n_s, n_f)$ for a CHR program P (notation as in definition 1) is defined as follows: for every occurrence $c : i \in \text{Occ}$:*

$$\begin{array}{ll} g(c:i) = g_{\text{rnum}(c:i)} & b(c:i) = B_{\text{rnum}(c:i)} \\ p_r(c:i) = H^r_{\text{rnum}(c:i)} \setminus \{c:i\} & p_k(c:i) = H^k_{\text{rnum}(c:i)} \setminus \{c:i\} \\ n_s(c:i) = n_f(c:i) = c:(i+1). & \end{array}$$

Theorem 1 (ω_r is equivalent to ω_o for O_{ref}). *For any CHR program P:*

$$\langle G, \Box, \emptyset, \emptyset, \emptyset\rangle_1 \overset{\omega_o}{\longmapsto}_{O_{\text{ref}}(P)} \langle \Box, \Box, S, B, T\rangle_n \quad \Leftrightarrow \quad \langle G, \emptyset, \emptyset, \emptyset\rangle_1 \overset{\omega_r}{\longmapsto}_P \langle \Box, S, B, T\rangle_n$$

Proof. For a refined occurrence representation $O_{\text{ref}}(P)$, the definition of ω_o transitions corresponds trivially to the definition of ω_c transitions. In [2], a proof is given for the equivalence of ω_c and ω_r semantics. Hence, P under ω_r and $O_{\text{ref}}(P)$ under ω_o are equivalent. □

Note that `pragma passive` constructions can be expressed using occurrence representations: if occurrence o is declared to be passive, it suffices to modify the continuation functions: all occurrences with (fail/success) continuation o should get a new (fail/success) continuation $n_f(o)$. If o is the first occurrence of some constraint, this approach will not prevent o from becoming active. An alternative way to express `pragma passive` is by replacing the guard of o by `fail`.

4 Guard and Continuation Optimizations

In this section, we will present optimizations that simplify the guard function and the continuation functions of (originally refined) occurrence representations, improving efficiency without affecting the behavior of the resulting program.

Definition 5 (Condition Simplification). *Given a condition g which is a conjunction of built-in constraints $g = g_1 \wedge g_2 \wedge \ldots \wedge g_n$ and a condition D, we define* $\mathsf{simpl}(g, D) = g' = g'_1 \wedge g'_2 \wedge \ldots \wedge g'_n$*, where*

$$g'_i \triangleq \begin{cases} \texttt{fail} & \text{if } \mathcal{D} \models D \wedge \bigwedge_{j<i} g_j \rightarrow \neg g_i \\ \texttt{true} & \text{if } \mathcal{D} \models D \wedge \bigwedge_{j<i} g_j \rightarrow g_i \text{ and } \mathcal{D} \not\models D \wedge \bigwedge_{j<i} g_j \rightarrow \neg g_i \\ g_i & \text{otherwise.} \end{cases}$$

1. Solve: $\langle c, A, S, B, T\rangle_n \rightarrowtail_{So} \langle \Box, A, S', B', T'\rangle_{n'}$
where c is a built-in constraint. If $\mathcal{D} \models \neg\bar{\exists}_\emptyset c \wedge B$, then $S' \triangleq S$, $B' \triangleq c \wedge B$, $T' \triangleq T$, $n' \triangleq n$. Otherwise ($\mathcal{D} \models \bar{\exists}_\emptyset c \wedge B$), there is a series of transitions $\langle S_1, A, S, c \wedge B, T\rangle_n \rightarrowtail^* \langle \Box, A, S', B', T'\rangle_{n'}$, where the triggered constraints $S_1 \triangleq \mathit{solve}(S, B, c)$ are a subset of S such that $L \subseteq S_1 \subseteq U$, where
• *Lower bound*: $L \triangleq \{x \in S \mid \mathit{vars}(x) \cap \mathit{affected\_vars}_B(c) \neq \emptyset\}$
• *Upper bound*: $U \triangleq \{x \in S \mid \mathit{vars}(x) \not\subseteq \mathit{fixed}(B)\}$ where $\mathit{fixed}(B)$ is the set of variables fixed by B ($v \in \mathit{fixed}(B)$ if $\mathcal{D} \models \bar{\exists}_v(B) \wedge \bar{\exists}_{\rho(v)}\rho(B) \rightarrow v = \rho(v)$ for arbitrary renaming ρ). Hence, ground constraints are not triggered.

2a. Activate: $\langle c, A, S, B, T\rangle_n \rightarrowtail_A \langle c : 1\#n, A, \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$
where c is a (non-identified) CHR constraint.

2b. Reactivate: $\langle c\#i, A, S, B, T\rangle_n \rightarrowtail_R \langle c : 1\#i, A, S, B, T\rangle_n$
where $c\#i$ is a CHR constraint in the store (back in the queue through **Solve**).

---

3. Drop: $\langle c : j\#i, A, S, B, T\rangle_n \rightarrowtail_{Dp} \langle \Box, A, S, B, T\rangle_n$ where $c : j \notin \mathrm{Occ}$.

---

4. Simplify: $\langle o\#i, A, H \cup S, B, T\rangle_n \rightarrowtail_{Si} \langle \Box, A, S', B', T'\rangle_{n'}$
where $o = c : j \in \mathrm{Occ}^r$ and $H = P_k \cup P_l \cup \{c\#i\}$, and

$$\langle \theta(b(o)), A, P_k \cup S, \theta \wedge B, T \cup \{h\}\rangle_n \rightarrowtail^* \langle \Box, A, S', B', T'\rangle_{n'}$$

where θ is a matching substitution such that $\mathcal{D} \models B \rightarrow \bar{\exists}_H \theta(\mathsf{sat}_h(o, P_k, P_r))$. Furthermore, $h \triangleq (\mathrm{id}(H), \mathrm{rnum}(o)) \notin T$. If no such matching substitution exists then $\langle o\#i, A, S, B, T\rangle_n \rightarrowtail_{\neg Si} \langle n_f(o)\#i, A, S, B, T\rangle_n$

---

5. Propagate: $\langle o\#i, A, \{c\#i\} \cup S, B, T\rangle_n \rightarrowtail_{Pr} \langle G, A, S_k, B_k, T_k\rangle_{n_k}$
where $o = c : j \in \mathrm{Occ}^k$. Let $S_0 \triangleq \{c\#i\} \cup S, B_0 \triangleq B, T_0 \triangleq T, n_0 \triangleq n$.
Now assume, for $1 \leq l \leq k$ and $k \geq 0$, there is a series of transitions

$$\langle C_l, [o\#i|A], S_{l-1} \setminus P_r^l, B_{l-1}, T_{l-1} \cup \{h_l\}\rangle_{n_{l-1}} \rightarrowtail^* \langle \Box, [o\#i|A], S_l, B_l, T_l\rangle_{n_l}$$

where $H = \{c\#i\} \cup P_k^l \cup P_r^l \subseteq S_{l-1}$ and $h_l = (\mathrm{id}(H), \mathrm{rnum}(o)) \notin T_{l-1}$, and a matching substitution θ_l exists such that $\mathcal{D} \models B_{l-1} \rightarrow \bar{\exists}_H \theta_l(\mathsf{sat}_h(o, P_k^l, P_r^l))$ and $C_l = \theta_l(b(o))$, where θ_l renames apart all variables only appearing in $g(o)$ and $b(o)$ (separately for each l). Furthermore, for $k+1$ no such transition is possible. The resulting goal G is $G \triangleq \Box$ if $\mathcal{D} \models \bar{\exists}_\emptyset(\neg B_k)$ (i.e. failure occurred), $G \triangleq n_f(o)\#i$ if $k = 0$ (i.e. the rule was not applied; in this case we annotate the transition with $\neg Pr$ instead of Pr) and otherwise ($k \geq 1$) $G \triangleq n_s(o)\#i$.

---

6. Goal: $\langle [c|C], A, S, B, T\rangle_n \rightarrowtail_G \langle G, A, S', B', T'\rangle_{n'}$ where $[c|C]$ is a list of built-in and CHR constraints, $\langle c, A, S, B, T\rangle_n \rightarrowtail^* \langle \Box, A, S', B', T'\rangle_{n'}$, and $G \triangleq \Box$ if $\mathcal{D} \models \neg\bar{\exists}_\emptyset B'$ (i.e. calling c caused failure), otherwise $G \triangleq C$.

Fig. 1. Transition rules of ω_o.

In other words, $\mathsf{simpl}(g, D)$ returns the result of removing parts of g that are entailed by D and earlier parts of g. For example:

$$\mathsf{simpl}(X > 3 \wedge Y < 1 \wedge X > 2, X \leq 3 \vee Y < 0) = X > 3 \wedge \mathtt{true} \wedge \mathtt{true}$$

The following implication is obviously satisfied:

$$\mathcal{D} \models D \rightarrow (g \leftrightarrow \mathsf{simpl}(g, D))$$

Definition 6 (Earlier occurrences condition). *For every occurrence $o \in$ Occ and multiset S, we define the following condition:*

$$\mathsf{EOC}(o, S) \triangleq \xi(h(o)) \trianglelefteq \xi(S) \wedge \bigwedge \Big\{ \neg\theta(\mathsf{sat}_{kr}(o', S)) \mid o' \prec o \wedge \xi(\theta(o')) \in \xi(h(o)) \Big\}$$

where $c : i \prec c : j$ iff $i < j$ and $(p_r(c : i) \neq \emptyset \vee c : i \in \mathrm{Occ}^r)$, and $d : i \prec c : j$ (for $d \neq c$) iff $\mathrm{rnum}(d : i) < \mathrm{rnum}(c : j)$ and $(p_r(d : i) \neq \emptyset \vee d : i \in \mathrm{Occ}^r)$.

This is a conjunction of negated matching conditions, for all possible matching substitutions θ and earlier occurrences o'. Intuitively, if S is (a subset of) the constraint store, the $\mathsf{EOC}(o, S)$ condition expresses that no earlier rules were applicable that would have removed a partner-constraint of o. Note that different matching substitutions θ must be considered for a previous occurrence $o' = c : i$ if the head constraints $h(o')$ contain at least two other occurrences of the constraint c or at least two occurrences of another constraint.

If mode or type information is available, it can be added to the $\mathsf{EOC}(o, S)$ conjunction without affecting the following results, as long as this information is correct at any given point in any derivation. For example, the EOC condition for the second occurrence of `sum/2` in Example 2 from Section 2 could be the following (the last part is derived from type information) :

$$\mathsf{EOC}(\mathtt{sum(A,B)}, S) = \mathtt{sum(A,B)} \in \xi(S) \wedge \neg\big(\mathtt{A=[]} \wedge \mathtt{sum(A,B)} \in \xi(S)\big) \wedge \big(\mathtt{A=[]} \vee \mathtt{A=[\_|\_]}\big)$$

4.1 Optimizations

Definition 7 (Guard optimization). *Given an occurrence representation $O = (g, b, p_r, p_k, n_s, n_f)$. Optimizing the guard of occurrence o results in an occurrence representation $G(O, o) = (g', b, p_r, p_k, n_s, n_f)$, where $\forall o' \neq o : g'(o') = g(o')$ and $g'(o) = \mathsf{simpl}(g(o), \mathsf{EOC}(o, h(o)))$.*

Let us illustrate this definition by considering the following example:

```
X in1 A:B <=> A>B | fail.
X in2 A:B <=> A =:= B | X is A.
X in3 A:B, X in4 C:D <=> A<B, C<D | X in max(A,C):min(B,D).
```

Computing $\mathsf{EOC}(\mathtt{X\ in_3\ A:B}, h(\mathtt{X\ in_3\ A:B}))$, we get $\neg(\mathtt{A} > \mathtt{B}) \wedge \neg(\mathtt{A} =:= \mathtt{B}) \wedge \neg(\mathtt{C} > \mathtt{D}) \wedge \neg(\mathtt{C} =:= \mathtt{D})$. Optimizing the guard of the third occurrence of `in/2` results in the empty guard `true`, because both `A<B` and `C<D` are entailed by the

above EOC condition. The (partial) entailment checker we have implemented is strong enough to discover such entailed conditions.

Now consider the EOC condition for occurrence $\texttt{in}_4$: $\mathsf{EOC}(\texttt{X in}_4\ \texttt{C} : \texttt{D}, S) = \neg(\texttt{C} > \texttt{D}) \wedge \neg(\texttt{C} =:= \texttt{D}) \wedge \neg(\texttt{C} < \texttt{D}) \wedge \ldots = \texttt{fail}$. Because anything is entailed by `fail`, the guard of the fourth occurrence is optimized to `fail`. This means we can skip this always-failing occurrence. Note that if we would have used the optimized guard for the third occurrence, we would get an EOC condition containing $\neg\texttt{true}$. The following continuation optimizations modify the continuation functions to skip occurrences like $\texttt{in}_4$.

Definition 8 (Failure Continuation optimization). *Given an occurrence representation $O = (g, b, p_r, p_k, n_s, n_f)$. Optimizing the failure continuation of occurrence o results in an occurrence representation $C_f(O, o) = (g, b, p_r, p_k, n_s, n'_f)$, where $\forall o' \neq o : n'_f(o') = n_f(o')$, and $n'_f(o) = n_f(n_f(o))$ if*

$$\mathcal{D} \models \mathsf{EOC}(n_f(o), h(n_f(o))) \wedge \neg\exists\theta\ \mathsf{sat}_{kr}(o, \theta(h(n_f(o)))) \rightarrow \neg g(n_f(o))$$

(otherwise $n'_f(o) = n_f(o)$).

In the above example, optimizing the failure continuation of $\texttt{in}_3$ results in $n'_f(\texttt{in}_3) = \texttt{in}_5$. Note that $\texttt{in}_5$ may be a non-existent occurrence.

Definition 9 (Success Continuation optimization). *Given an occurrence representation $O = (g, b, p_r, p_k, n_s, n_f)$. Optimizing the success continuation of occurrence o results in an occurrence representation $C_s(O, o) = (g, b, p_r, p_k, n'_s, n_f)$, where $\forall o' \neq o : n'_s(o') = n_s(o')$, and $n'_s(o) = n_f(n_s(o))$ if*

$$\mathcal{D} \models \mathsf{EOC}(n_s(o), h(n_s(o))) \wedge g(o) \rightarrow \neg\theta(g(n_s(o)))$$

where $\xi(o) = \xi(\theta(n_s(o)))$; otherwise $n'_s(o) = n_s(o)$.

Consider the following example:

```
fib1(0,M) ==> M = 1.
fib2(1,M) ==> M = 1.
fib3(N,M) ==> N > 1 | fib(N-1,M1), fib(N-2,M2), M is M1 + M2.
```

Optimizing the success continuation of $\texttt{fib}_i$ (for $i \in \{1, 2\}$) results in $n'_s(\texttt{fib}_i) = \texttt{fib}_4$. Note that it is not meaningful to optimize the success continuation of an occurrence that is removed by applying its rule.

Note that the definitions of these optimizations crucially depend on entailments. This may be problematic because of the undecidability (in general) and complexity properties of testing entailment. We have implemented an incomplete entailment checker, which exhaustively propagates conditions entailed from the left hand side until the right hand side is found. Its worst-case time complexity is quite bad, but this seems not to be a problem in practice: in most CHR programs, constraints are defined by a small number of rules, so the EOC condition in the left hand side of the entailment is often small.

4.2 Correctness

Because of limited space, we will only present some results, without proof. The proofs are given in [12]. First we introduce the auxiliary notion of EOC-satisfying occurrence representations. Intuitively, such representations have increasing continuation functions and at any point in a derivation, the EOC condition for the active constraint is entailed by the built-in store.

Definition 10 (EOC-satisfying). *An EOC-satisfying occurrence representation $O = (g, b, p_r, p_k, n_s, n_f)$ is an occurrence representation where*

- $\forall\ c : i \in Occ : n_f(c : i) = c : j$ *and* $n_s(c : i) = c : k$ *where* $j > i$ *and* $k > i$;
- *If a derivation reaches an execution state* $\sigma_k = \langle c : j\#i, A, H \uplus S, B, T\rangle_n$, *and* $\xi(H) = \xi(\theta(h(c : j)))$, *then* $\mathcal{D} \models B \rightarrow \mathsf{EOC}(c : j, H)$.

Intuitively it should be clear that the refined occurrence representations from definition 4 have this property.

Lemma 1. *Refined occurrence representations are EOC-satisfying.*

We can show that for EOC-satisfying occurrence representations, the optimizations preserve applicability of ω_o transitions and EOC-satisfiability.

Lemma 2. *Any EOC-satisfying occurrence representation O is operationally equivalent w.r.t. ω_o semantics (in the strong sense: exactly the same transitions are applicable to any execution state in a derivation) to $G(O, o)$, $C_s(O, o)$ and $C_f(O, o)$, for every occurrence o.*

Lemma 3. *If an occurrence representation O is EOC-satisfying, then $G(O, o)$, $C_s(O, o)$ and $C_f(O, o)$ are also EOC-satisfying (for every occurrence o).*

Combining these results, we get the following correctness result:

Theorem 2 (Correctness). *Repeated application of G, C_s and C_f to a refined occurrence representation O results in an occurrence representation O' which is operationally equivalent to O w.r.t. ω_o semantics.*

5 Implementation and Experimental evaluation

We have implemented the optimizations in the K.U.Leuven CHR compiler [8], which can be found in recent releases of SWI-Prolog [15]. In our implementation, replacing parts of a rule guard by `true` is called guard simplification (see [13]), while replacing part of an occurrence guard by `fail` is called occurrence subsumption, and it basically causes the occurrence to be declared passive. Guard simplification is a special case of guard optimization, and occurrence subsumption corresponds to failure continuation optimization. Figure 2 illustrates the effect of the optimizations on a small example.

Experimental results. To get an idea of the efficiency gain obtained by our optimizations, we have measured the performance of several CHR benchmarks, both with and without the optimizations. All benchmarks were performed with SWI-Prolog [15] version 5.5.2, on a Pentium 4 (1.7 GHz) GNU/Linux machine with a low load. We have measured similar results [11] in hProlog [1].

```
:- op(700,xfx,in).

:- constraints
    in(?int,+interval).

:- chr_type
    interval ---> int:int.


X in A:B <=>
    A > B |
    fail.

X in A:B <=>
    A =:= B |
    X is A.

X in A:B, X in C:D <=>
    A < B, C < D |
    X in min(A,C):max(B,D).
```

(a) Example CHR program.

```
X in I :-
   'in/2__0'(X,I,_).

'in/2__0'(_,A:B,Z) :-
   A > B, !,
   remove_constraint(Z),
   fail.

'in/2__0'(X,A:B,Z) :-
   A =:= B, !,
   remove_constraint(Z),
   X is A.

'in/2__0'(X,A:B,Z) :-
   A < B,
   find_partner(in(Y,I)),
   Y == X,
   I = C:D, C < D, !,
   remove_constraint(Z),
   X in min(A,C):max(B,D).

'in/2__0'(X,C:D,Z) :-
   C < D,
   find_partner(in(Y,I)),
   Y == X,
   I = A:B, A < B, !,
   remove_constraint(Z),
   X in min(A,C):max(B,D).

'in/2__0'(X,I,Z) :-
   insert_constraint(Z)
```

(b) Not optimized.

```
X in I :-
   'in/2__0'(X,I,_).

'in/2__0'(_,A:B,Z) :-
   A > B, !,
   remove_constraint(Z),
   fail.

'in/2__0'(X,A:B,Z) :-
   A =:= B, !,
   remove_constraint(Z),
   X is A.

'in/2__0'(X,J,Z) :-

   find_partner(in(Y,I)),
   Y == X,
   !,
   remove_constraint(Z),
   J = A:B, I = C:D,
   X in min(A,C):max(B,D).

'in/2__0'(X,I,Z) :-
   insert_constraint(Z)
```

(c) Optimized.

Fig. 2. Comparing generated code. Note that the redundant clause for the fourth occurrence of `in/2` and the redundant guard of the last rule are removed in (c).

Figure 3 gives an overview of our results. The first column indicates the benchmark name and the parameters that were used. These benchmarks are available at [7]. The second column indicates whether the optimizations were enabled, where "type" means "yes and additional type information was provided". Mode declarations were provided for all programs, which allows a speedup factor of two to three [13] in these cases. We have measured the additional speedups on top of the speedups we get from using mode information. The next two columns show the size of the generated Prolog code, not including constraint-store related auxiliary predicates. The last column shows the runtime in seconds and a percentage comparing it to the non-optimized version.

Benchmark	*Optimize*	*# clauses*	*# lines*	*Runtime (%)*	
sum	no	3	10	5.03	(100)
(10000,500)	type	2	6	4.49	(89)
nrev	no	6	20	13.97	(100)
(30,50000)	type	4	11	8.44	(60)
dfsearch	no	4	16	37.58	(100)
(16,500)	yes	4	15	31.63	(84)
	type	3	11	29.97	(80)
bool_chain	no	180	2861	12.8	(100)
(200)	yes	147	2463	7.0	(55)
fib	no	10	154	11.2	(100)
(22)	yes	9	125	8.5	(76)
leq	no	18	218	14.1	(100)
(60)	yes	13	162	11.7	(83)

Fig. 3. Benchmark results.

We compared the `sum`, `nrev` and `dfsearch` benchmarks to a native Prolog version of the program. The native Prolog version turned out to be almost identical to the generated code for the CHR program, the only difference being some redundant cuts (`!/0`) in the latter. We could not measure any difference in runtime. There is no straightforward way to make a native Prolog version of the `bool_chain`, `fib` and `leq` benchmarks, since they crucially depend on storing constraints.

Overall, for these benchmarks, doing guard simplification and occurrence subsumption — combined with never-stored analysis and use of mode information to remove redundant variable triggering code — results in cleaner and more efficient code which is much closer to what a Prolog programmer would write. As a result, a major performance improvement is observed in these benchmarks.

Writing auxiliary predicates in Prolog. CHR programs that implement deterministic algorithms (like the first three benchmarks) have a relatively low performance when compiled naively using the general schema, compared to native Prolog versions. For that reason, CHR programmers usually write such algorithms as auxiliary predicates in Prolog instead of formulating them as CHR constraints. Such mixed-language programs often use inelegant 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. By implementing these parts as multi-headed CHR rules, the need for 'host-language interface' constraints like `getFoo/1` is drastically reduced. Thanks to our new optimizations and other analyses, the programmer can now implement the entire program in CHR, relying on the compiler to generate efficient code.

6 Conclusion

We have described new guard and continuation optimizations. We have defined them formally and showed their correctness, implemented them in K.U.Leuven CHR compiler and evaluated them experimentally. Guard optimization encourages CHR programmers to include all preconditions for rule application in the rule guards, since redundant tests are compiled out. Continuation optimization dramatically reduces the need for `pragma passive` directives. Hence, the optimizations allow writing CHR programs that are more declarative, readable and self-documenting, without sacrificing efficiency.

Our new optimizations contribute to the state-of-the-art level of performance of code generated by the K.U.Leuven CHR compiler. Guard optimization reduces the overhead of testing redundant guard conditions, while continuation optimization reduces the overhead of trying rules that are not applicable. Furthermore, our optimization helps other analyses (like the never-stored analysis) to reduce constraint store related overhead. Earlier work introduced mode declarations used for hash tabling and other optimizations. We have added type declarations for CHR programs. Using both mode and type information we have realized further optimization of the generated code.

Related work. The idea of continuation optimization was originally introduced in [6]. Guard optimization originates from [11, 13], where a weaker optimization called guard simplification was introduced. Guard simplification is basically guard optimization for every occurrence in a rule, which is weaker than guard optimization in the case of multi-headed rules.

Future work. It would be interesting to implement a stronger version of $\mathsf{simpl}(g, D)$ (Definition 5, page 7) by replacing an expensive condition g_j by a cheaper condition g'_j, as long as $\mathcal{D} \models D \wedge \bigwedge_{k<j} g_k \rightarrow (g_j \leftrightarrow g'_j)$. For example, consider the following program:

```
p(X) <=> X >= 0, g(X) | ...
p(X) <=> X < 0, \+ g(X) | ...
p(X) <=> g(X) | ...
```

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 $\neg(X >= 0 \wedge g(X)) \wedge \neg(X < 0 \wedge \neg g(X))$ entails $g(X) \leftrightarrow X < 0$.

Our current knowledge base for entailment checking is limited to the most common Prolog built-ins. To be able to recognize more redundant guards, one could extend this knowledge base, add support for additional declarations that would be added to the knowledge base during the program analysis, and even analyze the host-language implementation of user-defined predicates used in guards, inferring extensions to the knowledge base automatically.

When the "earlier occurrences condition" is large, compilation time may become an issue. We intend to improve the scalability of our implementation, although it does not present an immediate problem.

The information entailed by the failure and success of guards seems also useful in other program analyses and transformations. One application is program specialization: the code for executing a constraint is specialized for a particular call from an occurrence body. This may lead to the elimination of more redundant guards (and even redundant rules) for the specialized case.

Finally we would like to integrate our optimizations into the bootstrapped CHR compiler which is currently being developed by Christian Holzbaur et al.

References

1. Bart Demoen. hProlog home page. http://www.cs.kuleuven.ac.be/~bmd/hProlog.
2. Gregory Duck, Tom Schrijvers, and Peter Stuckey. Abstract Interpretation for Constraint Handling Rules. Report CW 391, K.U.Leuven, Department of Computer Science, Leuven, Belgium, September 2004.
3. Gregory Duck, Peter Stuckey, María García de la Banda, and Christian Holzbaur. Extending Arbitrary Solvers with Constraint Handling Rules. In D. Miller, editor, *Proceedings of the 5th International Conference on Principles and Practice of Declarative Programming (PPDP'03)*. ACM Press, 2003.
4. Gregory Duck, Peter Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *Proceedings of the 20th Intl. Conference on Logic Programming (ICLP'04)*, September 2004.
5. 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.
6. Christian Holzbaur, María García de la Banda, Peter Stuckey, and Gregory Duck. Optimizing compilation of Constraint Handling Rules in HAL. In *Special Issue of Theory and Practice of Logic Programming on CHR*, 2005. To appear.
7. Tom Schrijvers. CHR benchmarks and programs. Available at the K.U.Leuven CHR home page at http://www.cs.kuleuven.ac.be/~toms/Research/CHR/.
8. 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.
9. Tom Schrijvers and Thom Früwirth. Implementing and Analysing Union-Find in CHR. Technical Report CW 389, K.U.Leuven, Dept. Computer Science, July 2004.
10. 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.
11. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Simplification in CHR programs. Technical Report CW 396, K.U.Leuven, Dept. CS, November 2004.
12. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and Continuation Optimization for Occurrence Representations of CHR. Technical Report CW 420, K.U.Leuven, Dept. CS, July 2005.
13. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Simplification in CHR programs. In *Proceedings of the 19th Workshop on (Constraint) Logic Programming (W(C)LP'05)*, Ulm, Germany, February 2005.
14. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the 18th Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, 1995.
15. Jan Wielemaker. SWI-Prolog home page. http://www.swi-prolog.org.