Guard Reasoning for CHR Optimization

Jon Sneyers
Tom Schrijvers
Bart Demoen

Report CW 411, May 2005



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

Guard Reasoning for CHR Optimization

Jon Sneyers
Tom Schrijvers
Bart Demoen

Report CW 411, May 2005

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 potential misbehavior under the theoretical operational semantics. We introduce two source to source transformations: *guard simplification* and *occurrence subsumption*. By removing redundant guards and eliminating redundant generated code for subsumed occurrences, they improve performance and allow CHR programmers to write self-documented rules with a clear logical reading. Formal correctness proofs are given, 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 Reasoning for CHR Optimization

Jon Sneyers
jon@cs.kuleuven.ac.be

Tom Schrijvers*
toms@cs.kuleuven.ac.be

Bart Demoen
bmd@cs.kuleuven.ac.be

K.U.Leuven, Department of Computer Science
Celestijnenlaan 200A
3001 Heverlee, Belgium

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 potential misbehavior under the theoretical operational semantics. We introduce two source to source transformations: *guard simplification* and *occurrence subsumption*. By removing redundant guards and eliminating redundant generated code for subsumed occurrences, they improve performance and allow CHR programmers to write self-documented rules with a clear logical reading. Formal correctness proofs are given, 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 assume that the reader is familiar with the syntax and semantics of CHR, referring to [5] for an overview. Although examples are given in Prolog, the optimizations can be applied in any context.

The (original) theoretical operational semantics (ω_t) of CHRs, as defined in [5], is nondeterministic as the order in which rules are tried is not specified. However, all implementations we know of use a more specific operational semantics, called the *refined* operational semantics (ω_r) [4]. In ω_r, the rules are tried in textual order. Usually, CHR programmers take this refined operational semantics into account. As a result, their 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 of more declarative code with a clear logical reading. Then again, some programming idioms are harder to implement and the compiled code can be 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 must 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 such redundant preconditions improves the efficiency of the compiled code, but it also makes the program less self-documented.

To improve the efficiency of the generated code, compiler directives called *pragmas* can be used to declare that a certain constraint occurrence does not need to be compiled. Syntactically, the CHR programmer marks such occurrences with a `#ID` identifier and adds "`pragma passive(ID)`" at the end of the rule. These pragmas also make CHR programs less readable and less declarative because of the combination, in one syntactical construction, of both declarative rules and purely operational execution strategy directives.

In this paper, we propose two compiler optimizations that are a major step towards allowing CHR programmers to write more readable and declarative programs while obtaining the same efficiency as programs written with the specifics of the refined operational semantics in mind.

The first 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 ω_r semantics. The transformed program is simpler, possibly allowing more optimizations from other analyses. For example, guard simplification can reveal the never-stored property [2], as we will show later.

The second, called *Occurrence Subsumption*, automatically detects occurrences that do not need to be compiled. Because of this, `pragma passive` directives no longer have to be provided manually. Thanks to both optimizations, CHR programmers can focus on writing a declarative specification and rely on the compiler to produce efficient code.

This paper is partially based on [17, 16], in which guard simplification is introduced. It is structured as follows: first we summarize the definition of CHR and the refined operational semantics, described in detail in [4]. Section 3 presents an intuitive overview of guard (and head matching) simplification, occurrence subsumption, and the use of type and mode information to enhance them. In section 4, formal definitions of the optimizations are given, followed by correctness proofs. Section 5 deals with the implementation of the optimizations in the K.U.Leuven CHR compiler [14]. Then, in section 6, the results of several benchmarks are discussed. Finally, section 7 concludes this paper, summarizing our contributions and discussing related and future work.

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

2. CONSTRAINT HANDLING RULES

We use $[H|T]$ to denote the first (H) and remaining elements (T) of a sequence, $+\!\!+$ for sequence concatenation, ϵ for empty sequences, $\uplus$ for multiset union, and $\subseteq$ for multiset subset. We shall sometimes omit existential quantors to get a lighter notation. 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 assume this underlying solver supports at least equality, `true` and `fail`. We consider all three types of CHR rules to be special cases of simpagation rules:

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

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

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 $H_i^k +\!\!+ H_i^r$.

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. 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] and an illustrating example can be found in section 2.1 of [13].

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

The Refined Operational Semantics ω_r

An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with some unique integer i, used to differentiate among copies of the same constraint. We introduce functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences of identified CHR constraints in the obvious manner. An *occurrenced* identified CHR constraint $c\#i : j$ indicates that only matches with occurrence j of constraint c should be considered when the constraint is active.

Formally, the execution state of ω_r semantics is a tuple $\langle A, S, B, T\rangle_n$ where A, S, B, T and n, represent the execution stack, the CHR store, the builtin store, the propagation history and the next free identity number respectively.

The *execution stack* A is a sequence of constraints, identified CHR constraints and occurrenced identified CHR constraints, with a strict ordering in which only the top-most constraint is active. The *CHR constraint store* S is the multiset of *identified* CHR constraints that can be matched with rules in the program P. The *builtin constraint store* B contains any builtin constraint that has been passed to the underlying solver. We model it as an abstract logical conjunction of constraints. The *propagation history* T is a set of sequences, each recording the identities of the CHR constraints which fired a rule, and the rule itself. This is necessary to prevent trivial non-termination for propagation rules. Finally, the counter n represents the next free integer which can be used to identify a CHR constraint.

1. **Solve** $\langle [c|A], S' \uplus S, B, T\rangle_n \rightarrowtail \langle S +\!\!+ A, S' \uplus S, c \wedge B, T\rangle_n$
   if c is a builtin constraint and B fixes the variables of S'.
2. **Activate** $\langle [c|A], S, B, T\rangle_n \rightarrowtail \langle [c\#n : 1|A], S', B, T\rangle_{(n+1)}$
   if c is a CHR constraint, where $S' = \{c\#n\} \uplus S$.
3. **Reactivate** $\langle [c\#i|A], S, B, T\rangle_n \rightarrowtail \langle [c\#i : 1|A], S, B, T\rangle_n$
4. **Drop** $\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail \langle A, S, B, T\rangle_n$
   if there is no j^{th} occurrence of c in P.
5. **Simplify** $\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail \langle C +\!\!+ A, H_1 \uplus S, \theta \wedge B, T \cup \{h\}\rangle_n$
   if the jth occurrence of the constraint c is d_j in a rule R in P of the form $H_1' \setminus H_2', d_j, H_3' \iff g \mid C$ and $\exists\theta : c = \theta(d_j)$, $chr(H_k) = \theta(H_k')$ $(k = 1, 2, 3)$, $\mathcal{D} \models B \rightarrow (\theta \wedge g)$, and $T \not\ni h = (id(H_1 +\!\!+ H_2 +\!\!+ c\#i +\!\!+ H_3), R)$.
6. **Propagate** $\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail \langle C +\!\!+ [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus S, \theta \wedge B, T \cup \{h\}\rangle_n$
   if the jth occurrence of the constraint c is d_j in a rule R in P of the form $H_1', d_j, H_2' \setminus H_3' \iff g \mid C$ and $\exists\theta : c = \theta(d_j)$, $chr(H_k) = \theta(H_k')$ $(k = 1, 2, 3)$, $\mathcal{D} \models B \rightarrow (\theta \wedge g)$, and $T \not\ni h = id(H_1 +\!\!+ c\#i +\!\!+ H_2 +\!\!+ H_3), R)$.
7. **Default** $\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail \langle [c'|A], S, B, T\rangle_n$
   if no other transition applies, where $c' = c\#i : (j + 1)$.

Figure 1: The transitions defining ω_r execution.

Given an initial goal G, the *initial state* is: $\langle G, \emptyset, true, \emptyset\rangle_1$. Execution proceeds by exhaustively applying transitions from figure 1 to the initial execution state until the builtin store is unsatisfiable or no transitions are applicable.

3. OVERVIEW

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

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

EXAMPLE 1 (GCD).

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

A query containing two (or more) `gcd/1` constraints with integer arguments, will eventually result in a constraint store containing one `gcd(`k`)` constraint where k is their greatest common divisor. For example, the query `gcd(9),gcd(15)` causes the second rule to fire, resulting in `gcd(9),gcd(6)`. This rule keeps firing until the store contains `gcd(3),gcd(0)`. Now the first rule fires, removing `gcd(0)` from the store. The remaining constraint does indeed contain the greatest common divisor of 9 and 12, 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 if the second rule is tried, the guard of the first rule must have failed – otherwise the active constraint would have been removed. Hence the condition `N =\= 0` is redundant Under ω_t semantics, this second version of the CHR program is no longer guaranteed to terminate, since applying the second rule indefinitely (which is a valid execution strategy under ω_t semantics) when the constraint store contains e.g. `gcd(3),gcd(0)` results in an infinite loop.

3.1 Guard simplification

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

EXAMPLE 2 (ENTAILED GUARD).

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

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

EXAMPLE 3 (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 a contradiction, so we know the third rule can never fire. □

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

3.2 Head matching simplification

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

EXAMPLE 4 (HEAD MATCHING SIMPLIFICATION).

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

We can rewrite the second rule to `p(X,Y) <=> ...`, because the (implicit) condition `X == Y` is entailed by the negation of the guard of the first rule. In the refined operational semantics, this does not change the behavior of the program. 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 detects `p/2` to be never-stored, and more efficient code can be generated. □

3.3 Type and mode declarations

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

EXAMPLE 5 (SUM).

```
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 [15], optional mode declarations were introduced to specify the mode – ground (+) or unknown (?) – of constraint arguments. Inspired by the Mercury type system [18], 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:

```
:- 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 some type `T`, a variable that can be instantiated with builtin types like `int`, `float`, the general type `any`, or any user-defined type. The constraint declaration on the second line includes mode and type information. It is read as follows: `sum/2` is a CHR constraint which has two arguments: a ground list of integers and an integer, which can be ground or a variable. Using this knowledge, we can rewrite the second rule of the example program to "`sum(A,S) <=> A = [X|Xs], sum(Xs,S2), S is X + S2.`", keeping its behavior intact while again helping never-stored analysis.

3.4 Occurrence Subsumption

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

The K.U.Leuven CHR compiler already had two optimizations that can be considered to be special cases of occurrence subsumption. *Symmetry analysis* checks rules R with a head containing two constraints c_1 and c_2 that are symmetric, in the sense that there is a variable renaming θ such that $\theta(c_1) = c_2$ and $\theta(c_2) = c_1$ and $\theta(R) = R$. In such rules, one of the c_i's is made `passive`. The other optimization looks at rules which make a constraint have *set semantics*, of the form $c_1 \backslash c_2 \Leftrightarrow true | B$, without head matchings, where

c_1 and c_2 are identical modulo functional dependencies. In this case, c_1 can be made `passive` (or c_2, but it is better to keep the occurrence which immediately removes the active constraint). A more efficient constraint store can be used for c if it has set semantics.

In section 4.1 of [9, 10], a concept called *continuation optimization* is introduced. *Fail* continuation optimization is essentially the same as occurrence subsumption, while *success* continuation optimization uses similar reasoning to improve the generated code for occurrences in propagation rules. The HAL CHR compiler, discussed in that paper, performs a very simple fail continuation optimization, which only considers rules without guards and does not use information derived from the failing of earlier guards.

EXAMPLE 6 (SIMPLE CASE).

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

Suppose the active constraint is `c(X,Y)`. If the first occurrence does not fire, this means that either `c(Y,X)` is not in the constraint store, or `p(X),p(Y)` fails. If the second occurrence fires, then `c(Y,X)` must be in the constraint store, and the guard `p(Y),p(X)` must succeed. So it is impossible for the second occurrence to fire if the first one did not. It is not even tried if the first occurrence did fire, removing the active constraint. Hence, compiling the second occurrence is redundant and we can change the rule to `c(A,B), c(B,A)#Id <=> p(A),p(B) | true pragma passive(Id).` □

In the following (more or less artificial) examples, current analyses only detect a small subset of the redundant occurrences detected by our new analysis. Underlined occurrences can be made `passive` so they do not need to be compiled. All these redundant occurrences are detected by the current K.U.Leuven CHR compiler.

EXAMPLE 7 (MORE COMPLICATED CASES).

1. `in(X,A:B), in(X,C:D) <=> A < B, C < D |` Body.
2. `a(X,Y,Z), a(Y,Z,X), a(Z,X,Y) <=>` Body.
3. `b(X,Y,Z), b(Y,Z,X), b(Z,X,Y) <=>`
   `(p(X); p(Y)) |` Body.
4. `c(A,B,C), c(A,C,B), c(B,A,C),c(B,C,A),`
   `c(C,A,B),c(C,B,A) <=>` Body.
5. `d(A,B,C), d(A,C,B), d(B,A,C),d(B,C,A),`
   `d(C,A,B),d(C,B,A) <=> p(A),p(B) |` Body.
6. `e(A,B,C), e(A,C,B), e(B,A,C),e(B,C,A),`
   `e(C,A,B),e(C,B,A) <=> p(A) |` Body.
7. `f(A,B), f(B,C) <=> A \== C |` Body.
   `f(A,B), f(B,C) <=>` Body. □

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

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

EXAMPLE 8 (TOO COMPLICATED CASE).

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

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

4. FORMAL DESCRIPTION AND PROOFS

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

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

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

DEFINITION 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* $\text{functor}(H_j) \subseteq \text{functor}(H_i)$.

Now we can define a logical expression $\mathsf{nesr}(R_i)$ ("**n**o **e**arlier **s**ub**r**ule (fired)") stating the implications of the fact that all constraint-removing earlier subrules of rule R_i have been tried unsuccessfully.

DEFINITION 4 (NESR). *For every rule* R_i*, we define:*

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

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

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

4.1 Definition of Guard Simplification

Consider a CHR program P with rules R_i which have guards $g_i = \bigwedge_k g_{i,k}$. Applying guard simplification to this program means rewriting some parts of the guards to `true`, if they are entailed by the "no earlier subrule fired" condition (and already evaluated parts of the guard). The entire

guard is rewritten to `fail`, if the *negation* of some part of it is entailed by that condition. This effectively removes the rule. Because head matchings are made explicit, head matching simplification (section 3.2) is an implicit part of guard simplification.

DEFINITION 5 (GUARD SIMPLIFICATION). *Applying the* guard simplification *transformation to a CHR program P (with rules $R_i = H_i \Leftrightarrow \bigwedge_k g_{i,k} | B_i$) results in a new CHR program $P' = GS(P)$ which is identical to P except for the guards, i.e. its rules R'_i are of the form $H_i \Leftrightarrow g'_i | B_i$, where*

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

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

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

Note that this definition is slightly stronger compared to the definition given in [17, 16], because it takes into account the left-to-right evaluation of the guard. As a result, internally inconsistent guards like `X > Y, Y > X` can be simplified to `fail`, and internally redundant guards can be simplified, e.g. the condition `X >= Y` can be removed from `X > Y, X >= Y`.

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

4.2 Definition of Occurrence Subsumption

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

DEFINITION 6 (NEOCC). *Given a non-propagation rule R_i containing in its head multiple occurrences $c_m, \ldots, c_n$ of the same constraint c and other partner constraints d. We define for every c_k $(m \leq k \leq n)$:*

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

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

As the reader can verify, $\mathsf{fc}(R_i, c_l)$ is the firing condition for rule R_i to fire if c_l is the active constraint. The condition $\mathsf{neocc}(R_i, c_k)$ ("**n**o **e**arlier **occ**urrence (fired)") describes that if the kth occurrence of c is tried, i.e. application of rule R_i is tried, the earlier occurrences inside rule R_i must have failed (since R_i is not a propagation rule). Now we can define formally which occurrences can be made passive.

DEFINITION 7 (OCCURRENCE SUBSUMPTION). *Given a rule R_i as in the previous definition. We say c_k $(m < k \leq n)$ is* occurrence subsumable *iff*

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

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

4.3 Correctness proofs

Detailed definitions of execution state, transition and derivation can be found in [4]. The summary from section 2 should suffice to understand the theorems and proofs below.

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

LEMMA 1. *If in a derivation $s_0 \rightarrowtail^* s_k$ 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

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

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:

  (a) All constraints from the head of R_x are in the CHR store. This means eventually, either rule R_x will be tried with d as the active constraint, or another partner constraint gets triggered (but not c, because of how we defined $s_{l'}$), in turn maybe triggering other partner constraints, but any way R_x will be tried with one of the partner constraints as the active constraint. Because the builtin store now does entail g_x, the rule fires and a tuple is added to the propagation history. In execution state s_k, this tuple will still be in the propagation history, preventing the application of rule R_x. This is of course a contradiction.

  (b) Not all constraints from the head of R_x are in the CHR store, so some have to be added before s_k is reached, and a similar early-firing happens at the moment the last partner constraint is added, also leading to a contradiction.
- If none of the constraints from the head of R_x are triggered, some of them are not in the CHR store yet, because if they are all there, at least one of them should be triggered, otherwise the change in the builtin store

would not affect the entailment of g_x. As a result, some of the constraints occurring in the head of R_x have to be added before s_k is reached so we get a similar early-firing situation as above, again leading to a contradiction.

All these cases lead to a contradiction, so our assumption was wrong. This shows that during the derivation $s_{l'} \rightarrowtail^* s_k$, 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 that derivation the last of these partner constraints (d) is added to the CHR store, so all constraints needed for R_x are in the CHR store. However, the only transition that could have added d to the CHR store is **Activate**, which also makes d the active constraint. We get an early-firing situation like above because the guard of R_x is entailed and every partner constraint (including c) is now in the CHR store. So we get a contradiction, proving that during the derivation $s_{l'} \rightarrowtail^* s_k$, 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 now show that the "no earlier subrule fired" formula $\mathsf{nesr}(R_i)$ 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 $\mathcal{D} \models B \rightarrow \mathsf{nesr}(R_{c,j})$.*

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

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

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

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 can fire, applying rule R_a, because:

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

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

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

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

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

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

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

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

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

Now we can show that subsumable occurrences do not need to be compiled (can be made passive). More formally:

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

PROOF. Suppose the **Simplify** or **Propagate** transition can be applied, firing rule R. Using the notation from definition 6, this means that $\mathcal{D} \models B \rightarrow \mathsf{fc}(R, c_j)$ Also, lemma 2 tells us that $\mathcal{D} \models B \rightarrow \mathsf{nesr}(R)$. Because of lemma 1 we know that rule R has been tried for the earlier occurrences of c in that rule. These tries must have failed, because R is a constraint-removing rule (c_j is occurrence subsumable) which cannot be applied twice on the same constraints. So

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

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

5. IMPLEMENTATION

We have implemented the presented analyses and transformations in the K.U.Leuven CHR compiler [14], which can be found in recent releases of SWI-Prolog [19]. We wrote a separate auxiliary module for checking entailment. This section starts with an overview of our implementation of guard simplification and occurrence subsumption, which depends heavily on the entailment checker. In 5.2, the implementation of the entailment check module is discussed. Finally we take a look at the code the compiler generates for an example CHR program, and how this code can be improved by guard simplification.

5.1 Overview

The guard simplification / occurrence subsumption compilation phase rewrites every rule in the CHR program. In the rewritten rules, the redundant parts of the guard have been removed, the head matchings (an implicit part of the guard) are made as general as possible and subsumed occurrences are declared to be passive. As a result, the generated code is more efficient because redundant checks are removed, and also the next compilation phases – like storage analysis – are 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. rewrite the rule `c([X|Xs],Y,Y) <=> G | B`
   to an equivalent head normalized rule:
   `c(A,Y,B) <=> A = [X|Xs], B == Y, G | B`.

2. iteratively construct a conjunction similar to $\mathsf{nesr}(R_i)$ from section 4, containing the negations of the guards of the earlier subrules $R_j \prec R_i$, consider all possible substitutions;

   E.g. for the program:
   `r1@ c(X) <=> p(X) |` B_1
   `r2@ c(2) <=> q |` B_2
   `r3@ c(A), c(B) <=>` G_3 `|` B_3

   the following conjunction is computed for the third rule:
   `(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 == []; ...))).`

   To prevent such infinite loops, we stop at the highest nesting depth occurring in R_i and its earlier subrules. E.g for `sum/2` from example 5, the type condition would be

   `(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 4): check if it is entailed by the derived information and remove it if it is (i.e. replace it with `true`); if its negation is entailed, replace it with `fail`;

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 the simplification rule

   `c([X|Xs],[],A,A,[B|Bs]) <=> B>0 | d(X,A)`

   to `c(Z,_,A,_,[B|_]) <=> B > 0 | Z=[X|_], d(X,A)`

   if the derived information entails that the first arguments of `c/5` is a non-empty list, the second argument is an empty list and the third and fourth argument are identical.

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;

7. for every occurrence c_k of a constraint that occurs more than once in R_i, compute $\mathsf{neocc}(R_i, c_k)$ and do occurrence subsumption by checking whether $\neg\mathsf{fc}(R_i, c_k)$ is entailed by $\mathsf{nesr}(R_i) \wedge \mathsf{neocc}(R_i, c_k)$, i.e. check whether occurrence c_k can be safely set to '`passive`'.

   E.g. for the program:

   `r1@ c(X,Y), c(Y,Z) <=> X \== Z |` B_1
   `r2@ c(A,B), c(B,C) <=>` B_2

   the last occurrence of `c/2` can be made `passive` (does not need to be compiled) since nesr(`r2`) entails `A == C`, which causes the heads to be symmetric. Because of this symmetry, the last occurrence of `c/2` clearly is occurrence subsumable.

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

5.2 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 condition A (e.g. $X < Y \wedge Y < Z$), i.e. $A \rightarrow B$. Since in general this problem is undecidable, the entailment checker tries to prove that B

[1]Some examples: `X < Y` → `X >= Y`, `X == Y` → `X \== Y`, `true` → `fail`, `var(X)` → `nonvar(X)`, `\+ Cond` → `Cond`, `(A; B)` → `(NA, NB)` where `A` → `NA` and `B` → `NB`.

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 =< 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.

5.3 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 [6]:

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

The CHR compiler (without guard simplification) generates general code for the `filter/3` constraint. Because no information is known about the arguments of `filter/3`, the compiled code has to take into account variable triggering and the possibility that none of the rules apply and the constraint has to be stored. Following the compilation scheme explained in [11], 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, !,
       ... % removecode
       Out = [E|Out1], filter(In,P,Out1).

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

% third occurrence
filter(List, _ ,Out,C) :-
       List == [], !,
       ... % removecode
       Out = [].

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

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

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

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

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

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

This transformed program is compiled to more efficient Prolog-code, because never-stored analysis detects `filter/3` to be never-stored after the third rule. 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(_,_,[]).
```

6. EXPERIMENTAL RESULTS

In order to get an idea of the efficiency gain obtained by guard simplification and occurrence subsumption, we have measured the performance of several CHR benchmarks, both with and without the optimization. All benchmarks were performed in SWI-Prolog [19] version 5.5.2, on a Pentium 4 (1.7 GHz) machine running Debian GNU/Linux (kernel version 2.4.25) with a low load.

6.1 Guard simplification results

Figure 2 gives an overview of our results. The first column indicates the benchmark name and the parameters that were used. These benchmarks are available at [12]. The second and third column indicate whether mode declarations were provided and whether the guard simplification phase was enabled – an empty cell meaning this has no influence on the resulting compiled code (so it can be "yes" or "no"). If the third column shows "type", it means guard simplification was enabled and additional type information was provided. The fourth column shows the size of the resulting compiled Prolog code as a (*#Clauses* ; *#Lines*) pair, not including auxiliary predicates. The last column shows the runtime in seconds and a percentage comparing the runtime to that of the version with mode information but without guard simplification. If a cell contains an equality sign ("="), we could not measure any performance difference compared to the version in the row just above that cell. If a cell contains an equivalence sign ("≡"), the Prolog code for that row is identical to the one in the row just above. For every benchmark, the results for a hand-written Prolog version are included, representing the ideal target code.

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

Individual benchmarks

The first benchmark, `sum`, computes the sum of the elements of a list of 10000 numbers (all 1) 500 times, using the simple algorithm from example 5:

```
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 can move the head matching to the body, enabling never-stored analysis to remove redundant code to add `sum/2` to the constraint store. As in the other benchmarks, no significant performance difference could be measured between the resulting compiled program[2]

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

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

Figure 2: Benchmark results (guard simplification).

```
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, guard simplification removes the guard in the second rule. As a result, the constraint is detected as being never-stored, also excluding the possibility of variable triggering. In this case, the generated code is identical to the handwritten Prolog code. The guard `X > Y` is removed because it is (entailed by) the negation of `X =< Y`. When `X =< Y` fails, we know `X` and `Y` are ground terms evaluating to numbers, and `X > Y`. If in some other host language, `X =< Y` would fail if its arguments are invalid – instead of resulting in some fatal error message or exception – then it would have a different negation, for instance `(X > Y ; \+ number(X) ; \+ number(Y))`. In that case, guard simplification would not remove the guard of the second rule, except when mode and type information is given.

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

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

is essentially identical to the handwritten Prolog program:

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

The example in section 5.3 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 again essentially identical (except for some redundant cuts) to the handwritten Prolog code.

Conclusion

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

Naively compiled to general code, these CHR programs that implement 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. 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 such 'host-language interface' constraints like `getFoo/1` is drastically reduced.

Other CHR programs, like typical constraint solvers, where variable triggering occurs and the constraints are typically not never-stored, do not benefit this much from guard simplification. Redundant guards are of course removed, but in most cases this does 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 eliminates all redundant conditions so this does not affect efficiency.

6.2 Occurrence subsumption results

In figure 3, the performance of the generated code for four benchmarks is given, both with and without the occurrence subsumption optimization. The symmetry and set semantics dependency analyses were disabled in both cases because they are special cases of occurrence subsumption.

Benchmark	*OS*	*Prog. size*	*Runtime (%)*	
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)
zebra	no	11 ; 198	11.5	(100)
(10)	yes	10 ; 168	10.3	(90)

Figure 3: Benchmark results (occ. subsumption).

The second column indicates whether occurrence subsumption was enabled, the other columns are as in figure 2.

These benchmarks are typical constraint solvers, having constraints that need to be stored. They do not implement a simple deterministic algorithm. It is much harder to hand-compile them to clean Prolog, so we made no attempt to compare the generated code to hand-written target code.

Source files for the benchmarks are available at [12]. The `bool_chain` benchmark has the same CHR rules as the classic `bool` benchmark. Instead of the boolean adder (in which no constraint needs to be stored at any given time), a chain of boolean `and`-constraints is constructed and partially solved.

Occurrence subsumption seems to result in a small to medium performance improvement, if there are subsumable occurrences (which is of course true for these benchmarks). Artificial examples can of course be constructed, for which a more spectacular performance gain is obtained, e.g. using rules like the one defining `c/3` in example 7. Occurrence subsumption also reduces the size of the generated code, by eliminating entire clauses. Compared to guard simplification, this size reduction is more visible, unless of course – as in the case of benchmarks from the previous section – guard simplification reveals the never-stored property, also allowing major simplification of the generated code.

6.3 Performance of our implementation

In most cases, the additional compile time spent in the guard simplification and occurrence subsumption phase (abbreviated to G&O from now on) is acceptable. For relatively small CHR programs like the benchmarks discussed above, the time cost of applying G&O is more or less insignificant, in the order of 50 milliseconds.

In general, the time complexity of the G&O 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 $\frac{\#\text{rules}}{\#\text{constraints}}$ roughly indicates the (relative) amount of time spent simplifying guards, as can be seen in figure 4. The number of rules, number of constraints and their ratio is shown in columns "R", "C" and "R/C", respectively. The column labeled "G&O" gives the amount of time (in seconds) spent in the G&O optimizations, while "T" refers to the total compilation time with all optimizations disabled except G&O. These times are given for both the hProlog version and the SWI-Prolog version of the K.U.Leuven CHR compiler.

In both cases, the relative performance of guard simplification worsens as the ratio $\frac{\#\text{rules}}{\#\text{constraints}}$ increases. However,

				hProlog			*SWI-Prolog*		
Program	*R*	*C*	*R/C*	G&O	T	%	G&O	T	%
unionfind	6	6	1.0	.00	.04	0	.04	.08	48
timed aut	23	17	1.4	.05	.31	16	.34	.69	49
w-f sem	43	18	2.4	.07	.34	21	.46	.92	50
finite dom	13	6	2.2	.08	.20	40	.68	.83	82
chr comp	139	73	1.9	1.5	3.3	47	11.3	13.3	85
bool solv	78	8	9.8	.42	.59	71	2.7	3.0	92
infin dom	81	9	9.0	1.0	1.3	76	13.5	13.9	97
entailchk	123	3	41.0	1.8	2.2	85	47.6	48.0	99

Figure 4: Compilation times.

there is a significant performance difference between the two Prolog implementations. Most likely, this is due to the less efficient implementation of disjunctions and multiple clauses in SWI-Prolog. Hence, the generated code for the entailment check module, which contains many clauses for the many occurrences of `known/1` constraints, is compiled to significantly less efficient WAM code in SWI-Prolog.

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 hProlog case, G&O takes a reasonable 1.5 seconds on a total compile time of 3.3 seconds. In extreme cases where the number of rules per constraint is exceptionally large, the G&O phase tends to dominate the compilation time. For example, the entailment checking module described in section 5.2 contains 123 rules and only 3 constraints. Again in hProlog, G&O takes 1.8 seconds on a total of 2.2 seconds, or about 85%.

7. 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. They no longer need to write mixed-language programs for efficiency: they can simply write the entire program in CHR. Non-declarative auxiliary "host-language interface" constraints like `getFoo/1` (see section 6.1) can be avoided. Thanks to guard simplification, occurrence subsumption and other analyses like storage analysis, the K.U.Leuven CHR compiler is able to generate efficient code with minimal constraint store related overhead. While guard simplification by itself does not reduce this overhead (although it does remove the overhead of checking entailed guards), it helps other analyses to do so.

7.1 Related work

As mentioned before, occurrence subsumption is essentially the same as *fail continuation optimization* [9, 10], although our implementation performs much more complex implication reasoning, resulting in a stronger optimization.

Guard simplification is somewhat similar to *switch detection* in Mercury [7]. In Mercury, disjunctions – explicit or implicit (multiple clauses) – are examined for determinism analysis. In general, disjunctions cause a predicate to have multiple solutions. However, if for any given combination of input values, only one of the disjuncts can succeed, the disjunction does not affect determinism. Because they superficially resemble switches in the C programming language, such disjunctions are called *switches*.

Switch detection checks unifications involving variables that are bound on entry to the disjunction and occurring in the different branches. In a sense, this is a special case of guard simplification, since guard simplification considers other tests as well, using a more general entailment checking mechanism. Guard simplification analysis can be used to remove redundant guard conditions on the source level, because CHR rules are committed-choice. It is harder to express the switch detection optimization as a source to source transformation for Mercury programs.

Head matching simplification is similar to *indexing* in Prolog compilers [8]; the former working on multiple rules sharing head constraints, the latter on multiple clauses for the same predicate. The similarity is rather superficial but it seems that Prolog indexing is relevant in the context of CHR, and needs further exploration.

7.2 Future work

Guard simplification and occurrence subsumption can be combined into one analysis. In some intermediate representation, there can be a separate copy of each rule for every constraint occurrence c, where all heads except c are passive. This representation is closer to the generated Prolog code, where each occurrence gets a separate clause in which (after matching the partner constraints) the rule guard and body are duplicated. From this angle, guard simplification is simplifying the guards of all copies of a certain rule at once, while occurrence subsumption is simplifying the guard of one specific copy to `fail`, removing that copy. A stronger and more general optimization can be obtained by simplifying the guard of each copy separately. This optimization can no longer be expressed as a source to source transformation, except by explicitly constructing the rule copies, which might lead to an undesirable code size explosion.

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.

Another interesting concept is *success continuation optimization* [10], which – as far as we know – has not been implemented in any CHR compiler so far. This appears to be a promising technique that may be implemented in future versions of our CHR compiler.

When there are many earlier subrules to consider in the guard simplification analysis, the performance of our current implementation may become an issue, especially in SWI-Prolog. Rules with many head constraints that share a lot of these constraints are an even bigger performance issue,

because of the combinatorial blowup caused by constructing all possible mappings from the head of an earlier subrule to the current rule head. For example, if some constraint c occurs n times in the head of an earlier subrule, and m ($\geq n$) times in the current head, there are $\frac{m!}{(m-n)!}$ conditions to be added to the nesr conjunction. In future work we hope to improve the scalability of our implementation, although it does not present an immediate problem.

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.

8. REFERENCES

[1] Bart Demoen. The hProlog home page, January 2005. 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] 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] Thom Frühwirth et al. 40 CHR Constraint Solvers Online, January 2005. *http://www.pms.informatik.uni-muenchen.de/~webchr/*.

[7] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the 19th Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.

[8] Timothy Hickey and Shyam Mudambi. Global compilation of Prolog. *Journal of Logic Programming*, 7:193–230, 1989.

[9] Christian Holzbaur, María García de la Banda, David Jeffery, and Peter J. Stuckey. Optimizing compilation of Constraint Handling Rules. In P. Codognet, editor, *Proceedings of the 17th International Conference on Logic Programming (ICLP)*, pages 74–89. Springer LNCS, 2001.

[10] Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. In *Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules, to appear*, 2005.

[11] 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.

[12] Tom Schrijvers. CHR benchmarks and programs, October 2004. Available at http://www.cs.kuleuven.ac.be/~toms/Research/CHR/.

[13] Tom Schrijvers and Bart Demoen. Antimonotony-based Delay Avoidance for CHR. Technical Report CW 385, K.U.Leuven, Department of Computer Science, July 2004.

[14] Tom Schrijvers and Bart 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.

[15] 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.

[16] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Simplification in CHR programs. Technical Report CW 396, K.U.Leuven, Department of Computer Science, November 2004.

[17] 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. to appear.

[18] 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.

[19] Jan Wielemaker. The SWI-Prolog home page, January 2005. http://www.swi-prolog.org.