Guard Simplification in CHR programs

Jon Sneyers
Tom Schrijvers
Bart Demoen

Report CW 396, November 2004



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

Guard Simplification in CHR programs

Jon Sneyers
Tom Schrijvers
Bart Demoen

Report CW 396, November 2004

Department of Computer Science, K.U.Leuven

Abstract

Constraint Handling Rules (CHR) is a high-level language commonly used to write constraint solvers. Most CHR programs depend on the refined operational semantics, resulting in an obfuscated logical reading and non-termination or worse under the theoretical operational semantics. We introduce a source to source transformation called *guard simplification* which allows CHR programmers to write self-documented rules with a clear logical reading. It improves performance by removing guards entailed by the implicit "no earlier (sub)rule fired" precondition and optional type and mode declarations. A correctness proof of the transformation is given, its implementation in the K.U.Leuven CHR compiler is presented and experimental results are discussed.

Keywords : Constraint Handling Rules, optimized compilation, program transformation, program analysis.
CR Subject Classification : D.3.2 [Programming Languages] Language Classifications — Constraint and logic languages; D.3.4 Processors — Code generation, Compilation, Optimization, Preprocessors.

Guard Simplification in CHR programs

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 language commonly used to write constraint solvers. Most CHR programs depend on the refined operational semantics, resulting in an obfuscated logical reading and non-termination or worse under the theoretical operational semantics. We introduce a source to source transformation called *guard simplification* which allows CHR programmers to write self-documented rules with a clear logical reading. It improves performance by removing guards entailed by the implicit "no earlier (sub)rule fired" precondition and optional type and mode declarations. A correctness proof of the transformation is given, its implementation in the K.U.Leuven CHR compiler is presented and experimental results are discussed.

1 Introduction

Constraint Handling Rules (CHR) is a high-level multi-headed rule-based programming language extension commonly used to write constraint solvers. We will assume the reader to be familiar with the syntax and semantics of CHR, referring to [5] for an overview. Examples are given in a Prolog context, although the results are valid in general.

The theoretical operational semantics ω_t of CHRs, as defined in [5], is relatively nondeterministic as the order in which rules are tried is not specified. However, all implementations of CHR we know of use a more specific operational semantics, called the *refined* operational semantics ω_r [4]. In ω_r, the order in which rules are tried is the textual order in which the rules occur in the CHR program. Usually, CHR programmers take this refined operational semantics into account when they write CHR programs. As a result, their CHR programs could be non-terminating or could even produce incorrect 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 only work correctly under ω_r semantics. Sticking to ω_t semantics has the advantage that it results in more declarative code with

* Research Assistant of the fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen)

a clear logical reading, but it has the disadvantages that it is harder to implement some programming idioms and that the compiled code is less efficient. Using ω_r semantics results in more efficient compiled code and allows easier implementation of some programming idioms like key lookup, but at a cost: it becomes much less obvious from the CHR program what the preconditions for application of a rule really are. Indeed, under ω_t semantics, rules have to contain in their guards all the preconditions needed, while under ω_r semantics, the CHR programmer can and does omit the preconditions that are implicitly entailed by the rule order. Omitting these redundant preconditions may contribute to more efficient compiled code, but at the same time it makes the program less self-documented.

In this paper, we propose a compiler optimization that is a major step towards allowing CHR programmers to write more readable and declarative programs while getting the same efficiency as programs written with the specifics of the refined operational semantics in mind. This optimization, called *Guard Simplification*, is a source-to-source transformation of CHR programs, removing redundant guard conditions (and head matchings, an implicit part of the guard) based on reasoning about behavior of the program under the refined operational semantics. The transformed program is simpler, possibly allowing more optimization from other analyses. For example, guard simplification can reveal the never-stored property [2], as we will show later. Thanks to guard simplification, the CHR programmer can focus on writing a declarative specification and rely on the compiler to produce efficient code.

This paper is structured as follows: The next section presents an intuitive overview of the reasoning behind guard simplification and its effects, illustrated with some examples. The related concept of head matching simplification is also introduced and the use of type and mode information to enhance guard and head matching simplification is discussed. In section 3, a formal definition of the guard simplification transformation is given, followed by a correctness proof. Section 4 deals with the implementation of the guard simplification analysis in the K.U.Leuven CHR compiler [8] and the effects of guard simplification on the generated code. Then, in section 5, the results of several benchmarks are discussed, in order to compare the efficiency of CHR programs before and after guard simplification. Finally, section 6 concludes this paper, summarizing our contributions and discussing related and future work.

2 Overview

Consider the following example CHR program computing the greatest common divisor of two integers using Euclid's algorithm:

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

A query containing two `gcd/1` constraints with integer arguments, say `gcd(9)` and `gcd(15)`, will eventually result in a constraint store containing one `gcd/1` constraint with the greatest common divisor in its argument. For example, starting with `gcd(9),gcd(15)`, the second rule will fire, resulting in `gcd(9),gcd(6)`. The second rule will fire again twice, resulting in `gcd(6),gcd(3)` and then `gcd(3),gcd(0)`. Now the first rule will fire, removing `gcd(0)` from the CHR store. The remaining constraint indeed contains the greatest common divisor of 9 and 15, namely 3.

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 we know the first rule is always tried first. In the theoretical operational semantics, this second version of the CHR program is no longer guaranteed to terminate, since applying the second rule indefinitely on a constraint store containing e.g. `gcd(3),gcd(0)` (which is a valid execution strategy under the theoretical operational semantics) results in an infinite loop.

The source to source transformation discussed in this paper transforms a CHR program P into another CHR program $P' = GS(P)$ which is equivalent under the refined operational semantics ω_r. Although the original program might have been valid under any execution strategy covered by the theoretical operational semantics ω_t, the transformed program will in general only show identical behavior when ω_r semantics are used. This is not an issue, since all recent CHR implementations use ω_r semantics.

2.1 Guard simplification

As mentioned earlier, under the refined operational semantics of CHRs, the order in which the rules are tried is the textual order of the rules in the CHR program. We number the rules accordingly, so that for $i < j$,

rule R_i appears before rule R_j in the CHR program. 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 with some simple examples.

Example 1: an 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 if they would have fired, 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 $N \leq 0$. The second rule, `zero`, did not fire either, so we derive that $N \neq 0$. Now we can combine these results to get $N < 0$, which is exactly the guard of the third rule. Because we know this guard will always be true, we can safely remove it. This will result in slightly more efficient generated code (because the redundant test is removed), but – more importantly – this might also be useful for other analyses. In this example, after the guard simplification, the *never-stored* analysis [2] is able to detect that the constraint `sign/2` is never-stored because now the third rule is an unguarded single-head simplification rule, removing all `sign/2` constraints immediately.

Example 2: a rule that can never fire

```
neq  @ p(A) \ q(B) <=> A \== B | ...
eq   @ q(C) \ p(C) <=> true | ...
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 of course a contradiction, so we know the third rule can never fire. Most often such never firing rules are in fact 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.

Generalizing from the previous examples, we can summarize guard simplification as follows: If a (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.

2.2 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:

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

Never-stored analysis as it is currently implemented in the K.U.Leuven CHR system is not able to detect `p/2` to be a never-stored constraint, because none of these two rules remove all `p/2` constraints. 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. Now we say the head matchings of the second rule are simplified, because the head contains less matching conditions. As a result, never-stored analysis can now detect `p/2` to be never-stored, and more efficient code can be generated.

2.3 Type and mode declarations

Head matching simplification can be much more effective if some knowledge of the argument types of constraints is given. Consider this example:

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

If 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. In [11], optional mode declarations were introduced to specify the mode – ground (+) or unknown (?) – of constraint arguments. Inspired by the Mercury type system [12], 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 the following lines:

```
option(type_definition,
       type(list(X), [ [], [X | list(X)] ])).
option(type_declaration, sum(list(int),int)).
option(mode, sum(+,?)).
```

The first line is a recursive and generic type definition for lists of some type `X`, a variable that can be instantiated with builtin types like `int`, `float`, the general type `any`, or any user-defined type. The next line says the first argument of constraint `sum/2` is of type 'list of integers' and the second is an integer. In the last line, the first argument of `sum/2` is declared to be ground on call while the second argument can be 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.`", keeping its behavior intact while again helping never-stored analysis to detect `sum/2` to be a never-stored constraint.

3 Formal description and proofs

We will now formalize the guard simplification transformation intuitively described above. We use $\subseteq$ to denote multiset subset, $+\!\!+$ for sequence concatenation and ϵ for empty sequences. Constraints are either CHR constraints or *builtin* 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 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*

$$R_i = \quad H_i^k \setminus H_i^r \iff g_i \mid B_i$$

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 builtin constraints, and B_i (body) is a conjunction of constraints. We will write H_i as a shorthand for $H_i^k +\!\!+ H_i^r$.

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 more detail in [3] and an illustrating example can be found e.g. in section 2.1 of [10].

3.1 Auxiliary definitions

We introduce some additional notation for the functor/arity of a constraint.

Definition 2 (Functor). *For every CHR constraint $c = p(t_1, \ldots, t_n)$, we define $functor(c) = p/n$. For every multiset C of CHR constraints we define $functor(C)$ to be the multiset $\{functor(c) | c \in C\}$.*

We will now 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 3 (Earlier subrule). *The rule R_j is an* earlier subrule *of rule R_i (notation: $R_j \prec R_i$) iff $j < i$ and $functor(H_j) \uplus functor(H_i)$.*

Now we can define a logical expression $nesr(R_i)$ stating the implications of the fact that all constraint-removing earlier subrules of rule R_i have been tried unsuccessfully.

Definition 4 ("No earlier subrule fired"). *For every rule R_i, we define:*

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

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

3.2 Guard simplification

Consider a CHR program P with rules R_i which have guards $g_i = \bigwedge_k g_{i,k}$. If we apply guard simplification to this program, we rewrite some guards to `true` (or `false`) if they (or their negations) are entailed by the "no earlier subrule fired" condition.

Definition 5 (Guard simplification). *Applying* guard simplification *to a CHR program P results in a new CHR program $P' = GS(P)$ with rules $R'_i = H_i^k \setminus H_i^r \iff \bigwedge_k g'_{i,k} \mid B_i$, where*

$$g'_{i,k} = \begin{cases} \texttt{true} & \text{if } \mathcal{D} \models nesr(R_i) \rightarrow g_{i,k}; \\ \texttt{false} & \text{if } \mathcal{D} \models nesr(R_i) \rightarrow \neg g_{i,k}; \\ g_{i,k} & \text{otherwise.} \end{cases}$$

We will now show that P and P' behave exactly the same way under the refined operational semantics.

3.3 Correctness proof

First we will 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 $s_0 \rightarrowtail^* s_k$ 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 $s_k \rightarrowtail_{simplify} s_{k+1}$ or $s_k \rightarrowtail_{propagate} s_{k+1}$ 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 $s_0 \rightarrowtail^* s_l \rightarrowtail^* s_k$ 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 builtin store entails the guard of rule R_x.*

Proof. Consider the execution state $s_{l'} = \langle [c\#i : 1|A_{l'}], S_{l'}, B_{l'}, T_{l'} \rangle_{n_{l'}}$ ($s_0 \rightarrowtail^* s_{l'} \rightarrowtail^* s_k$) 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 $s_{l'} \rightarrowtail^* s_k$, the builtin store does not entail the guard g_x of R_x. Then the builtin 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:
  - If 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 builtin store now does entail g_x, the rule will fire 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.

- If not all constraints from the head of R_x are in the CHR store, some will have to be added before s_k is reached, and a similar early-firing will happen at the moment the last partner constraint is added. So this also leads 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 builtin store would not affect the entailment of g_x. As a result, some of the constraints occurring in the head of R_x will 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 must have been wrong. This shows that during the derivation $s_{l'} \rightarrowtail^* s_k$, the builtin store always entails the guard of R_x.

Suppose at some point in the derivation $s_{l'} \rightarrowtail^* s_k$, the CHR store does not contain all partner constraints needed for rule R_x. Then somewhere in the derivation $s_{l'} \rightarrowtail^* s_k$ 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 $s_{l'} \rightarrowtail^* s_k$, 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 $s_l = s_{l'}$. □

Using the previous lemma we will now show that the "no earlier sub-rule fired" formula $nesr(R_i)$ is logically implied by the builtin store at the moment the rule R_i is applied.

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

Proof. From the previous lemma follows the existence of an intermediate execution state s_l ($0 < l < k$), 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 $R_{c,j}$.

To show that $\mathcal{D} \models B \rightarrow nesr(R_{c,j})$, it suffices to show the following:

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

Suppose this is not the case, so assume there would 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 will 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 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 will remove 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 $\mathcal{D} \models B \rightarrow nesr(R_{c,j})$. □

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 is a trivial corollary of this theorem.

Theorem 1 (Guard simplification and applicability of transitions). *Given a CHR program P and its guard-simplified version $P' = GS(P)$. Given an execution state $S_i = \langle A, S, B, T \rangle_n$ occurring in some derivation for the P program under ω_r semantics, exactly the same transitions are possible from S_i for P and for 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 $A = [c\#i : j | A']$. Assume the jth occurrence of c is in the kth rule of P.

According to lemma 2, we now know that $\mathcal{D} \models B \rightarrow nesr(R_k)$. Since the rule R'_k is identical to R_k except for its guard g'_k, the same transition is possible for P', because the only way this guard g'_k can fail when g_k succeeds is if for some part $g_{k,x}$ of the conjunction g_k we have $\mathcal{D} \models nesr(R_k) \rightarrow \neg g_{k,x}$. But this is impossible, since then $\mathcal{D} \models B \rightarrow \neg g_k$, and the transition was not possible in the first place for P.

By a similar argument, we get that if a **Simplify** or **Propagation** transition is possible for P', the same transition is also possible for P. So the applicability of **Simplify** and **Propagation** is also 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. □

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

Proof. According to the previous theorem, $\rightarrowtail_P \equiv \rightarrowtail_{P'}$, so all states are trivially P, P'-joinable. □

4 Implementation

We have implemented the presented analysis and transformation in the K.U.Leuven CHR compiler [8]. In the process, a separate module has been written for entailment checking. We will first give an overview of our implementation of guard simplification, which depends heavily on this entailment checker. Then we will explain how we implemented the entailment check module. Finally we will take a look at the generated code for an example CHR program, and how it is improved by guard simplification.

4.1 Overview

The guard simplification phase tries to rewrite every rule in the CHR program. In the rewritten rules, the redundant parts of the guard have been removed and the head matchings (an implicit part of the guard) are made as general as possible. As a result, the generated code will be more efficient because redundant checks are removed, and also the next compiler phases – like storage analysis – will be able to do more optimizations. Schematically, our implementation does the following for every rule R_i:

1. make head matchings explicit, inserting fresh variables in the arguments of head constraints as needed;
   E.g. change `c([X|Xs],Y,Y)` to `c(A,B,C)` with matching `A=[X|Xs], B==C`.
2. iteratively construct a conjunction similar to $nesr(R_i)$ from section 3, containing the negations of the guards of the earlier subrules $R_j \prec R_i$, consider all possible substitutions;
   E.g. for this program:
   ```
   r1@ c(X) <=> p(X) | ...
   r2@ c(2) <=> q | ...
   r3@ c(A), c(B) <=> ... | ...
   ```
   the following conjunction is computed:
   $nesr($`r3`$) =$`(A\==2 ∨ \+ q) ∧ (B\==2 ∨ \+ q) ∧ \+ p(A) ∧ \+ p(B)`.
3. add type information by looking up the type definitions corresponding to the argument types of the head constraints of R_i, unfolding them to the nesting depth needed;
   For recursive types like `list`, the type condition can be infinite. E.g. if `L` is of type `list`, the type condition would be `(L==[] ; (L=[A|B], (B==[] ; (B=[C|D], (D==[] ; (D=[E|F], ...))))))`. To prevent such infinite loops, we will stop at the highest nesting depth occurring in R_i and its earlier subrules. So for example for `sum/2` from section 2.3, the head constraint `sum(L,S)` would result in the condition `(L==[] ; (L=[A|_], integer(A))),(var(S) ; integer(S))`.
4. for every part of the guard of R_i (the $g_{i,k}$'s from section 3): 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`;
5. 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;
   E.g. we can rewrite `c([X|Xs],[Y|Ys],A,A,[B|Bs]) <=> B > 0 | d(X,A), c(Xs)` to `c(Z,_,A,A2,[B|Bs]) <=> B > 0 | Z=[X|Xs], d(X,A), c(Xs)` if the derived information entails that the first and second arguments are non-empty lists and the third and fourth argument are identical.
6. 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.

The negation of a condition is computed in a straightforward way for builtins[1] and for user-defined predicates `p` we simply use `(\+ p)`.

4.2 Checking entailment

The entailment checking module is used in the guard simplification analysis to test whether some condition B (e.g. $X < Z$) is entailed by another

[1] Some examples: `X < Y` → `X >= Y`, `X == Y` → `X \== Y`, `true` → `fail`, `\+ X` → `X`

condition A (e.g. $X < Y \wedge Y < Z$), i.e. $A \rightarrow B$. Since in general this problem is undecidable, the entailment checker will try to prove that B is entailed by A by propagating the implications of (host language) builtin conditions in A, like `<`, `=:=`, `functor/3`, `==` and unification, succeeding if B is found and failing otherwise. Hence if the entailment checker succeeds, $A \rightarrow B$ must hold, but if it fails, either $A \not\rightarrow B$ holds or $A \rightarrow B$ holds but was not detected. It does not try to discover implications of user-defined predicates, which would require a complex analysis of the host-language program. The core of this entailment checker is written in CHR. Schematically, it works as follows:

1. add the parts of the conjunction in A to the constraint store (wrapped in `known/1` constraints);
2. simplify the conditions to some normalized form (e.g. convert $\geq, >, <$ to $\leq$ and $\neq$) and evaluate ground conditions (e.g. remove $3 < 5$, replace $5 < 3$ by `fail`);
3. propagate entailed conditions until fixpoint (e.g. if $X \leq Y$ and $Y \leq Z$ are in the store, add $X \leq Z$);
4. if $known(B)$ is in the store: succeed;
5. if B is directly entailed by something in the store: succeed (e.g. $X < 3$ is entailed by $X < 0$, $X \leq 8$ is entailed by $X =:= 2$) – we need some special rules to cover these cases since we cannot simply propagate all these weak conditions entailed by a stronger one (there are infinitely much);
6. if the store contains a disjunction $A_1 \vee A_2$: add A_1 and test B, then backtrack, add A_2 and test B; succeed if both tests succeed, else fail;
7. otherwise: 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 $known(\neg B_2)$ to the store and test B_1. We can stop (and succeed) if B_1 is entailed, otherwise we backtrack, add $known(\neg B_1)$ to the store and return the result of testing entailment of B_2.

4.3 Generated code comparison

Let us now look at the Prolog code the CHR compiler generates for some example CHR program. Consider this fragment from a prime number generating program from the CHR web site [13]:

```
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 the typical 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 [7], the generated code looks like:

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

% first occurrence
filter(List,P,Out,C) :-
       nonvar(List), List = [X|In], 0 =\= X mod P, !,
       ... % remove from constraint store if needed
       Out = [E|Out1], filter(In,P,Out1) .

% second occurrence
filter(List,P,Out,C) :-
       nonvar(List), List = [X|In], 0 =:= X mod P, !,
       ... % remove from constraint store if needed
       filter(In,P,Out) .

% third occurrence
filter(List, _ ,Out,C) :-
       List == [], !,
       ... % remove from constraint store if needed
       Out = [] .

% insert into store in case none of the rules matched
filter(List,P,Out,C) :-
       ... % insert into constraint store
```

If we enable the guard simplification phase, 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 use the `filter/3` constraint 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 declarations to the CHR program:

```
option(type_declaration, filter(list(int),int,list(int))).
option(mode, filter(+,+,?)).
```

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 can detect `filter/3` to be never-stored after the third rule. Also no variable triggering needs to be considered since the relevant arguments are known to be ground. The generated code for the guard simplified program looks like:

```
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(_,_,[]).
```

Benchmark	*Language*	*Guard simpl.*	*Mode decl.*	*Type decl.*	*Clauses*	*Lines*	*Run time (ms)*	*Relative run time*
sum		yes/no	no	no	4	46	1,890	100.0%
(10000,500)	CHR	yes/no	yes	no	3	10	1,680	88.9%
		yes	yes	yes	2	6	1,260	66.7%
	handwritten Prolog code				2	5	1,250	66.1%
Takeuchi		no	no	yes/no	4	50	15,060	100.0%
(1000)	CHR	no	yes	yes/no	3	17	9,910	65.8%
		yes	yes/no	yes/no	2	12	9,190	61.0%
	handwritten Prolog code				2	12	9,190	61.0%
nrev		yes/no	no	no	8	92	4,480	100.0%
(30,50000)	CHR	yes/no	yes	no	6	20	2,820	62.9%
		yes	yes	yes	4	11	1,030	23.0%
	handwritten Prolog code				4	7	920	20.5%
cprimes		no	no	yes/no	14	160	10,730	100.0%
(100000)		no	yes	yes/no	11	42	6,230	58.1%
	CHR	yes	no	no	12	120	10,670	99.4%
		yes	yes	no	10	35	6,140	57.2%
		yes	yes	yes	8	25	5,990	55.8%
	handwritten Prolog code				8	23	5,990	55.8%
dfsearch		no	no	yes/no	5	67	20,290	100.0%
(16,500)		no	yes	yes/no	4	16	17,130	84.4%
	CHR	yes	no	no	5	66	18,410	90.7%
		yes	yes	no	4	15	16,120	79.4%
		yes	yes	yes	3	11	12,080	59.5%
	handwritten Prolog code				3	8	11,330	55.8%

Fig. 1. Benchmark results.

5 Experimental results

In order to get an idea of the efficiency gain obtained by guard simplification, we have measured the performance of several CHR benchmarks, both with and without guard simplification. All benchmarks were performed in hProlog 2.4.5-32 [1], on a Pentium 4 (1.7 GHz) machine running Debian GNU/Linux (kernel version 2.4.25) with a low load.

Figure 1 gives an overview of our results. The first column indicates the benchmark name and the parameters that were used. These benchmarks are available at [9]. The third to fifth column indicates whether the guard simplification phase was enabled, whether mode declarations were provided and whether type declarations were provided – 'yes/no' meaning this has no influence on the resulting compiled code. The sixth and seventh column show the size of the resulting compiled Prolog code, not including auxiliary predicates. The last two columns show the run time

in milliseconds and a percentage comparing the run time to that of the slowest instance for that benchmark. For every benchmark, the results for a hand-written Prolog version are included, representing the ideal target code.

5.1 Individual benchmarks

The first benchmark, `sum`, 500 times creates a list consisting of 10000 times the number 1 and computes the sum of its elements using the simple algorithm from the example in section 2.3:

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

If type and mode declarations are provided, guard (or rather head matching) simplification will move the head matching to the body, enabling never-stored analysis to remove redundant code to add `sum/2` to the constraint store. No performance difference could be measured between the resulting compiled program[2]

```
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:

```
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 the guard simplification phase will remove the guard in the second rule. As a result, the constraint will be detected as being never-stored, also excluding the possibility of variable triggering. In this case, the generated code is identical to the

[2] For readability, variables have been renamed in the generated code shown here.

handwritten Prolog code. But what if X or Y happen to be variables? Could not the transformed second rule fire then, while the original version would not? The answer is no, because trying the first guard would result in a fatal error message (at least in hProlog), both in the original program and in the transformed program. If the host-language for CHR would have a different behavior, say `X =< Y` would fail if X is nonground, the negation of `X =< Y` has to be `(X > Y ; \+ ground(X) ; \+ ground(Y))` instead of just `X > Y`. Now guard simplification would need mode declarations saying X and Y are ground to remove the guard of the second rule. It is trivial to make this kind of modification to the entailment checker if needed.

In the third benchmark, `nrev`, a list of length 30 is reversed 50000 times using the classic naive algorithm. There is a small performance difference between the handwritten Prolog version and the guard simplified version of the CHR program with type and mode declarations. This is caused by the overhead of redundant cuts (`!/0`) in the generated code. Except for these 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 from section 4.3 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 essentially identical (except for some redundant cuts) to the handwritten Prolog code.

5.2 General results

Overall, for these benchmarks, the net effect of the guard simplification transformation – together with never-stored analysis and usage 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.

Compiled to general code (using a non-optimizing compiler), these kind of CHR programs implementing deterministic algorithms have a relatively low performance compared to their native Prolog alternatives. As a result, CHR programmers usually write deterministic predicates in Prolog instead of formulating them as CHR constraints. 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.

Other CHR programs, like typical constraint solvers, where variable triggering occurs and the constraints are typically not never-stored, will not benefit this much from guard simplification. Redundant guards will of course be removed, but in most cases this will not result in a drastic improvement in code size or performance since guards are usually relatively cheap. The main advantage of guard simplification is that relying on it, the CHR programmer is able to write programs that have a more declarative reading and that are more self-documenting. All preconditions needed for a rule to fire can be put in the guard – guard simplification will eliminate all redundant conditions so this will not affect efficiency.

The only difference between the original program and the guard-simplified transformed program is that some conditions (namely those that can be proved to be entailed) are not evaluated in the transformed program. This should only improve efficiency. Thus there are no cases in which guard simplification transforms a program to a less efficient version.

5.3 Compile time added by guard simplification

In most cases, the additional compile time spent in the guard simplification phase is very reasonable. For relatively small CHR programs like the benchmarks discussed above, the time cost of applying guard simplification is more or less insignificant, in the order of 50 milliseconds.

When compiling larger CHR programs, the time complexity of the guard simplification compilation phase depends heavily on the number of earlier subrules to check while building the "no earlier subrule fired"-condition. For a large number of rules per constraint, the amount of rules sharing head constraints also tends to increase. Because of this, the ratio

Program	*Rules*	*Constr.*	*R/C*	*GS (ms)*	*Total (ms)*	*GS/T*
union-find	6	6	1.0	0	40	0.0%
timed automata	23	17	1.4	50	310	16.1%
well-founded semantics	43	18	2.4	70	340	20.6%
finite domain solver	13	6	2.2	80	200	40.0%
CHR compiler	139	73	1.9	1,540	3,270	47.1%
boolean solver	78	8	9.8	420	590	71.2%
(in)finite domain solver	81	9	9.0	950	1,250	76.0%
entailment checker	123	3	41.0	1,830	2,150	85.1%

Fig. 2. Compilation times.

$\frac{\#\text{rules}}{\#\text{constraints}}$ roughly indicates the (relative) amount of time spent simplifying guards, as can be seen in figure 2. The column called "*GS*" gives the amount of time spent in the guard simplification phase, while "*Total*" refers to the total compilation time with all optimizations disabled except guard simplification. The K.U.Leuven CHR compiler source code contains 139 CHR rules and 73 constraints, so on average every constraint occurs in less than two rules. In the compilation, guard simplification takes a reasonable 1.5 seconds on a total compile time of 3.3 seconds (not counting the other optimizations).

In extreme cases where the number of rules per constraint is exceptionally large, the guard simplification phase tends to dominate the compilation time. For example, the entailment checking module described in section 4.2 contains 123 rules and only 3 constraints. In this case, guard simplification takes 1.8 seconds on a total of 2.1 seconds, or about 85%.

6 Conclusion

We have presented a compiler analysis called guard simplification that allows CHR programmers to write more declarative CHR programs that are more self-documented. Indeed, all preconditions for rule application can now be included in the guard, without efficiency loss. Earlier work introduced mode declarations used for hash tabling and other optimizations. In addition, we have provided a way for CHR programmers to add type declarations to their programs. Using both mode and type declarations we have realized further optimization of the generated code.

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. Now they no longer need to write mixed-language programs for efficiency: they can simply write the entire program in CHR, because thanks to guard simplification and other analyses like storage analysis, the K.U.Leuven CHR compiler is able to generate efficient code with the constraint store related overhead reduced to a minimum. While guard simplification in itself does not reduce this overhead (although it does remove the overhead of checking entailed guard conditions), it enables other analyses to do so. 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 where this is needed. These non-declarative auxiliary constraints (like `getFoo/1`) can be avoided entirely by writing these parts in CHR, using the expressiveness of multi-headed rules.

6.1 Related work

The guard simplification analysis is somewhat similar to switch detection in Mercury [6]. In Mercury, disjunctions in predicate bodies – explicit or implicit, i.e. 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. Such disjunctions are called *switches*, since they superficially resemble switches in the C programming language.

Switch detection only 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 because CHR rules are committed-choice. In Mercury, removing the last test in a disjunction would change the behavior of the clause because on backtracking, this branch would then always be entered. Besides, often unifications introduce new variables that are used in the rest of the clause, also preventing removal.

Head matching simplification vaguely reminds of indexing in Prolog compilers; the former working on multiple rules sharing head constraints, the latter on multiple clauses for the same predicate. The similarity is rather superficial and it seems to us that work done on the topic of Prolog indexing is not very relevant in the context of guard simplification in CHR.

6.2 Future work

When there are many earlier subrules to consider in the guard simplification analysis, the performance of our current implementation may become an issue. In future work we hope to improve the scalability of our implementation, although this does not present an immediate problem.

Our current guard entailment knowledge base has only limited knowledge about builtins and their negation. To be able to recognize more redundant guards, we intend to not only extend the knowledge base, but also investigate the following two approaches. Firstly, we could have the user declare additional facts with his program. These facts would be added to the knowledge base during the program analysis. Secondly, by analyzing the implementation of user-defined predicates used in guards, the necessary facts for the knowledge base could be inferred automatically.

The information entailed by the failure and success of guards, used here to eliminate redundant guards, seems also useful in other program analyses and transformations. One application would be program specialization: the code for executing a constraint is specialized for a particular call in the body of a CHR rule. All the information regarding the necessary success and failure leading up to this call, may serve as initial information to perform guard simplification. This may lead to the elimination of more redundant guards (and even redundant rules) for the specialized case.

Finally we would like to integrate the analyses presented in this paper into the bootstrapped CHR compiler which is currently being implemented by Christian Holzbaur et al.

References

1. Bart Demoen. The hProlog home page, October 2004. http://www.cs.kuleuven.ac.be/~bmd/hProlog.
2. Gregory J. Duck, Tom Schrijvers, and Peter J. Stuckey. Abstract Interpretation for Constraint Handling Rules. Technical Report CW 391, K.U.Leuven, Departement of Computer Science, 2004.
3. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. Extending Arbitrary Solvers with Constraint Handling Rules. In D. Miller, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 79–90, Uppsala, Sweden, August 2003. ACM Press.
4. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of constraint handling rules. In *Proceedings of the 20th International Conference on Logic Programming (ICLP'04)*, Saint-Malo, France, September 2004. Springer LNCS.
5. T. 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. 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.
7. Christian Holzbaur and Thom Frühwirth. Compiling Constraint Handling Rules into Prolog with Attributed Variables. In G. Nadathur, editor, *Proceedings of the First International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, Paris, France, September/October 1999. Springer LNCS.
8. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, number 2004-01, 2004.
9. Tom Schrijvers. CHR benchmarks and programs, October 2004. Available at http://www.cs.kuleuven.ac.be/~toms/Research/CHR/.
10. Tom Schrijvers and Bart Demoen. Antimonotony-based Delay Avoidance for CHR. Technical Report CW 385, K.U.Leuven, Department of Computer Science, July 2004.
11. Tom Schrijvers and Thom Früwirth. Implementing and Analysing Union-Find in CHR. Technical Report CW 389, K.U.Leuven, Department of Computer Science, July 2004.
12. 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, February 1995.
13. Various. 40 CHR Constraint Solvers Online, November 2004. Available at *http://www.pms.informatik.uni-muenchen.de/~webchr/*.