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, obfuscating their logical reading and causing different (termination) behavior 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. Performance is improved by removing guards entailed by the implicit "no earlier (sub)rule fired" precondition and optional type and mode declarations. A formal description 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.

The next section presents a short intuitive overview of guard simplification, illustrated with some examples. In section 3, a formal definition of the guard simplification transformation is given. Section 4 briefly deals with the implementation of the guard simplification analysis in the K.U.Leuven CHR compiler [8]. 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.

2 Overview

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 exhibit identical behavior when ω_r semantics is used. This is not an issue, since all recent CHR implementations use ω_r semantics.

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.

2.1 Guard simplification

When a simpagation rule or a simplification rule fires, some or all of its head constraints are removed. As a result, for every rule R_i, we know that when this rule is tried, any non-propagation rule R_j with $j < i$, where the set of head constraints of rule R_j is a (multiset) subset of that of rule R_i, did not fire for some reason. Either the heads did not match, or the guard failed. Let us illustrate this 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 [13], 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

We will now formalize the guard simplification transformation intuitively described above. 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 (not both empty), g_i (guard) is a conjunction of builtin constraints, and B_i (body) is a conjunction of constraints.

We assume all arguments of the CHR constraints in the head 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].

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

Definition 2 (Earlier subrule). *The rule R_j is an* earlier subrule *of rule R_i (notation: $R_j \prec R_i$) iff $j < i$ and the (renamed) constraints occurring in the head of R_j form a (multiset) subset of the head constraints of R_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 3 ("No earlier subrule fired"). *For every rule R_i, we define:*

$$nesr(R_i) = \bigwedge \left\{ (\neg(\theta_j \wedge g_j)) \mid R_j \prec R_i \text{ and } H_j^r \text{ is not empty} \right\}$$

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

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 4 (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*

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

Because of space limitations, we will simply formulate our correctness result without a proof. A detailed, but rather straightforward proof of the following theorem can be found in [12].

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'.*

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

4 Implementation

We have implemented guard simplification as a new compilation phase in the K.U.Leuven CHR compiler [8]. Essentially, the guard simplification phase does the following: For every rule R, head matchings are made explicit, $nesr(R)$ is constructed (the conjunction of the the negations of the guards of the earlier subrules $R_i \prec R$) and type information is added to this. Then any part of the guard entailed by this big conjunction is replaced by `true` – if its negation is entailed, it is replaced by `fail`. Finally, entailed head matchings are moved to the body if possible.

A separate entailment checking module has been written 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 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. A detailed description of our implementation of both guard simplification itself and the entailment checker can be found in [12].

In order to compare the generated code both with and without guard simplification, we present the Prolog code the CHR compiler generates for some example CHR program. In this fragment from a prime number generating program (taken from the CHR web site [14]):

```
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 (as in [7]) for the `filter/3` constraint:

```
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:

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

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. These benchmarks are available at [9]. For every benchmark, the results for a hand-written Prolog version are included, representing the ideal target code.

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.

Other CHR programs, like typical constraint solvers, where variable triggering occurs and the constraints are typically not never-stored, will

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.

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.

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. For larger CHR programs, the time complexity of the guard simplification compilation phase depends heavily on the number of earlier subrules for every rule. In extreme cases where this number is exceptionally large, the guard simplification phase tends to dominate the compilation time.

For an extensive discussion of the experimental results we refer the reader to [12].

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.

The guard simplification analysis is somewhat similar to switch detection in Mercury [6]. Switch detection is used in determinism analysis to check unifications involving variables that are bound on entry to a disjunction and occurring in the different branches. In a sense, switch detection is a special case of guard simplification. Similarities and differences are elaborated in [12].

Possibilities for future work include: improving the scalability of our implementation, adding support for declarations of certain properties of guards and using information derived in the guard simplification phase to enhance other analyses and to do program specialization on calls in the rule body.

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*. ACM Press, 2003.
4. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *20th International Conference on Logic Programming (ICLP'04)*, Saint-Malo, France, September 2004.
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 Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.
7. Christian Holzbaur and Thom Frühwirth. Compiling Constraint Handling Rules. In *ERCIM/COMPULOG Workshop on Constraints*, CWI, Amsterdam, 1998.
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. ISSN 0939-5091.
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. 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.
13. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, February 1995.
14. Various. 40 CHR Constraint Solvers Online, December 2004. Available at *http://www.pms.informatik.uni-muenchen.de/~webchr/*.